Formal Methods for Real-Time Requirements Engineering

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes
von

Georg Rock

Saarbrücken, Januar 2004

Kolloquium	30. Januar 2004
Vorsitz	Prof. Dr. Harald Ganzinger, Max-Planck-Institut für Informatik
Gutachter	Prof. Dr. (PhD) Jörg Siekmann, Universität des Saarlandes
	Priv.-Doz. Dr. Werner Stephan, Universität des Saarlandes
	Prof. Dr. Wolfgang Reisig, Humboldt-Universität zu Berlin
Dekan	Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes

Contents

Acknowledgements **ix**

Abstract **xi**

Kurzzusammenfassung **xiii**

Extended Abstract **xv**

Zusammenfassung **xvii**

1. Introduction **1**

2. Real-Time **7**
2.1 Real-Time 7
2.2 Specification of Real-Time 8
2.2.1 Temporal Logic as Basis 9
2.2.2 Expressing Timing Constraints 10
2.2.3 Real-Time Logics 12
2.3 Models of Time 13
2.3.1 Basics 14
2.3.2 Relations between Basic Time Elements 14
2.3.3 Time Model Used 16

3. Gasburner Scenario **19**
3.1 An Abstract Version 20

4. Formal Methods **23**
4.1 TLA - Temporal Logic of Actions 23
4.1.1 Fundamentals of TLA 23
4.1.2 Enabled Predicate 29

4.1.3 Fairness . . . 30
4.1.4 Further Definitions . . . 30
4.1.5 Real-Time in TLA . . . 32
4.1.6 Proofs in TLA . . . 34
4.2 VSE-II - Verification Support Environment . . . 37
4.2.1 Introduction . . . 37
4.2.2 Abstract Data Types . . . 38
4.2.3 Concurrent System Specifications . . . 40
4.2.4 Assumption Guarantee Specifications . . . 42
4.2.5 Structured Deduction . . . 43
4.2.6 A Simple Real-Time Example in VSE-II . . . 43
4.2.7 The Gasburner in VSE-II . . . 54
4.2.8 A more Complex Real-Time Example in VSE-II . . . 60
4.3 Hybrid Automata . . . 71
4.3.1 Syntax . . . 71
4.3.2 Semantics . . . 73
4.3.3 Property Specifications . . . 77
4.3.4 Example Proof with Hybrid Automata . . . 79

5. Integration of Hybrid Automata and VSE-II 83
5.1 Translating Hybrid Automata to VSE-II . . . 85
5.1.1 Translation Function . . . 85
5.2 Main Theorem . . . 96
5.2.1 Discretisation . . . 96
5.2.2 Granularity Change . . . 104
5.3 Nested Temporal Operators . . . 118
5.3.1 Leads-to Properties . . . 119
5.3.2 A more general Approach . . . 122
5.3.3 Invariants . . . 122
5.3.4 Generalisation . . . 123

6. Observer Methodology 125
6.1 Observer Models for Real-Time Properties . . . 126
6.2 A General Specification Scheme for Observer Models . . . 128
6.3 Gasburner as Real-Time Observation . . . 130
6.3.1 Underlying Datatypes . . . 131
6.3.2 Scheduling . . . 132

6.3.3 Environment . . . 133
6.3.4 Controller . . . 134
6.3.5 Observer . . . 136
6.3.6 The Gasburner as VSE-II Specification . . . 141
6.3.7 The Behaviour of the Real Gasburner . . . 142
6.3.8 Handling of the Constant c . . . 142
6.3.9 Refinement Proof . . . 143
6.3.10 Summary . . . 144
6.4 Storm Surge Barrier . . . 145

7. Other Methods **147**
7.1 Duration Calculus . . . 147
7.1.1 Syntax of the Duration Calculus . . . 148
7.1.2 Semantics of the Duration Calculus . . . 148
7.1.3 The Gasburner Example in the Duration Calculus . . . 152
7.2 Timed CSP . . . 155
7.2.1 Introduction to Timed CSP . . . 155
7.3 Automata . . . 165
7.3.1 Timed Automata . . . 165
7.3.2 UPPAAL Timed Automata . . . 168

8. Conclusion and Future Work **173**

A. Specification of the SVKO **177**

B. Real-Time in TLA and VSE-II **203**
B.1 Lossy Queue . . . 203
B.1.1 Non-lossy Version . . . 206
B.1.2 Reasoning about Time . . . 211
B.1.3 The Lossy Queue in VSE-II . . . 211
B.1.4 The Non–Lossy Real-time Version of the Queue in VSE-II . . . 213
B.2 Specification of the Lossy and the Timed Queue . . . 217

C. VSE-II Proof Rules **225**
C.1 Basic Rules . . . 225
C.1.1 Axiom . . . 225
C.1.2 $\Box$ Rules . . . 225
C.1.3 $\Diamond$ Rules . . . 226

C.1.4 $\mathcal{U}$ Rules 226
C.1.5 Step Rule 226
C.1.6 Induction Rule 227
C.1.7 Quantifiers 227
C.2 Definitions 227
C.2.1 Mapping into Next State 227
C.2.2 Filters selecting Invariant Formulae 228
C.2.3 Progress Conditions 230

Bibliography **231**

Index **241**

List of Figures

1. Introduction . . . 1
2. Real-Time . . . 7
3. Gasburner Scenario . . . 19
3.1 Gasburner . . . 19
4. Formal Methods . . . 23
4.1 Syntactical structure of TLA . . . 24
4.2 Freely Generated Data Types . . . 39
4.3 Development Graph . . . 44
4.4 `Func` Theory . . . 45
4.5 TLSPEC `System` . . . 45
4.6 TLSPEC `Safety` . . . 46
4.7 Proof of the Safety Property . . . 47
4.8 Proof Structure . . . 48
4.9 Development Graph: Concurrent Version . . . 49
4.10 `CompSys` Specification . . . 49
4.11 Proof of `export-axiom1` . . . 51
4.12 Proof Structure . . . 52
4.13 Development Draph of the Gasburner Specification . . . 54
4.14 Proof Tree of the Gasburner Scenario . . . 59
4.15 The gates of the ECS . . . 61
4.16 The Complete Scenario . . . 62
4.17 A possible Behaviour of a Real-Time System . . . 65
4.18 The VSE-II Development Graph of the ECS . . . 68
4.19 Gasburner as Hybrid Automaton . . . 72
4.20 Gasburner as HyTech Specification . . . 80
5. Integration of Hybrid Automata and VSE-II . . . 83
5.1 Relation: VSE-II- Hybrid Automata- Specification . . . 85
5.2 Non-Normalised Hybrid Automaton . . . 92
5.3 Normalised Hybrid Automaton . . . 92

5.4 Graphical Representation: Hybrid Automaton 93
5.5 Leaking Gas Burner as (δ) Discretised (Hybrid) Automaton 97
5.6 Discretised and Granularity Changed Leaking Gas Burner 105
5.7 Simple Hybrid Automaton 118
5.8 Copied hybrid automaton 120
6. Observer Methodology 125
6.1 Observer Models 127
6.2 General Scenario (Observer Model Instantiated) 129
6.3 Development Graph of the Real Gasburner 130
6.4 Theories Defining the Underlying Datatypes 131
6.5 Scheduling 132
6.6 A Scheduling Alternative 132
6.7 Specification of the Environment 133
6.8 Specification of the Controller 135
6.9 Refinement Relation 137
6.10 Specification of the Observer (part 1) 138
6.11 Specification of the Observer (part 2) 139
6.12 Gasburner Hybrid Automaton 141
6.13 Example Behaviour with Refinement Mapping 143
6.14 Hybrid Automaton for the Timer 145
6.15 Hybrid Automaton for Environment 145
6.16 Hybrid Automaton for Open Close Signal 146
7. Other Methods 147
7.1 Stimulus Response Automaton 168
8. Conclusion and Future Work 173
A. Specification of the SVKO 177
B. Real-Time in TLA and VSE-II 203
B.1 Lossy Queue 203
B.2 TLA Specification of the Lossy Queue 205
B.3 TLA Specification of the Non-Lossy Queue 208
B.4 Example Behaviour of the Timed Queue 209
B.5 Specification of the Lossy Queue in VSE-II 212
B.6 Specification of Time. 213
B.7 Specification of the Naturals with an Infinity Element. 214
B.8 Specification of the Timed Queue (1) 215
B.9 Specification of the Timed Queue (2). 216
B.10 Combined Specification of Time and the Timed Queue 218

B.11 Refinement Mapping 218
C. VSE-II Proof Rules 225

Acknowledgements

In the first place I want to thank my supervisors Werner Stephan and Andreas Nonnengart. Without their constant support, strong encouragement and patience this work would not have been possible. I had the pleasure to work with them for many years now and they guided me through the maze of science. The many fruitful discussions we had turned out to be a source of ideas that never seems to run dry.

I am also indebted to Jörg Siekmann for his confidence and outstanding support in writing this thesis. He gave me the opportunity to work in such a well known and excellent research group at DFKI. I also want to thank all the members of the group providing such a fruitful research environment. Special thanks to Bruno Langenstein for so many interesting discussions and helpful comments.

I also want to thank Wolfgang Reisig for his instant commitment to become one of the referees and for many helpful comments.

I benefited from the comments and help of many people and it seems impossible to list them without forgetting the one or the other. I thank them all and want them to know that I highly appreciate their accompany.

Last but not least I want to thank Monika for her encouragement and patience and in particular my parents and my sister who made it at all possible for me to study computer science. Words cannot express my thankfulness.

Saarbrücken, January 2004 Georg Rock

Abstract

Timed model checking turned out to be a very successful technique for the verification of real-time systems. In general, however, large-scale systems require more than a mere real-time perspective: They utilise, for example, Abstract Data Types and Fairness Aspects. VSE-II (Verification Support Environment) is a general tool which supports the design and the verification process of such large-scale systems.

The basic machinery within VSE-II is theorem proving rather than model checking and one of its underlying formalisms is close to TLA (Temporal Logic of Actions), i.e. it is based on linear discrete time.

In this thesis we develop a technique to perform an *exact* discretisation of dense real-time aspects, i.e. a discretisation that is not just an approximation but rather mirrors dense behaviour exactly. This discretisation is achieved without an explicit or implicit introduction of rational numbers.

With the help of the exact discretisation we define an embedding of Hybrid Automata into VSE-II such that model checking strategies for Hybrid Automata can be used in VSE-II. Vice versa, the embedding allows the model checking strategies to benefit from the proof work done in VSE-II.

This thesis introduces a general methodology for formal requirements analysis, namely *observer models*, that deals with particular perspectives on a system rather than with particular aspects of it. This way, different specialised approaches can be integrated and used to describe the overall system requirements. One such view, for example, is a real-time view which uses a new discretisation technique.

Kurzzusammenfassung

In der Verifikation von Realzeit-Systemen haben sich Model-Checking Verfahren bewährt. Im Allgemeinen kann man jedoch sagen, dass große industrielle Anwendungen nicht nur die Realzeit Dimension aufweisen. Sie bestehen vielmehr aus einer Vielzahl weiterer Dimensionen (Sichten) wie eine Informationsflusssicht oder eine Security-Sicht. Zur Spezifikation dieser Sichten werden beispielsweise Abstrakte Datentypen oder auch Fairness Aspekte verwendet. VSE-II (Verification Support Environment) ist ein Werkzeug, welches den formalen Entwicklungsprozess vom Design bis hin zur Verifikation solcher Anwendungen unterstützt.

Der Kern des VSE-II Werkzeugs ist ein interaktives Beweissystem, das auf einem Sequenzenkalkül basiert, der neben der Logik erster Stufe und Dynamischer Logik auch die Temporale Logik der Aktionen (TLA) beinhaltet. TLA beruht auf einem Zeitmodell, welches linear und diskret ist.

In dieser Arbeit beschreiben wir eine Technik, die eine *exakte* Diskretisierung von dichten Realzeitaspekten erlaubt, so dass das VSE-II System diese Aspekte mit den vorhandenen Verfahren und Regeln behandeln kann. Die Diskretisierung ist so definiert, dass sie nicht nur eine Approximation ist, sondern sie spiegelt vielmehr das dichte Verhalten exakt wider. Dies wird ohne die explizite oder implizite Einführung von rationalen Zahlen erreicht.

Mit Hilfe der exakten Diskretisierung wird eine Einbettung von Hybriden Automaten in VSE-II definiert, die es ermöglicht Teilbeweise, die von Modelcheckingverfahren für Hybride Automaten gefunden wurden, ohne weiteren Beweis in VSE-II zu verwenden und umgekehrt.

Weiterhin wird eine Methodologie zur formalen Anforderungsanalyse eingeführt, die verschiedene Sichten auf ein System und nicht nur verschiedene Aspekte eines Systems behandelt. Diese Methodologie, genannt *Observer Models*, ermöglicht die Integration unterschiedlicher spezieller Werkzeuge bzw. Verfahren zur Beschreibung der einzelnen Sichten und somit zur Beschreibung der gesamten Systemanforderungen. Eine solche Sicht stellt beispielsweise eine Realzeit-Sicht dar, welche auf der oben erwähnten Einbettung beruht.

Extended Abstract

"Formal methods" are the means to apply mathematical techniques and mathematical based reasoning in the analysis of computer software or hardware. Formal methods attempt to supply precise techniques for specifying and reasoning about a system's behaviour using techniques based mainly on the axiomatic method of mathematics. The basic idea of the axiomatic method is to specify properties which are required or assumed as axioms in some formal language, and to supply everything that can be used in reasoning about the specified artifact. To show that the specification has some property not explicitly mentioned in the axioms, it must be proved that this property is a logical consequence of the axioms. Thus a formal specification can be probed and tested by posing and proving theorems that we call *lemmata*. Similarly, to show that a system meets its requirements, we must prove that each requirement is derivable from the axioms specifying the system. All we need to know in a formal proof about the particular problem area is encoded in the statement of the theorem and the axiomatisation of its premises. The "truth" of the theorem can be established by syntactical operations according to the inference rules. The validity of the proof depends on the "form" of the proof and neither on the unrecorded knowledge of the problem area, nor on the intuition about what the theorem says. Yet knowledge and intuition are essential for the invention of useful axiomatisations and theorems, for their interpretations and for the discovery of proofs. In fact, most of the systems supporting the formal development of software operate with interactive theorem provers as for example VSE [52, 51, 54] or the B-Tool [11, 27, 31]. From experience we know that many of the problems arising in the verification of a system need induction. We might take this as a hint that, in general, theorems cannot be proved fully automatically. But if there is a possibility to prove a lemma automatically we should use this advantage and save time and effort.

The aim of this work is to introduce a combination of interactive and automatic verification techniques. Such a combination is established with respect to a certain real-time specification and verification technique (Hybrid Automata). To achieve an embedding of Hybrid Automata into VSE-II, that allows us to use the proof results based on model checking strategies for Hybrid Automata in VSE-II and vice versa without doing any further proof work, a technique is introduced in this thesis that allows us to perform an *exact* discretisation of dense real-time aspects

such that VSE-II can cope with them. This discretisation is defined in such a way that is not just an approximation but rather mirrors dense behaviour exactly and that without an explicit or implicit introduction of rational numbers.

The author does not intend to favour the one or the other approach to deal with real-time, especially when the question about interactive theorem provers used in tools like VSE-II [52, 51, 54] or model checkers [72, 50] arises. All of these approaches have their advantages and disadvantages. In this work we try to exploit the advantages of these approaches and to incorporate them in an integrated methodology that we call *observer models*. The presented methodology is not limited to the area of real-time systems. It rather gives the possibility to integrate different techniques or tools to handle different perspectives of a system specification where real-time could be one of the perspectives. Other perspectives could be information flow or security. The methodology is built in such a way that a formal system engineer has the possibility to work purely interactively (if necessary), purely automatic (if possible) or in an interleaved way. Thus, parts of the problem can be solved automatically with push button technologies. These parts can now be used in the interactive proof components. But also, the interactively proven theorems can be used to support the automatic techniques by providing additional information.

We claim that it is not always necessary to invent a new logic together with a new calculus and build a new tool around all this in order to establish such an integration. This work rather exploits methods and tools available on the market, takes the parts that fit best and makes a methodology available to integrate them syntactically as well as semantically. This leads to synergetic effects so that the resulting method is more than just the sum of its components.

The thesis first introduces the logics that are used to implement real-time specifications and to realize the mentioned methodology: VSE-II (Verification Support Environment), TLA (Temporal Logic of Actions) and Hybrid Automata. Then an approach to handle real-time systems with VSE-II is illustrated on a real Storm Surge Barrier that is completely formalised in VSE-II. A generalisation of this approach results in the new methodology *observer models* that is illustrated with the help of VSE-II, which represents (but is not limited to) the basic specification and verification engine used within observer models and Hybrid Automata. For this purpose we have defined a syntactical and semantical embedding of Hybrid Automata in VSE-II that allows us to exploit automatically computed results with respect to Hybrid Automata in VSE-II and interactively proven results with respect to VSE-II in Hybrid Automata.

Finally, a few other formal approaches to deal with real-time together with a summary of the work and possibilities for future extension are presented.

Zusammenfassung

"Formale Methoden" verstehen sich als das Werkzeug zur Anwendung von mathematischen Techniken und mathematisch basierten Beweisverfahren zur Analyse von Software und Hardware. Formale Methoden stellen präzise Techniken zur Spezifikation und Verifikation von Systemen zur Verfügung, die auf der axiomatischen Methode der Mathematik basieren. Die Grundidee der axiomatischen Methode ist es die erwarteten oder angenommenen Eigenschaften als Axiome in einer formalen Sprache zu spezifizieren und alles notwendige, welches zum Schlussfolgern über das spezifizierte Artefakt benötigt wird, zur Verfügung zu stellen. Um Eigenschaften nachzuweisen, die nicht explizit in den erwähnten Axiomen vorkommen, müssen diese als logische Folgerungen aus den Axiomen abgeleitet werden. Somit kann eine formale Spezifikation durch das Aufstellen von Theoremen und dem anschließenden Beweisversuch der entsprechenden Theoreme, die *Lemmas* genannt werden, getestet und somit präzisiert werden. Um nachzuweisen, dass ein System die gestellten Anforderungen erfüllt, muss auf ähnliche Weise gezeigt werden, dass die Anforderungen aus den Axiomen ableitbar sind, die das Verhalten des Systems spezifizieren. Alles, was in einem formalen Beweis in einem speziellen Gebiet benötigt wird, ist in der Axiomatisierung der Prämissen und der Formulierung des zugehörigen Theorems enthalten. Die "Gültigkeit" eines Theorems wird durch syntaktische Manipulationen, die nach vorgegebenen Ableitungsregeln durchgeführt werden, bestimmt. Die Gültigkeit des Beweises hängt von der Form des Beweises und nicht von unbenutztem Wissen aus der Problembeschreibung oder von der Interpretation des Theorems ab. Trotzdem sind Intuition und das genaue Kennen des Problems essentiell zur Erstellung einer sinnvollen Axiomatisierung und ihrer Theoreme. Die meisten Systeme, die die formale Entwicklung von Software unterstützen, basieren auf einem interaktiven Beweissystem, wie beispielsweise VSE [52, 51, 54] oder das B-Tool [11, 27, 31]. Die Erfahrung zeigt, dass viele der zu beweisenden Eigenschaften nur mit Hilfe von Induktion bewiesen werden können. Dies könnte man nun als Hinweis darauf sehen, dass Theoreme, die im Gebiet der Softwareverifikation auftauchen, nicht voll automatisch bewiesen werden können. Aber, wenn es die Möglichkeit gibt ein Lemma automatisch zu beweisen, sollte man diesen Vorteil nutzen und Zeit auf Aufwand sparen.

Das Ziel dieser Arbeit ist es eine Kombination von interaktiven und automatischen Verifikationstechniken zu realisieren. Eine solche Kombination ist zwischen

VSE-II und einer Realzeit Spezifikations- und Verifikationstechnik (Hybride Automaten) erarbeitet worden. Um diese Einbettung von Hybriden Automaten in das VSE-II System in der Art zu ermöglichen, dass Beweisergebnisse, die aus Model-Checking Strategien für Hybride Automaten resultieren in VSE-II zu nutzen und umgekehrt, ist eine Technik in dieser Arbeit entwickelt worden, die eine exakte Diskretisierung von dichtem Realzeitverhalten ermöglicht, so dass diese mit VSE-II behandelt werden können. D.h., es ist eine Diskretisierungstechnik definiert worden, die nicht nur eine Approximation darstellt, sondern eher das dichte Verhalten *exakt* widerspiegelt und dies ohne die explizite oder implizite Einführung der rationalen Zahlen erreicht.

Wenn die Frage nach interaktiven Beweissystemen, wie sie beispielsweise in VSE-II [52, 51, 54] verwendet werden, oder Model-Checkern, wie beispielsweise in [72, 50] beschrieben, gestellt wird, favorisiert der Autor nicht den einen oder anderen Ansatz zur Behandlung von Realzeit. All diese Ansätze haben ihre Vorteile aber auch ihre Nachteile. In dieser Arbeit versuchen wir die Vorteile beider Ansätze zu nutzen und sie in einer integrierten Methodologie, die wir *Observer Modelle* genannt haben, zu vereinen. Die vorgestellte Methodologie ist nicht auf die Behandlung von Realzeit-Systemen beschränkt. Sie erlaubt vielmehr die Integration verschiedener Techniken oder Tools, die verschiedene Sichten auf ein System repräsentieren, wobei die Realzeit-Sicht eine spezielle Sicht darstellt. Andere Sichten sind beispielsweise eine Informationsfluss-Sicht oder eine Security-Sicht. Die Methodologie ist so entworfen, dass ein Systemingenieur die Möglichkeit hat interaktiv (wenn nötig), vollautomatisch (wenn möglich) oder mit beiden Ansätzen zu arbeiten. Somit können Teile des Problems mit "Push-Button" Technologien gelöst werden, die wiederum im interaktiven Beweis Verwendung finden können. Ebenso können die interaktiv bewiesenen Theoreme in den automatischen Verfahren benutzt werden, in dem zusätzliche Informationen zur Verfügung gestellt werden.

Es muss nicht immer eine neue Logik zusammen mit einem neuen Kalkül entwickelt werden, welche dann in einem neuen Tool integriert werden, um eine solche Methodologie zu realisieren. In dieser Arbeit werden vielmehr Methoden und Tools, die verfügbar sind verwendet, und die Teile, die geeignet sind, werden syntaktisch wie auch semantisch integriert. Dies resultiert in einer Synergie, wobei die resultierende Methode mehr ist als nur die Summe ihrer Komponenten.

In dieser Arbeit werden zunächst die Logiken eingeführt, die sowohl zur Spezifikation von Realzeit-Systemen verwendet werden als auch zur Realisierung der genannten Methodologie: VSE-II (Verification Support Environment), TLA (Temporale Logik der Aktionen) und Hybride Automaten. Hiernach wird eine Methode zur Behandlung von Realzeitsystemen an dem Beispiel eines Sturmflutwehres illustriert, welches komplett in VSE-II spezifiziert wurde. Die Verallgemeinerung dieser Methode resultiert in der bereits oben erwähnten neuen Methodologie der *Observer Modelle*. Diese werden mit Hilfe von VSE-II, welches die grundlegende Spezifikations- und Verifikationsmaschinerie zur Verfügung stellt, und Hybriden

Automaten illustriert. Wir haben eine syntaktische und semantische Einbettung von Hybriden Automaten in VSE-II definiert, die es uns ermöglicht, automatisch berechnete Ergebnisse in einem interaktiven Beweis zu verwenden und umgekehrt.

Zum Abschluss der Arbeit werden einige weitere formale Ansätze zur Behandlung von Realzeit vorgestellt. Die Arbeit endet mit einer Zusammenfassung und der Erwähnung von zukünftig möglichen Erweiterungen.

1

Introduction

Before considering special terminologies and special logics for expressing real-time properties, we like to present an informal view of what is called formal methods and the role of real-time within formal methods.

"Formal methods" are the means to apply mathematical techniques and mathematical based reasoning to the analysis of computer software or hardware. It is fully accepted and often even required that engineers in established disciplines use mathematical models or mathematical reasoning to predict or to simulate the behaviour of the systems they have built. For example, in today's car manufacturing industry it is common not only to develop the design of a new car on a computer but the whole car including all of its safety functionalities. Even the behaviour of a car in critical driving and crash situations is computed by special programs, which implement in some sense a "model" of the car and its behaviour. The real tests made later in the development of a new car are performed primarily to validate accuracy of the model and to measure final performance parameters. Another example can be found in the development of airplanes. The computational design of the wings of an airplane requires mathematical models to predict their behaviour. In these areas mathematical models are not only fully accepted, they are a necessity.

But this process of reasoning with mathematical models is not so widespread in the development of software or hardware. In this field prototyping and testing still remain the principal methods to explore designs and validate implementations. These methods or strategies certainly have their justification, but their usefulness depends on the area in which they should be applied. If no particular safety or

security requirements are demanded these more or less informal methods can be applied since they are faster and (in general) less expensive. Some people would say now that informal methods are always faster and first of all less expensive with respect to the development time and costs. At first glance, this may be the case, but when we consider the whole life-cycle of a product from the development through marketing to the maintenance phase, then things change. Often the maintenence costs are not considered in the costs of the project which are computed right after the project is finished and the system is sold.

The informal methods are possibly dangerous as they only provide partial coverage of the range of behaviours that a piece of software may exhibit. The problem is that computer hardware and software are discrete systems. Their behaviour is determined by succession of discrete state changes. The succession of states does not need to produce a behaviour that varies smoothly or continuously. Instead, it can exhibit discontinuities and abrupt transitions. The ability to select alternative courses of action is a source of power and flexibility provided by computer systems, but it is also the reason why their behaviour is hard to predict and to validate. Tests merely provide information on the state sequences which are actually examined. Without continuity there is little reason to extrapolate the behaviour from tested cases to untested ones. The caveats of testing are becoming particularly acute in the case of distributed and real-time systems, such as the embedded systems used for spacecraft control. Coordination of the various subsystems and redundant components can depend on delicate timing relationships which can be difficult and very complex to fully explore with tests. Consequently, a large number of major spacecraft anomalies occur due to timing problems.

The difficulty of predicting and testing timing behaviour results from the possibility that hardware failures may occur at any time. For example, some observations were lost when the heavy radiation environment at Jupiter caused one of the clocks on the Voyager spacecraft to jump several seconds [25]. More robust clock synchronisation was provided for later encounters, but the mechanisms of fault tolerance and recovery are complex and add more difficulty of validating behaviour by testing. Thus, for example, the failure in the first attempt to launch the Space Shuttle was due to a synchronisation problem of the redundant computers [40].

Formal methods confront the discrete behaviour of computer systems by using discrete mathematics as a model. Discrete mathematics builds directly on mathematical logic and proofs of theorems take the place of the numerical calculations which, as mentioned before, are familiar in most other engineering disciplines. That is, instead of using a mathematical model to calculate a value for some numerical quantity, in formal methods for computer science we prove theorems about the modelled behaviour. In formal methods the problem of discontinuities and the unsoundness of extrapolating from a finite number of tests are overcome using methods of proof based on mathematical induction, so that an infinite (or finite but very large) number of possible behaviours is fully covered in a finite proof.

Furthermore, formal methods offer much more to computer science than just

proofs of correctness for programs and digital circuits. Many of the problems in software and hardware design are due to imprecision, ambiguity, incompleteness, misunderstanding and just plain mistakes in the statement of top-level requirements, in the description of intermediate designs, or in the specifications of components and interfaces. Some of these problems can be attributed to the difficulty of describing large and complex artifacts in natural language, but others may be due to the lack of a suitably precise conceptual framework for describing and reasoning about certain classes of behaviours. Real-time systems seem to fall into the latter class. With the exception of scheduling theory, there is little foundation for an engineering discipline of real-time systems.

Formal methods attempt to supply precise techniques for specifying and reasoning about a system's behaviour using techniques based mainly on the axiomatic method of mathematics. The basic idea of the axiomatic method is to specify properties required or assumed as axioms in some formal language, and to provide everything that can be used in reasoning about the artifact. To show that the specification has some property not explicitly mentioned in the axioms, it must be proved that this property is a logical consequence of the axioms. Thus, a formal specification can be probed and tested by posing and proving theorems that we call *lemmata*. Similarly, to show that a system meets its requirements, we must prove that each requirement is derivable from the axioms specifying the system. All we need to know in a formal proof about the particular problem area is encoded in the statement of the theorem and the axiomatisation of its premises. The "truth" of the theorem can be established by syntactical operations according to the inference rules. The validity of the proof depends on the "form" of the proof and neither on the unrecorded knowledge of the problem area, nor on the intuition about what the theorem might say. Yet knowledge and intuition are essential to the invention of useful axiomatisations and theorems, to their interpretations and to the discovery of proofs. In fact, most of the systems, which support the formal development of software, operate with interactive theorem provers as for example VSE [52, 51, 54] or the B-Tool [11, 27, 31]. From experience we know that most of the problems in the verification of a system need induction. We might take this as a hint that, in general, theorems cannot be proved fully automatically.

Above all, if we use formal methods, then the specification and the reasoning about a system is a repeatable exercise, whereas this is not the case in informal methods. Thus, if errors were found during the specification or the verification process, then these errors can be checked by examining exactly what has been done before.

Despite the arguments mentioned above, formal methods are still not widespread in the development of (software) systems. The reason may be the combination of knowledge that a system engineer must have in applying a formal method to a problem. We will exemplary state what this means using the VSE-tool as an example for a tool supporting the whole formal development process from specification through deduction/proving to maintenance of the development. The engineer has to learn

the input language which consists of a language describing temporal logic components as well as abstract datatypes[1]. The writing of specifications is rather easy, the difficult part is to specify the system the engineer has in mind, i. e., using the syntactical constructs to specify the desired semantical behaviour of the system. The engineer has to learn the semantics of the logics and the operators used in the specification language[2]. At this point, a programmer's work using an ordinary programming language is usually finished. After a successful compilation the program is tested. The formal system engineer now has to make a type check that checks whether the specification is type-correct and then to prove that the specification fulfils the desired properties. This is done by transforming the specification into its logical representation. During this transformation process proof obligations arise that have to be proved. The engineer has to know about the different logics used in the tool and their corresponding proof rules. There are heuristics that help but in general he has to know the rules for temporal logic, first-order logic and dynamic logic, if datatype refinements occur. With this more technical knowledge, the engineer has to apply the rules in order to prove that the specified system fulfils the desired properties. In the course of such a proof it is often necessary to invent invariants that help in proving the properties. Thus, finding such a proof is far from being a trivial task and cannot be automated in general.

In considering real-time requirements on systems things do not get easier and it is even more important to have a methodology realizing a smooth transition from informal or semi-formal to formal development techniques.

The aim of this work is to introduce a combination of interactive and automatic verification techniques. This has been done with respect to special real-time specification and verification techniques. But the approach is not limited to this area. For such an integration we claim that it is not always necessary to invent a new logic together with a new calculus and build a new tool around all this. This work rather exploits methods and tools available on the market, takes the parts that fit best and integrates them syntactically as well as semantically. This results in synergetic effects so that the resulting method is more than just the sum of its components.

At this point I do not necessarily favour the one or the other approach to handle real-time, especially when the question about interactive theorem provers used in tools like VSE-II [52, 51, 54] or model checkers [72, 50] arises. All of these approaches have their advantages and disadvantages. In this work we take a more general view to apply the paradigm presented above to exploit the advantages of these approaches and to incorporate them into an integrated methodology. Such a methodology gives a formal system engineer the possibility to work purely inter-

[1] In the case of VSE-II this is not difficult, since there is a syntax oriented editor implemented as part of the front end of the tool. Using this editor writing syntactical correct expressions is a trivial task.

[2] Experience shows that this part is the first difficult hurdle an engineer has to overcome using formal methods.

actively (if necessary), purely automatically (if possible) or in an interleaved way, i.e. parts of the problem can be solved automatically with push button technologies and these parts can then be used in the interactive search for a proof. But also, the interactively proven theorems can be used to support the automatic techniques by providing additional information.

The thesis is organised in six parts. The first part contains a short survey on the possibilities to handle real-time formally. In addition, the time models used in this work are presented. The second part introduces a real-time example that is analysed in all the formal development techniques presented. This allows us to compare the various approaches. The third part is concerned with the description of the formal methods used in this work. We then turn our attention to the integration of the formal approaches such that we can use the one to support the other. The fifth part presents a methodology for (real-time) requirements engineering and exploits the results from the fourth part. The final part of this work consists of Chapters 7, 8 and the Appendixes A, B and C. There we present other formal approaches to handle real-time, summarise this work and discuss future possibilities. Detailed VSE-II specifications of systems mentioned in the earlier chapters conclude this thesis.

2

Real-Time

2.1 Real-Time

A real-time system depends not only on the values of its outputs but also on the times at which they are produced. Generally, a real-time system executes a collection of tasks that are subject to *deadline*, *periodic* or *aperiodic* constraints.

A task has to produce its output before its deadline. Periodic constraints are activated for example by a timer whereas aperiodic constraints can be activated by some external event at any time. Strict constraints arise for example in some systems dependent on I/O devices that require precise timing between inputs and outputs [68] and in some systems it can be important not to produce a result too early, so that deadlines are better treated as intervals than points in time.

Hard real-time exists in systems in which the passing of a deadline time constraint can have severe consequences. For soft real-time constraints there is still some use of the results of a task that has missed its time constraint [2]. What we can see is that real-time systems, that are considered to be correct or useful, must work within their specified time constraints, either without exception or with high probability [44].

In the development of real-time systems there are two main issues:

1. The derivation of timing constraints.
2. The construction of a system structure (particularly a scheduling regimen) that guarantees the satisfaction of those constraints.

The second of these (especially scheduling theory) has received the most attention, but it can be argued that the first is more fundamental.

A real-time system usually contains some qualitative timing requirements, such as constraints on the simultaneity and ordering of events. For example, in a railroad crossing scenario, cars and trains should not be on the railroad at the same time for obvious reasons. Often qualitative requirements are not enough, so there are quantitative requirements. For instance, the gasburner that should close the valve 1 second after leaking and there is no new ignition for at least 30 seconds, or a control loop that must execute at a rate of 40 Hz with no more difference than 2 msec.

A question that arises in this context is whether real-time is needed to specify or to reason about qualitative timing requirements. Since simultaneity and ordering can be expressed without explicit use of real-time, the question can be answered with "no": As an example consider the railroad crossing scenario where one requirement says that a train and a car should not be at the crossing at the same time. This can be expressed with the (pseudo) formula[1]:

$$\mathit{always}(\neg\, \texttt{at\_crossing(car)} \vee \neg\, \texttt{at\_crossing(train)})$$

The ordering of events can be handled as in the following example. `Event1` should happen always before `Event2`. This requirement can be expressed by the formula:

$$\mathit{always}(\texttt{Event1} \Rightarrow \mathit{eventually}(\texttt{Event2})) \wedge$$
$$\mathit{always}(\mathit{always\text{-}in\text{-}past}(\texttt{Event2} \Rightarrow \mathit{eventually\text{-}in\text{-}past}(\texttt{Event1}))$$

Until now, we have used temporal expressions only informally using natural language notions that everyone can understand or at least reasonably interpret.

The next section shows how real-time is introduced into temporal logics including the different operators and what decisions can be taken depending on the problem to be handled.

2.2 Specification of Real-Time

Temporal logics do not provide the means to reason about time in a direct or quantitative sense. Temporal logic is the designation for a modal logic whose operators are however interpreted in a temporal manner. The main operators $\Box$ and $\Diamond$ are interpreted as "always" and $\Diamond$ as "eventually", respectively. Pnueli [82] was one of the first to propose temporal logic to specify the behaviour of reactive systems. With temporal logic *qualitative* temporal requirements can be expressed including invariants, precedence and responsiveness [83]. But traditional temporal operators

[1] Formulas written in emphasised style represent temporal operators. Since we have not introduced these operators yet we cannot use them here.

cannot express quantitative temporal requirements or hard real-time requirements as timing deadlines for the behaviour of reactive systems. One possibility is to extend temporal logics in order to deal with quantitative timing behaviours. Lamport showed in [4] that another way is to use an explicit time variable (see 2.2.2.3 and 2.2.3). Before we proceed we first present some of the design decisions that can be made in choosing the underlying temporal logic. We can classify most systems of TL (Temporal Logics) used for reasoning about concurrent programs along a number of axes:

- propositional versus first-order,
- branching versus linear,
- points versus intervals,
- discrete versus continuous and
- past versus future tense.

We will concentrate in our work on two approaches which are linear time, discrete, compositional future tense (VSE-I [52], VSE-II [54]) and branching time, continuous, compositional future tense (Hybrid Automata [47]). However, to give the reader an idea of the wide range of possibilities in formulating a system of Temporal Logic, we describe various alternatives in more detail in the next section.

2.2.1 Temporal Logic as Basis

If we decide to use temporal logic we have to make decisions concerning the kind of temporal logic to use. It might be point or interval based or its operators talk about the past or the future or both. Some additional possibilities are:

- One decision to be taken is whether we need first-order logic or just the propositional fragment. Clearly, this choice depends on the system/problem to be specified and the data domains over which the reactive systems operate.
- Many temporal logics used in the specification and verification of reactive systems are divided into linear-time and branching-time logics.
  - *Linear time* logics are interpreted over linear structures of states. Every state sequence represents an execution of the reactive system under consideration. An example of a linear time logic is TLA [66]. The typical response property that "every inquiry must be followed by an answer" is defined in TLA by the formula

$$\Box(P \rightarrow \Diamond Q)$$

It says that in any possible behaviour in which inquiry P happens, i.e. is observed in some state, there must be a state at some later point in which the answer Q is given, i.e. Q is observed in this state.

Real-Time extensions of PTL [58] introduce time bounded temporal operators. They can be used to express the time-bounded version of the response property that says that "every inquiry must be followed by an answer within n time units". It is given for example by the following formula:

$$\Box(P \rightarrow \Diamond_{\leq n} Q)$$

– *Branching time* temporal logics, on the other hand, are interpreted over tree structures of states. Every tree represents a reactive system where every path in the tree represents an execution sequence of the system. Examples of branching time logics are CTL [41, 38] and CTL* [35]. The above mentioned property that "every inquiry must be followed by an answer" is expressed in CTL by the formula:

$$\forall\Box(P \rightarrow \forall\Diamond Q)$$

This formula asserts that in any behaviour, i.e. in all branches and in every node on that branch, if P is observed then in all possible continuations of that branch Q is observed at some later point.

Real-Time variants of CTL allow us to put time bounds on the distance between the observation of P and the observation of Q.

- The next decision is concerned with the choice of the temporal operators. The temporal operator most typically employed is the *until* operator (It can be used to define the *always* operator [70]). Some temporal logics exclude the *next* operator to ensure that the formulae of the logic remain *invariant under stuttering* [66]. Other logics contain past operators such as *since*. In branching time logics path quantifiers are used whereas in linear time logics these path quantifiers are not available, in fact, every formula is interpreted over all paths.

2.2.2 Expressing Timing Constraints

In defining the syntax of an extension of a temporal logic we have to decide how to incorporate timing constraints (timing requirements) into the logic. Among all the possibilities available we chose three of the principal techniques in this work that are taken from [20].

2.2.2.1 Bounded Temporal Operators

A common idea to introduce real-time in a temporal logic is to extend the syntax by bounded temporal operators which replace the unrestricted ones. Examples for

such operators are $\Box_{[1,2]}P$ and $\Diamond_{[1,2]}Q$. The first formula expresses that P holds within 1 to 2 time units . The second formula asserts that Q holds somewhere between 1 to 2 time units. This approach of introducing real-time is proposed in [60, 59, 85] whereas [30] can be viewed as a precursor.

There are many interesting articles about bounded operators that treat the expressiveness and complexity of bounded operators [37, 13, 19, 18] and their applications.

2.2.2.2 Freeze Quantification

The bounded-operator notation can relate only adjacent temporal contexts. Consider, for instance, the property that "every inquiry P is followed by a response Q and, then, by another response R such that R happens within 5 time units of the inquiry P". There is no direct way of expressing this timing requirement using time-bounded operators. This shortcoming of bounded temporal operators can be remedied by extending temporal logic with explicit references to the times of temporal contexts. We discuss two such methods. In this subsection, we access the time of a state through a quantifier, which binds ("freezes") a variable to the corresponding time; in the next subsection, we access the time of a state through a (dynamic) state variable.

The idea of freeze quantification was introduced and has been analysed in [21, 45, 12, 46]. We present the propositional linear-time case here for completeness. The freeze quantifier $x.$ binds the associated variable x to the time of the current temporal context: the formula $x.\Phi(x)$ holds at time t if and only if $\Phi(t)$ does. Thus, in the formula $\Diamond y.\Phi$, the time variable y is bound to the time of the state at which Φ is "eventually" true. By admitting atomic formulae that relate the times of different states, we can write the time-bounded response property as

$$\Box x.(P \rightarrow \Diamond y.(Q \wedge y \leq x + 3))$$

We read this formula as "in every state with time x, if P holds, then there is a later state with time y such that Q holds and y is at most $x + 3$". The non-local property that "every inquiry P is followed by a response Q and, then, by another response R within 5 time units of the stimulus P" may be specified by the formula

$$\Box x.(P \rightarrow \Diamond(Q \wedge \Diamond z.(R \wedge z \leq x + 5)))$$

Freeze quantifiers allow us to refer to times that are associated with states. Consequently, the freeze quantifier $x.$ behaves differently from standard first-order quantifiers over time; it is, for example, its own dual:

$$\neg(x.\Phi) \leftrightarrow x.(\neg\Phi)$$

Since the expressive power of a modal logic with freeze quantification lies, in general, between the expressive power of the corresponding propositional and first-

order modal logics, a logic with freeze quantification is referred to as half-order [45].

2.2.2.3 Explicit Time Variable

The third possibility we sketch here is to introduce real-time by an explicit clock variable. This approach is based on standard first-order temporal logic.

The syntax simply uses a variable *now* as the *clock variable* that can be incremented in a computation step. In addition, we allow quantification over global variables on the time domain. An example for a time-bounded response property is the following:

$$\forall x.\Box((P \wedge now = x) \rightarrow \Diamond(Q \wedge now \leq x + 3))$$

This formula asserts that whenever P holds and *now* has the value of x then eventually Q will hold and the value of *now* is less or equal to $x + 3$. Examples of this method describing real-time can be found for example in [84, 4].

This approach has some similarities to the freeze quantification approach described earlier as variables like x catch a point in time given by the variables *now*. That way, this point in time can be related to later on in the formula.

2.2.3 Real-Time Logics

We now present some real-time temporal logics and point to the decisions that have been taken within these logics concerning the various choices we have sketched before. We divide the logics we consider in linear time and branching time logics:

Linear-time logics The logics MTL (*Metric Time Logics*) [19], TPTL (*Timed propositional temporal logic*) [21], RTTL (*Real-Time Temporal Logic*) [81], XCTL (*Explicit Clock Temporal Logic*) [42] and TLA (*Temporal Logic of Actions*) [66] are linear time logics.

- MTL is a propositional bounded-operator logic. Its temporal operators are time-bounded versions of *until*, *next*, *since* and *previous* (the past dual of *next*) operators.

- TPTL is a propositional half-order logic [20] that uses the future operators *until* and *next*. The atomic timing constraints contain the primitives $\leq$ (comparison), $\equiv_c$ (congruence modulo a constant), and $+c$ (addition by an integer constant).

- RTTL is a first-order explicit-clock logic with no restrictions on the assertion language for atomic timing constraints.

- XCTL is a propositional explicit-clock logic whose assertion language for timing constraints allows for comparison and addition. The timing constraints of XCTL are richer than those of the previously mentioned logics, which prohibit the addition of time variables. XCTL, however, prohibits explicit quantification over time variables. Thus, all global time variables are universally quantified.

- TLA is a first-order explicit clock logic. It is described in detail in Section 4.1.

Most of the logics mentioned above assume integer time whereas TLA is not restricted to a special time domain. It is rather a logic in which the user specifies the time domain according to the requirements. If, for example, the system is analog, as e.g. the steam boiler [10], then the reals can be chosen as time domain, whereas in other examples the integers or the naturals might be more suitable.

Branching-time logics The Real-Time Computation Tree Logic (RCTL) [37] is a propositional branching-time logic for synchronous systems. It is a bounded-operator extension of CTL.

The logic TCTL (Timed Computation Tree Logic) [13] is a propositional branching-time logic. It is also a bounded-operator extension of CTL with a less restrictive semantics than RTCTL.

We have presented possibilities concerning the choice of a temporal logic and how to extend this temporal logic in order to to express quantitative timing constraints.

However, we have not specified exactly the time model which underlies the logic. Most of the logics mentioned here use integers as time domain but others take the reals or the integers as the time domain. In the next section we present a short overview of the models of time.

2.3 Models of Time

The models of time can be subdivided along the following lines:

- *basics* and

- *relations*.

In the *basics* we look at several choices that can be made with respect to basic time elements. The *relations* part points out what relations we might have between basic time elements.

2.3.1 Basics

There are three main subdivisions of the basic time elements. The basic elements can be points, intervals or events.

2.3.1.1 Points as Basic Time Elements

Points in time might be indicated by a clock which counts time with a certain measure so that every point in time is eventually reached by the clock. When we think of a point in time we imagine the state of the world or that part of the world we are interested in and which is observable at this point. Intervals can be simulated by convex sets of points.

2.3.1.2 Intervals as Basic Time Elements

Often intervals are taken to be a convex subset of some ground set, say R, the real numbers. But if intervals are the basic time elements, then there are some difficulties such as determining the order of two actions taking place in the same interval or determining the position of a moving object in an interval. Intervals are especially suited as basic elements in continuous time models. Furthermore, intervals can be used as well as points as basic time elements since we can simulate points simply by indivisible intervals.

2.3.1.3 Events as Basic Time Elements

Events and actions (actions are atomic events), where events are built from actions, do not involve proper time. In fact they constitute time. Thus, time becomes a derived notion, and does not exist explicitly. In real–time systems, it is often the case that time is imitated by an event, for example the ticking of a clock.

In this thesis we say that points are taken as the basic time elements and that intervals can be viewed at as constructed from convex sets of points and events take place over intervals. The following considerations are with respect to points as basic time elements.

2.3.2 Relations between Basic Time Elements

There are two obvious relations on time, independent of the choice of the basic time elements: *equality* ($=$) and the *precedence* ($<$) relation. Equality is a natural relation in any logical structure and the precedence relation is characteristic for models of time. There are some other relations which seem to be natural and which can be derived from equality and precedence:

- Inequality: $Ineq(x, y) \Leftrightarrow \neg x = y$

- Later than: $Lt(x, y) \Leftrightarrow y < x$
- Betweenness: $Bet(x, y, z) \Leftrightarrow y < x < z \vee z < x < y$

Many possible properties of the precedence relation $<$ are considered in temporal logic. We give a formulation of the properties in first order logic, if possible:

Transitivity :	$\forall x, y, z.\, (x < y \wedge y < z \rightarrow x < z)$
Irreflexivity :	$\forall x.\, \neg x < x$
Asymmetry :	$\forall x, y.\, (x < y \rightarrow \neg y < x)$
Linearity :	$\forall x, y.\, (x < y \vee x = y \vee y < x)$
L − Linearity :	$\forall x, y, z.\, ((y < x \wedge z < x) \rightarrow (y < z \vee y = z \vee z < y))$
R − Linearity :	$\forall x, y, z.\, ((x < y \wedge x < z) \rightarrow (y < z \vee y = z \vee z < y))$
F − Directedness :	$\forall x, y, z.\, (x < y \wedge x < z \rightarrow \exists w.\, (y < w \wedge z < w))$
P − Directedness :	$\forall x, y, z.\, (y < x \wedge z < x \rightarrow \exists w.\, (w < y \wedge w < z))$
P − Serial :	$\forall x.\, \exists y.\, y < x$
F − Serial :	$\forall x.\, \exists y.\, x < y$
Density :	$\forall x, y.\, (x < y \rightarrow \exists z.\, (x < z \wedge z < y))$

Some of the relations given here have corresponding axiom schemata in temporal logics. If we take the above properties as properties of the reachability relation of a Kripke structure [61], then we have the following correspondence to axiom schemata:

Transitivity :	$\Box\Phi \rightarrow \Box\Box\Phi$
Irreflexivity :	*not axiomatisable* [95]
Asymmetry :	*not axiomatisable* [95]
Linearity :	*not axiomatisable* [95]
L − Linearity :	$\boxed{\text{P}}(\boxed{\text{P}}\Phi \rightarrow \Psi) \vee \boxed{\text{P}}(\boxed{\text{P}}\Psi \rightarrow \Phi)$
R − Linearity :	$\boxed{\text{F}}(\boxed{\text{F}}\Phi \rightarrow \Psi) \vee \boxed{\text{F}}(\boxed{\text{F}}\Psi \rightarrow \Phi)$
F − Directedness :	$\diamondsuit\!\!\!\!\!\!\text{F}\ \boxed{\text{F}}\Phi \rightarrow \boxed{\text{F}}\diamondsuit\!\!\!\!\!\!\text{F}\ \Phi$
P − Directedness :	$\diamondsuit\!\!\!\!\!\!\text{P}\ \boxed{\text{P}}\Phi \rightarrow \boxed{\text{P}}\diamondsuit\!\!\!\!\!\!\text{P}\ \Phi$
P − Serial :	$\boxed{\text{P}}\Phi \rightarrow \diamondsuit\!\!\!\!\!\!\text{P}\ \Phi$
F − Serial :	$\boxed{\text{F}}\Phi \rightarrow \diamondsuit\!\!\!\!\!\!\text{F}\ \Phi$
Density :	$\Box\Box\Phi \rightarrow \Box\Phi$

The informal semantics of the past and future temporal operators is as follows:

- $\boxed{\text{P}}Q$ means that Q holds always in the past.
- $\boxed{\text{F}}Q$ means that Q holds always in the future.
- $\diamondsuit\!\!\!\!\!\!\text{P}\ Q$ means that Q holds sometimes in the past.
- $\diamondsuit\!\!\!\!\!\!\text{F}\ Q$ means that Q holds sometimes in the future.

We have used three different boxes, namely $\Box$, $\boxed{\text{P}}$ and $\boxed{\text{F}}$. We interpret $\Box Q$ as a formula asserting that Q holds in all worlds of a Kripke structure that are

reachable via the reachability relation R. In this case we do not say more about R. In the case of $\boxed{\text{P}}$ and $\boxed{\text{F}}$ we assume R to be the *earlier-later*, *later-earlier* relation respectively.

Some of the reachability relations that we can think of are not expressible in first-order logic and some of the relations do not have a corresponding axiom schemata expressible in propositional or first-order modal logics. Discreteness in the sense of "Between two time points (worlds or states) there are only finitely many time points (worlds or states)" is a property of the reachability relation which is not describable in first-order logic. But we can imagine some kind of weak discreteness which is first-order definable and for which, for example, a *next*-operator would make sense[2], namely by:

$$\forall x, y\ (x < y \rightarrow \exists z\,(x < z \wedge \neg\,\exists u\,(x < u < z)))$$

$$\forall x, y\ (x < y \rightarrow \exists z\,(z < y \wedge \neg\,\exists u\,(z < u < y)))$$

Again as in some examples of properties of the reachability relation we do not have a corresponding axiom schema in modal logics for these formulae. Before slipping too deep into this field we end these considerations by giving an axiom schema that expresses the discreteness used for example in logics like TLA (see Chapter 4.1) where the underlying reachability relation is transitive and reflexive:

$$\Box(\Phi \rightarrow \Diamond(\neg\Phi \wedge \Diamond\Phi)) \rightarrow (\Phi \rightarrow \Box\Diamond\Phi)$$

Since investigations concerning this correspondence are not a main concern of this work the interested reader is inter alia referred to [76, 34].

2.3.3 Time Model Used

We use two different time models in our work for the specification of time in VSE-II and Hybrid Automata. Both models have in common that points are the basic time entities. The time domain we have chosen to work with are naturals (or integers) in case of VSE-II and the rationals (or reals) in case of Hybrid Automata.

Since VSE-II is not tailored especially to specify and verify real-time systems, it does not depend on this or any other time domain. We could equally have chosen the reals for this work. It is rather a question of what we think is necessary (adequate) in order to specify and reason about real-time systems within VSE-II. In case of Hybrid Automata this question does not arise, since they are usually based on the rationals or the reals.

The time model for VSE-II used in this work is as follows:

[2]The next operator plays an important role in applications like program specification and verification and the existence of this operator is evidently closely related to the discreteness of the underlying reachability relation.

- Points as basic time entities.
- The points in time are of type integer.
- The earlier-later-relationship between the basic time entities is transitive, reflexive, discrete and has an F-Successor.

The time model for Hybrid Automata is:

- Points as basic time entities.
- The points in time are modelled by the rationals or the reals.
- The earlier-later-relation between the basic time entities is transitive, dense and has an F-Successor.

The main difference lies in the density, respectively the discreteness of the earlier-later relation. Furthermore, the semantics of the query language for Hybrid Automata is a branching time semantics whereas the query language for VSE-II has a linear time semantics.

In the sequel we shall show that a connection of the time models of VSE-II and that of Hybrid Automata can be achieved by translating/embedding Hybrid Automata into VSE-II. The question why we are not working directly with the one or the other method or tool is easy to answer: It is the aim of this work to connect both methodologies such that the advantages of the different approaches can be exploited in a way that it is adaptable to the problem at hand. If a problem is suited best to be attacked by model checking strategies and it is concerned with real-time, then Hybrid Automata work very well. On the other hand model checking strategies have their well known limitations. If one of these limits is reached one has to look for a more general approach which does not have these limitations. Here the solution is VSE-II. It is not concerned with the typical model checking limitations as it supports the use of induction techniques. Using tools like VSE-II or similar tools as for example the B-tool as an interactive verification system, then our embedding creates a smooth way from automatic to interactive strategies and in this respect integrates model checking into interactive theorem proving.

The embedding saves those properties that are already shown by automatic means. This means that not only the system itself but also the properties expressed in the Hybrid Automata property specification language (PSL) are translated to VSE-SL (the VSE-II specification language). This way the translated properties hold for the translated system. The approach described in this work to specify and reason about real-time systems translates continuous systems into discrete systems and handles these with well known conventional techniques if necessary.

Talking about system design and specification in general, we use the translation technique introduced in this work as an example for a general approach to specify industrial sized systems and their requirements. Our paradigm is the separation

of concerns. The presented methodology separates different requirement aspects and uses adequate techniques for their specification. In this work we have looked especially at real-time aspects of systems. But the approach is not limited to real-time. Information flow [71] could be another aspect of a system specification. It is clear that there are other specification techniques that are better suited to handle information flow than Hybrid Automata. Embedding these techniques into VSE-II as it is done in this work for Hybrid Automata results in a general formal development methodology where each aspect can be adequately formalised and treated. Having such a methodology at hand a formal system engineer has an instrument to handle small parts of a system as well as the complete system and all its requirements in an adequate formal and integrated way.

In what follows we show how to marry the two methodologies, Hybrid Automata and VSE-II, in a way that respective advantages are preserved and that each can benefit from the other. We will now start the presentation by giving a simple example that is used throughout this work for illustration.

3

Gasburner Scenario

As a running example to present the solution by different techniques for the same problem, we take the well-known gasburner scenario [3]. Since there exist several versions of the gasburner in the literature, we describe informally our choice here.

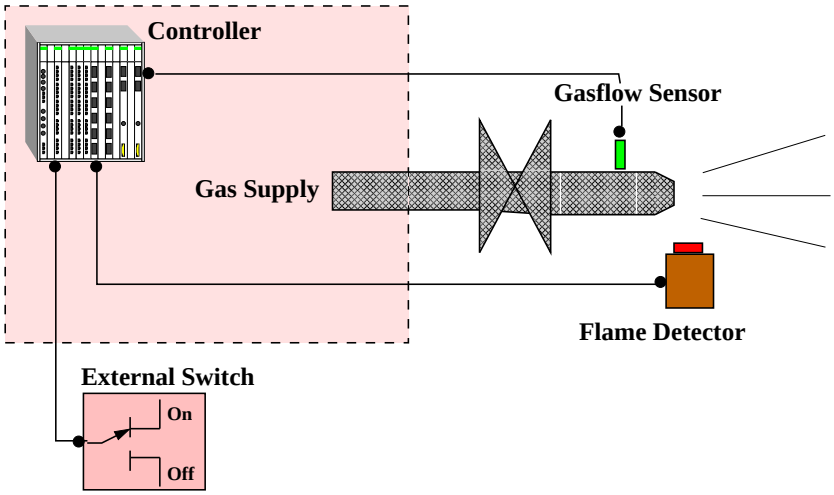

Figure 3.1: Gasburner

The gasburner scenario that we propose is illustrated in Figure 3.1. It consists of the following components:

- a controller,
- an external switch,
- a gasflow sensor and
- a flame detector.

The controller represents the central control unit of the gasburner. It is connected to a gasflow sensor and a flame sensor. Depending on the values provided by the sensors the controller computes whether to stop or release the gasflow. For example, if the gasflow sensor signals that gas flows out of the nozzle of the gasburner and the flame sensor signals that there is no flame, then, after a certain amount of time, the gasburner stops the gasflow, waits another amount of time and starts one more try to lit the flame. In real scenarios the waiting times depend for example on the room the gasburner is located in. In smaller rooms the critical gas concentration is reached much faster than in larger rooms. A safety engineer is supposed to compute the time bounds that avoid risks and this way, we abstract from the real gas concentration in the room. Equally one could specify the system by directly computing the gas concentration in the room the gasburner is located in.

In all scenarios of the gasburner presented in this work we abstract from the realistic version described here. For example, all versions abstract from the gas concentration measured in the room and so they all rely on time bounds computed by an external expert. For comparison, we assume the same time bounds for all versions.

3.1 An Abstract Version

Although there are solutions in the literature as in [64] where the gasburner specification problem is based on continuous mathematics by specifying integral equations in TLA$^+$, we abstract from continuous mathematics for the sake of simplicity. The safety requirement of the more concrete gasburner is that the gas concentration should not overtake a critical threshold. We assume that a safety engineer has calculated that the ventilation required for normal combustion would prevent dangerous accumulations of gas provided that the proportion of leakage time does not exceed one twentieth of the elapsed time in intervals of at least one minute.

So far a safety requirement is described. What remains are the design decisions which should be compatible to the safety requirements.

The first decision is that a leak is detectable/stoppable within one second. The second assumes that it is acceptable to wait another thirty seconds before risking another leak again by switching on the gasflow.

From the description given above we can isolate four informal statements which have to be formalised using the different methods proposed in the rest of this thesis:

1. The proportion of time spent in the leak state is not more than one twentieth of the elapsed time.

2. The system is observed for at least one minute, otherwise the requirements are trivially satisfied.

3. A leak is stoppable within one second.

4. After the detection of a leak there are at least thirty seconds of waiting time before the gas is again switched on.

The abstract version of the gasburner described in this section is used as the running example in this work. A more realistic gasburner scenario is used in later chapters to illustrate the formal approach and to introduce *observer models*. We shall show that according to a special refinement mapping the realistic version of the gasburner is a refinement of the abstract one given here.

Now we start the presentation of the formal methods used in this work. We begin with the description of the Temporal Logic of Actions (TLA) [66] followed by VSE-II (Verification Support Environment) [54, 89] and Hybrid Automata [15, 24, 22].

4

Formal Methods

4.1 TLA - Temporal Logic of Actions

The temporal logic used in VSE-II is based on the Temporal Logic of Actions (TLA [66]) and we shall now present it from a logical perspective. We just concentrate on its syntax and semantics. Lamport's original work [66] provides more additional, useful material we shall not explain in detail here. Other papers discuss mechanical verification in TLA [62, 39], refinement and composition [5, 7], model checking TLA specifications [97] and real-time systems [4]. We focus in this chapter on a description of the fundamentals and on the real-time aspects in TLA. This chapter is based on [66] and [6].

4.1.1 Fundamentals of TLA

TLA is a variant of temporal logic, designed for the specification and verification of reactive systems in terms of their actions. Its semantics is given in terms of possible world structures[1].

4.1.1.1 Syntax of TLA

The syntactical structure of TLA is divided into four layers which are illustrated by the pyramid shown in Figure 4.1.

[1] The syntax and the semantics of the state-based part in VSE-II is similar to the one presented here.

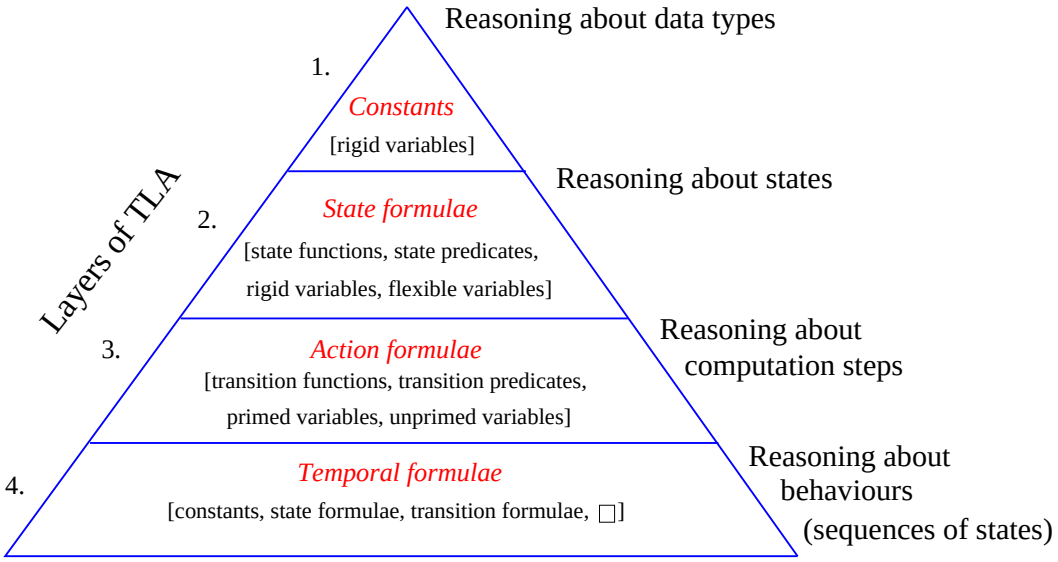

Figure 4.1: Syntactical structure of TLA

1. Layer one is concerned with formulae whose meaning is state independent. It covers the reasoning about abstract data types. These formulae are called *constants*. Within these formulae *rigid variables* may occur.

2. Layer two is concerned with formulae whose meaning is state dependent. These formulae are called *state formulae*. They are built from *state functions* and *state predicates* (*actions*) and may contain constants and rigid variables.

3. The third layer is concerned with reasoning about computation steps. These formulae are called *transition functions* and *transition predicates*. *Flexible variables* may occur primed and unprimed in these formulae. The primed variables are evaluated in a successor state.

4. The fourth layer is concerned with reasoning about behaviours, i.e. infinite sequences of states. The formulae of this layer are called *temporal formulae* (or *behaviour predicates*).

The building principle of this pyramid of TLA (Figure 4.1) is that the formulae of a lower layer are used to build formulae of a higher layer.

In the following we assume an infinite set of *variables* $\mathcal{V}$ which is partitioned into an infinite set $\mathcal{V}_\mathcal{R}$ of *rigid* variables and an infinite set $\mathcal{V}_\mathcal{F}$ of *flexible* variables. In addition, we assume a set of symbols $\mathcal{L}$, partitioned into a set $\mathcal{L}_\mathcal{P}$ of *predicate* symbols and a set $\mathcal{L}_\mathcal{F}$ of *function* symbols. To each of the symbols in $\mathcal{L}$ a natural number, its *arity*, is assigned. The sets $\mathcal{V}$ and $\mathcal{L}$ and the set of special symbols $\neg$, $\wedge$, $\Box$, $\exists$, $\exists\!\!\!\!-$, ... are pairwise disjoint.

With these definitions as a basis we define *transition formulae* where $\mathcal{V}'_{\mathcal{F}}$ represents the set of primed flexible variables $\{x' \mid x \in \mathcal{V}_{\mathcal{F}}\}$ and $\mathcal{V}_{\mathcal{E}}$ stands for the union of the sets of rigid, flexible and primed flexible variables.

Definition 4.1.1 (Transition function)
A transition function is a first-order expression over the predicate and function symbols of $\mathcal{L}$ and over the variables of $\mathcal{V}_{\mathcal{E}}$. The set of transition functions is the smallest set such that:

- *If $x \in \mathcal{V}_{\mathcal{E}}$, then x is a transition function.*
- *If $f \in \mathcal{L}_{\mathcal{F}}$ is an n-ary function symbol and $v_1, \ldots, v_n$ are transition functions, then $f(v_1, \ldots, v_n)$ is a transition function.*

Definition 4.1.2 (Transition predicate)
A transition predicate is a first-order predicate over the predicate and function symbols of $\mathcal{L}$ and over the variables of $\mathcal{V}_{\mathcal{E}}$. Transition predicates are commonly called actions and the set of actions is the smallest set such that:

- *If v_1 and v_2 are transition functions, then $v_1 = v_2$ is a transition predicate.*
- *If $p \in \mathcal{L}_P$ is a n-ary predicate symbol and $v_1, \ldots v_n$ are transition functions, then $p(v_1, \ldots, v_n)$ is a transition predicate.*
- *If A is a transition predicate, then so is $\neg A$.*
- *If A and B are transition predicates, then so is $A \wedge B$.*
- *If $x \in \mathcal{V}_{\mathcal{R}}$ and A is a transition predicate, then so is $\exists x.A$.*

Moreover, constant functions and constant predicates do not contain free flexible variables. The notion of "free variables" (FV) is defined over the set of variables $\mathcal{V}_{\mathcal{E}}$ as usual in first-order logic.

Definition 4.1.3 (State function, State predicate)
A state function is a transition function with no free primed variables. A state predicate is a transition predicate with no free primed variables.

The definition of temporal formulae is based on behaviour predicates.

Definition 4.1.4 (Behaviour predicates)
The set of behaviour predicates is the smallest set such that:

- *If P is a state predicate, then P is a behaviour predicate.*
- *If A is a transition predicate and v is a state function, then $\Box[A]_v$ is a behaviour predicate.*

- *If F is a behaviour predicate, then so is $\neg F$.*
- *If F and G are behaviour predicates, then so is $F \wedge G$.*
- *If F is a behaviour predicate, then so is $\Box F$.*
- *If $x \in \mathcal{V}_{\mathcal{R}}$ and F is a behaviour predicate, then so is $\exists x.F$.*
- *If $x \in \mathcal{V}_{\mathcal{F}}$ and F is a behaviour predicate, then so is $\exists\!\!\!\exists x.F$.*

The notion of free variable occurrences is extended to behaviour predicates (see also [6]) as follows. This definition is needed for the formulation of the proof rules given in Section 4.1.6.

DEFINITION 4.1.5 (FV_{beh})
The set of free variables of a behaviour predicate FV_{beh} is defined as follows:

- $FV_{beh}(P) = FV(P)$, *where P is a state predicate.*
- $FV_{beh}(\Box[A]_v) = \{x \in \mathcal{V} : x \in FV(A) \text{ or } x' \in FV(A)\} \cup FV(v)$, *where A is a state predicate and v is a state function.*
- $FV_{beh}(\neg F) = FV_{beh}(F)$
- $FV_{beh}(F \wedge G) = FV_{beh}(F) \cup FV_{beh}(G)$
- $FV_{beh}(\Box F) = FV_{beh}(F)$
- $FV_{beh}(\exists x.F) = FV_{beh}(F)/\{x\}$
- $FV_{beh}(\exists\!\!\!\exists x.F) = FV_{beh}(F)/\{x\}$

After the presentation of the syntactical constructs of TLA we give meaning to TLA expressions by introducing their semantics.

4.1.1.2 Semantics of TLA

The semantics of TLA formulae is formalised in terms of possible worlds.

DEFINITION 4.1.6 (STRUCTURE)
A structure $M = (\mathcal{U}, \mathcal{R}, \mathcal{F})$ where $\mathcal{U}$ is a non-empty set, $\mathcal{R}$ is a set of relations on $\mathcal{U}$ and $\mathcal{F}$ is a set of functions on $\mathcal{U}$. Each relation and each function has a natural number associated with it, its arity.

DEFINITION 4.1.7
A structure M is a structure for $\mathcal{L}$ if

- *for each n-ary relation symbol R in $\mathcal{L}$ there is a relation $R^I \in \mathcal{R}$ with $R^I \subseteq \mathcal{U}^n$, and*

- *for each n-ary function symbol F in $\mathcal{L}$ there is a function $F^I \in \mathcal{F}$ with $F^I : \mathcal{U}^n \to \mathcal{U}$.*

Definition 4.1.8 (Variable Interpretation)
An interpretation over a set of variables $\mathcal{V}$ is a mapping from $\mathcal{V}$ to $\mathcal{U}$.

Definition 4.1.9 (State)
A state s is an interpretation over $\mathcal{V}_F$, written $s(x)$ where $x \in \mathcal{V}_F$. Thus, $s(x) \in \mathcal{U}$ for every $x \in \mathcal{V}_F$.

Definition 4.1.10 (similar upto)
If $\mathcal{I}$ and $\mathcal{J}$ are interpretations over a set of variables $\mathcal{V}$, then $\mathcal{I}$ and $\mathcal{J}$ are similar up to x, written $\mathcal{I} \simeq_x \mathcal{J}$, if and only if $\mathcal{J} = \mathcal{I}[x/e]$, with $\mathcal{I}[x/e]$ equals $\mathcal{I}$ except that it maps x to e.

The definition of the semantics of TLA formulae is based on the definitions given above. We start by presenting the semantics of first-order expressions. Having this definition as a basis we were able to define the semantics of behaviour predicates.

Definition 4.1.11
The semantics of first-order expressions is the usual one:

- *If $x \in \mathcal{V}_\mathcal{E}$, then $[\![x]\!]_\mathcal{I}$ is $\mathcal{I}(x)$.*
- *$[\![f(v_1, \ldots, v_n)]\!]_\mathcal{I}$ is $f^\mathcal{I}([\![v_1]\!]_\mathcal{I}, \ldots, [\![v_n]\!]_\mathcal{I})$, where $f^\mathcal{I}$ is a function in the structure M.*
- *$[\![v_1 = v_2]\!]_\mathcal{I}$ is true if and only if $[\![v_1]\!]_\mathcal{L}$ and $[\![v_2]\!]_\mathcal{I}$ are equal.*
- *$[\![p(v_1, \ldots, v_n)]\!]_\mathcal{I}$ is true if and only if $p^\mathcal{I}([\![v_1]\!]_\mathcal{I}, \ldots, [\![v_n]\!]_\mathcal{I})$ is true, where $p^\mathcal{I}$ is a predicate in the structure M.*
- *$[\![\neg A]\!]_\mathcal{I}$ is true if and only if $[\![A]\!]_\mathcal{I}$ is not true.*
- *$[\![A \wedge B]\!]_\mathcal{I}$ is true if and only if $[\![A]\!]_\mathcal{I}$ is true and $[\![B]\!]_\mathcal{I}$ is true.*
- *$[\![\exists x.A]\!]_\mathcal{I}$ is true if and only if $[\![A]\!]_\mathcal{J}$ is true for some $\mathcal{J} \simeq_x \mathcal{I}$.*

Definition 4.1.12 (Interpretation)
Given an interpretation $\mathcal{I}$ over $\mathcal{V}_\mathcal{R}$ and a pair of states (s, t) we define the interpretation $[\![x]\!]_{\mathcal{I},(s,t)}$ as follows:

$$[\![x]\!]_{\mathcal{I},(s,t)} = \left\{ \begin{array}{l} \mathcal{I}(x) \text{ if } x \in \mathcal{V}_\mathcal{R} \\ s(x) \text{ if } x \in \mathcal{V}_\mathcal{F} \\ t(x) \text{ if } x \in \mathcal{V}'_\mathcal{F} \end{array} \right\}$$

where $s(x)$ and $t(x)$ represent the values of the variable x in state s and t respectively.

The semantics of a transition formula A under $\mathcal{I}$ and a pair of states (s,t) is $[\![A]\!]_{\mathcal{I},(s,t)}$. The semantics of a state formula at a pair of states does not depend on the second state of the pair. Therefore, we may shorten $[\![P]\!]_{\mathcal{J},(s,t)}$ if P is a state formula to $[\![P]\!]_{\mathcal{J},s}$. Analogously, the semantics of a constant formula at a pair of states does not depend on the states at all.

DEFINITION 4.1.13 (BEHAVIOUR)
A behaviour σ is an infinite sequence of states denoted by $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$.

If $\sigma = \langle s_0, s_1, \ldots \rangle$ is a behaviour, then $\sigma|_n$ is its suffix $\langle s_n, s_{n+1}, \ldots \rangle$[2]. A finite prefix of the first n elements of a behaviour σ is the sequence $s_0, s_1, \ldots, s_n$, written as $\sigma|^n$.

In order to allow for refinement and for composition of components the concept of *stuttering* has been introduced in TLA. The behaviour of a single component is simply a sequence of states that fit to the specified actions. In case of several components running in parallel we have to define what it means that the composed system executes a step. In TLA an interleaving semantics is realised by allowing only one component to do a step whereas all the variables owned by the other components are not changed. In general, we say that a step is a stuttering step for a component if it leaves all of its variables unchanged.

DEFINITION 4.1.14 (STUTTERING EQUIVALENCE)
Stuttering equivalence is the finest equivalence relation of behaviours such that any two behaviours $\rho \circ \langle t,t \rangle \circ \sigma$ and $\rho \circ \langle t \rangle \circ \sigma$ are stuttering equivalent, where $\circ$ is the concatenation of behaviours.

DEFINITION 4.1.15 (EQUAL UP TO)
Two behaviours σ and τ with $\sigma = s_0, s_1, \ldots$ and $\tau = t_0, t_1, \ldots$ are equal up to x if and only if $s_i \simeq_x t_i$ for all i.

DEFINITION 4.1.16 (SIMILAR UP TO)
Two behaviours $\sigma = s_0, s_1, \ldots$ and $\tau = t_0, t_1, \ldots$ are similar up to x, $\sigma \simeq_x \tau$, if and only if there exists σ' and τ' such that:

- *σ' and τ' are equal up to x,*
- *σ and σ' are stuttering equivalent, and*
- *τ and τ' are stuttering equivalent.*

Now we come to the definition of the semantics of behaviour predicates with respect to an interpretation $\mathcal{I}$ over $\mathcal{V}_\mathcal{R}$ and a behaviour $\sigma = s_0, s_1, \ldots$.

[2]For better readability we sometimes omit the parenthesis $\langle \ldots \rangle$ and simply write $s_0, s_1, \ldots$ for a sequence of states.

Definition 4.1.17 (Semantics of behaviour predicates)
Let P be a state predicate, A a transition predicate, F a behaviour predicate, and let $\mathcal{I}$ be an interpretation over $\mathcal{V}_R$ and $\sigma = s_0, s_1, \ldots$ be a behaviour, then:

- $[\![P]\!]_{\mathcal{I},\sigma}$ *is* $[\![P]\!]_{\mathcal{I},\sigma_0}$.
- $[\![\Box[A]_v]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![A \vee (v' = v)]\!]_{\mathcal{I},(s_n,s_{n+1})}$ *is true for all* $n \geq 0$.
- $[\![\neg F]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![F]\!]_{\mathcal{I},\sigma}$ *is false.*
- $[\![F \wedge G]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![F]\!]_{\mathcal{I},\sigma}$ *and* $[\![G]\!]_{\mathcal{I},\sigma}$ *are true.*
- $[\![\Box F]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![F]\!]_{\mathcal{I},\sigma|_n}$ *is true for all* $n \geq 0$.
- *If* $x \in V_R$, *then* $[\![\exists x.F]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![F]\!]_{\beta,\sigma}$ *is true for some* $\beta \simeq_x \mathcal{I}$.
- *If* $x \in V_F$, *then* $[\![\exists x.F]\!]_{\mathcal{I},\sigma}$ *is true if and only if* $[\![F]\!]_{\mathcal{I},\tau}$ *is true for some* $\tau \simeq_x \sigma$.

Having defined the syntax and the semantics of TLA we are now able to write TLA expressions and to interpret them. The behaviours we are able to describe until now have (in some cases) the possibility to stutter forever. To prevent such behaviours we introduce the notion of *fairness*. This is done using the *enabled* predicate of the next section.

4.1.2 Enabled Predicate

The predicate $\mathit{Enabled}(A)$, where A is an action, is true in a state s if and only if an A step can be taken in state s. Formally, the enabled predicate is defined as follows:

$$[\![\mathit{Enabled}(A)]\!]_s \;\widehat{=}\; \exists t \in St : [\![A]\!]_{(s,t)}$$

for any state s.

Syntactically $\mathit{Enabled}(A)$ is defined by:

$$\mathit{Enabled}(A) \;\widehat{=}\; \exists c_1, \ldots, c_n : A({v_1}'/c_1, \ldots, {v_n}'/c_n)$$

where $A({v_1}'/c_1, \ldots, {v_n}'/c_n)$ represents the action A after substitution of the primed variables ${v_i}'$ in A by the rigid variables c_i.

Example 4.1.1

$$\mathit{Enabled}(y = (x')^2 + n) \equiv \exists c : y = c^2 + n$$

If A represents an atomic operation of a program, then $\mathit{Enabled}(A)$ is true in those states where it is possible to perform the operation.

4.1.3 Fairness

We distinguish two kinds of fairness, namely weak fairness ($\mathcal{WF}$) and strong fairness ($\mathcal{SF}$). Informally, weak fairness expresses that an action has to be taken if it is enabled long enough.

Strong fairness says that an action has to be taken if it was often enough possible to execute it. "Often enough" can be interpreted as infinitely often so that the meaning of strong fairness is that an action is taken or that it is not infinitely often possible to execute it. Something that is not infinitely often possible is eventually impossible.

To give a formal definition of weak and strong fairness we use the temporal operator $\Diamond$, where $\Diamond F \mathrel{\widehat{=}} \neg\Box\neg F$. This formula asserts that it is not the case that F is always false. This means that F is *eventually* true.

$$\begin{array}{ll} \mathcal{WF}: & \Box(\Diamond\, \mathit{executed} \vee \Diamond\, \mathit{impossible}) \\ \mathcal{SF}: & \Box(\Diamond\, \mathit{executed} \vee \Diamond\Box\, \mathit{impossible}) \end{array}$$

Applying some temporal logic rules together with the linearity of behaviours we get:

$$\begin{array}{ll} \mathcal{WF}: & \Box\Diamond\, \mathit{executed} \vee \Box\Diamond\, \mathit{impossible} \\ \mathcal{SF}: & \Box\Diamond\, \mathit{executed} \vee \Diamond\Box\, \mathit{impossible} \end{array}$$

In order to give a complete formal definition of $\mathcal{WF}$ and $\mathcal{SF}$ we have to clarify how to interpret *executed* and *impossible*. To this end we take advantage of the enabled predicate defined above.

$$\begin{array}{ll} \mathit{executed} \text{ is defined by:} & \langle A\rangle_f \\ \mathit{impossible} \text{ is defined by:} & \neg \mathit{Enabled}(\langle A\rangle_f) \end{array}$$

where $\langle A\rangle_f$ is defined to $A \wedge f' \neq f$ where f is a state function. Thus, the formal definition of weak and strong fairness becomes:

$$\begin{array}{lcl} \mathcal{WF}_f(A) & \mathrel{\widehat{=}} & \Box\Diamond\langle A\rangle_f \vee \Box\Diamond\neg \mathit{Enabled}(\langle A\rangle_f) \\ \mathcal{SF}_f(A) & \mathrel{\widehat{=}} & \Box\Diamond\langle A\rangle_f \vee \Diamond\Box\neg \mathit{Enabled}(\langle A\rangle_f). \end{array}$$

4.1.4 Further Definitions

A TLA specification allows stuttering steps, i.e. steps that leave all the variables appearing in the formula unchanged. A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behaviour should not affect the truth of the formula.

DEFINITION 4.1.18 (FORMULA INVARIANT UNDER STUTTERING)
We say that a formula F is invariant under stuttering if and only if adding or deleting a stuttering step to a behaviour σ does not affect whether σ satisfies F.

In TLA there is no possibility to write formulae that are not invariant under stuttering.

Definition 4.1.19 (Property)
A property is a set of behaviours that is invariant under stuttering. The set of all behaviours satisfying a TLA formula can be identified with the formula itself.

Definition 4.1.20 (Safety Property)
A formula F is a safety property if and only if the following holds: F contains a behaviour if and only if it is satisfied by every finite prefix of the behaviour.

Intuitively, a safety property asserts that something "bad" does not happen. An example for a safety property is the formula $\Box[A]_{\langle f \rangle}$.

Definition 4.1.21 (Canonical Form)
A formula that consists of a conjunction of an initial predicate, a term of the form $\Box[A]_{\langle f \rangle}$ *and fairness properties is in* canonical form.

Definition 4.1.22 (Invariant under Stuttering)
A logic is invariant under stuttering*, if none of its formulae can distinguish between two stuttering-equivalent behaviours.*

Invariance under stuttering is important in connection with refinement. A fundamental result about TLA is that it is invariant under stuttering, namely:

Theorem 4.1.2 *If F is a behaviour predicate, $\mathcal{I}$ is an interpretation over $\mathcal{V}_{\mathcal{R}}$, and σ and τ are two stuttering equivalent behaviours, then $[\![F]\!]_{\mathcal{I},\sigma} = [\![F]\!]_{\mathcal{I},\tau}$*

An immediate consequence of theorem 4.1.2[3] is that a formula in canonical form is always invariant under stuttering[4]. Furthermore, assuming that the initial state is consistent, it is not possible to write down inconsistent TLA formulae (formulae in canonical form), because every behaviour has the possibility to stutter, i.e. nothing happens at all[5].

As we have seen in Section 2.1, TLA is a logic without past operators. As shown in [6] without a concept of past and the corresponding rules, a TLA proof system is incomplete. Therefore, a concept of history-determined [66] variables was introduced in TLA. Informally, a history-determined variable is a variable whose value can be inferred from the actual and the past values of the other variables.

A formal definition of history-determined variables is the following.

[3]The proof of this theorem is given in [6].

[4]A TLA formula is invariant under stuttering even if the subscript does not contain all variables.

[5]Often such behaviours, where at a certain point nothing happens, indicate a faulty or undesired behaviour and consequently a faulty specification.

DEFINITION 4.1.23
A variable h is a history-determined variable for a formula Φ if and only if Φ implies the formula $Hist(h, f, g, v)$, where f and v are state functions, g is a transition function and h does not occur freely in f or in v and h' does not occur freely in g and $Hist(h, f, g, v)$ is defined by

$$Hist(h, f, g, v) \;\widehat{=}\; (h = f) \wedge \Box[(h' = g) \wedge (v' \neq v)]_{\langle (h,v) \rangle}$$

We use history-determined variables in the example of the non-lossy queue in Appendix B.1.1.

4.1.5 Real-Time in TLA

In this section we show how TLA deals with real-time, the main part of which is taken from [4].

As indicated in Section 2.2.3 TLA is not specially tailored to handle real-time. TLA rather constitutes a more general logical framework in which real-time can be specified (using the logical constructs TLA provides).

Real-Time is expressed in TLA specifications with the help of the variable *now*. Although *now* has a special interpretation, it is a flexible variable in TLA. *now* has the following properties:

- The value of *now* is always a real number[6], i.e. $now \in \mathbb{R}$.
- The value of *now* in a behaviour never decreases.

These properties are expressed by the formula RT:

$$RT \;\widehat{=}\; (now \in \mathbb{R}) \wedge \Box[now' \in (now, \infty)]_{\langle now \rangle}$$

where $\mathbb{R}$ is the set of real numbers and (r, ∞) is an open interval representing the set $\{t \in \mathbb{R} : t > r\}$. The formula RT describes behaviours where either *now* is incremented or it remains unchanged (stuttering index).

It is convenient to separate time advancing steps from program steps. This separation results in a structured and more readable specification[7].

The separation is done by strengthening the constraints on *now* in RT as follows:

$$RT_v \;\widehat{=}\; (now \in \mathbb{R}) \wedge \Box[now' \in (now, \infty) \wedge (v' = v)]_{\langle now \rangle}$$

where v represents the tuple of the program variables.

[6] One can also use the rational or even natural numbers, if it is adequate for the system to be specified.

[7] To some extent the observer models described later in section 6 support exactly this idea of describing different parts of the specification separately and it extends this idea by generating different views on a specification.

Since RT_v is equivalent to $RT \wedge \Box[now = now']_{\langle v \rangle}$, we get a conjunction of a canonical form with RT_v:

$$Init \wedge \Box[N]_{\langle v \rangle} \wedge RT_v \;=\; Init \wedge \Box[N \wedge (now' = now)]_{\langle v \rangle} \wedge RT$$

Real-Time properties can be expressed[8] by placing timing bounds on actions. These timing bounds are represented by a so-called *timer*. These timers restrict the way the variable *now* can change.

Definition 4.1.24 (Timer)
A timer for a formula Φ is a state function t, such that

$$\Phi \rightarrow \Box(t \in \mathbb{R} \cup \{^{+}_{-}\infty\})$$

Definition 4.1.25 (Upper–Bound Timer)
A timer t is used as an upper–bound timer by conjoining the formula:

$$MaxTime(t) \;\widehat{=}\; (now \leq t) \wedge \Box[now' \leq t']_{\langle now \rangle}$$

$MaxTime(t)$ asserts that *now* is never greater than t.

Definition 4.1.26 (Lower–Bound Timer)
A timer t is used as a lower–bound timer for an action A by conjoining the formula:

$$MinTime(t, A, v) \;\widehat{=}\; \Box[\neg A \vee (t \leq now)]_{\langle v \rangle}$$

$MinTime(t, A, v)$ asserts that an $\langle A \rangle_v$ step can never occur if t is greater than *now*. If an $\langle A \rangle_v$ step is taken then *now* is at least t or *now* is already greater than t.

We now present two different TLA timers that have their origin in different interpretations of the timing conditions inferred from the following example.

Example 4.1.3 *A common type of timing constraints on an action A is the following. Action A must be taken within δ seconds after it is enabled. After an A step the next A step must occur within δ seconds when action A is re-enabled.*

We get at least two different interpretations for the timing constraints on A.

1. An A step must occur if A is continuously enabled for δ seconds. This constraint can be expressed using $MaxTime(t)$ where t is a timer that satisfies

[8] There are other possibilities to express real-time properties in TLA. The interested reader is referred to [64].

the following formula:

$$\begin{aligned} VTimer(t, A, \delta, v) \;\widehat{=}\; & t = \textbf{if } Enabled(\langle A\rangle_v) \textbf{ then } now + \delta \\ & \qquad\qquad\qquad\qquad \textbf{else } \infty \\ & \wedge \Box[t' = \textbf{if } (Enabled(\langle A\rangle_v))' \\ & \qquad \textbf{then if } \langle A\rangle_v \vee \neg Enabled(\langle A\rangle_v) \\ & \qquad\qquad \textbf{then } now + \delta \\ & \qquad\qquad \textbf{else } t \\ & \qquad \textbf{else } \infty \\ & \wedge\; v' \neq v]_{\langle t,v\rangle} \end{aligned}$$

Such a timer is called a *volatile* timer[9].

2. An A step must occur if A has been enabled for a total time of δ seconds. We express this constraint again by using $MaxTime(t)$ where t satisfies the formula

$$\begin{aligned} PTimer(t, A, \delta, v) \;\widehat{=}\; & t = now + \delta \\ & \wedge \Box[t' = \textbf{if } Enabled(\langle A\rangle_v) \\ & \qquad \textbf{then if } \langle A\rangle_v \textbf{ then } now + \delta \\ & \qquad\qquad\qquad\qquad \textbf{else } t \\ & \qquad \textbf{else } t + (now' - now) \\ & \wedge\; (v, now) \neq (v, now)']_{\langle t,v,now\rangle} \end{aligned}$$

Such a timer is called a *persistent* timer.

Since we have described these timers only by giving their informal interpretations and their definitions, we present in Appendix B a more demanding example where timing constraints are used to make a lossy queue non-lossy.

4.1.6 Proofs in TLA

Until now we have given the syntax and the semantics of TLA formulae together with special formulae to handle real-time. This is sufficient for writing specifications and for some informal reasoning about these specifications. However, for formal proofs we need rules of inference. We do not give all the proof rules for TLA and only sketch some of them and show how they are built and how they are used. The reader interested in detailed information concerning proofs in TLA is refered to [6, 4].

The basic rules for temporal logic and rules for existential quantification are mainly taken from [6].

[9] Other versions of the $VTimer$ are mentioned in [64, 63].

4.1.6.1 Invariants

Most proofs require the invention of invariants as a starting point. Whether an invariant holds can be proved with the following rule of inference:

$$\frac{P \wedge (N \vee v' = v) \rightarrow P'}{P \wedge \Box[N]_v \rightarrow \Box P}$$

In this rule P is a state predicate, N is an action and v is a state function. This rule is sound in the sense that if we assume that $P \wedge (N \vee v' = v) \rightarrow P'$ is valid, i.e. it is true for all its interpretations and behaviours, then so is $P \wedge \Box[N]_v \rightarrow \Box P$. Using this rule we can prove an invariant of a system that performs the actions specified in N.

When an invariant P is proved it can be used. This is done according to the following rule:

$$\frac{P \wedge P' \wedge (N \vee v' = v) \rightarrow (M \vee u' = u)}{\Box P \wedge \Box[N]_v \rightarrow \Box[M]_u}$$

As can be seen in this rule the invariant P is added to the antecedent of the formula to be proved. Applying the rule we semantically jump to an arbitrary state of a behaviour expressed syntactically by removing the $\Box$ symbols. In this state we can use P and P' to prove the theorem, since we know that P is an invariant for the system. A generalisation of this rule is given in [6].

4.1.6.2 Quantification

The rules for existential quantification are familiar from first-order logic:

$$\frac{G \rightarrow F}{\exists x.G \rightarrow F}, x \in \mathcal{V}_{\mathcal{R}}, x \notin FV_{beh}(F) \qquad \frac{G \rightarrow F[b/x]}{G \rightarrow (\exists x.F)}, x \in \mathcal{V}_{\mathcal{R}}, b \text{ constant function}$$

The rules for the temporal existential quantification ($\mathbf{\exists}$) are:

$$\frac{G \rightarrow F}{\mathbf{\exists} x.G \rightarrow F}, x \in \mathcal{V}_{\mathcal{F}}, x \notin FV_{beh}(F) \qquad \frac{G \rightarrow F[b/x]}{G \rightarrow (\mathbf{\exists} x.F)}, x \in \mathcal{V}_{\mathcal{F}}, b \text{ state function}$$

The existential quantification over flexible variables corresponds to hiding [66]. These rules play an important role in the refinement proofs for reactive systems.

4.1.6.3 Verification

Most TLA proofs [66, 4, 6] are done manually with the help of rules like the ones above.

Another direction within the TLA approach is taken by the TLA$^+$ model checker TLC[97]. TLA$^+$[65] is a specification language for concurrent and reactive systems that combines the temporal logic TLA with full first-order logic and Zermelo

Fränkel Set Theory. Furthermore, it supports large modular system specifications. A language that satisfies all the needs of industrial applications will be too expressive for model checking techniques and thus it can hardly be applied as the only means to verify systems. TLC can handle a subclass of TLA$^+$ specifications and has mainly been used to debug specifications. The largest of these specifications had more than 2000 lines of TLA$^+$ specification. The interested reader is best refered to [67] to see what TLC can cope with.

Since the VSE-II temporal logic specifications (TLSPEC's) that we are going to explain in the next chapter are rather similar to TLA, we skip the presentation of the gasburner example within TLA and present it there.

4.2 VSE-II - Verification Support Environment

4.2.1 Introduction

The VSE-II system is a tool for the formal development of software systems. It consists of

- a basic system for editing and type checking specifications and implementations written in the specification language VSE-SL,
- a facility to display the development structure,
- a theorem prover for treating the proof obligations arising from development steps as for example refinements,
- a central database to store all aspects of the development including proofs, and
- an automatic management of dependencies between development steps.

Compared to VSE-I [52, 53], which was based on a simple, non-compositional approach for state-based systems, VSE-II [54, 55] is extended with comprehensive methods in order to deal with *distributed* and *concurrent systems* [87]. Furthermore, it was enhanced by a more efficient and uniform proof support which makes use of implicit structuring of the proof obligations. The basic formalism used in VSE-II to specify and to verify state-based systems is close to TLA (Temporal Logic of Actions) [66]. A refined *correctness management* allows for an evolutionary software development [56, 90].

VSE-II is based on a methodology which uses the structure of a given specification (e.g. parameterisation, actualisation, enrichment, or modules) to distribute the deductive reasoning into local theories [89]. Each theory is considered as an encapsulated unit, with its own local signature and axioms. Relations between different theories, as they are given by the model-theoretic structure of the specification, are represented by

consequences or lemmata obtained by using local axioms and other formulae included from linked theories.

This method of a structured specification *and* verification is reflected in the central data structure of a *development graph* (see for example Figure 4.9), the nodes of which correspond to the units mentioned above. It also provides a graphical interface for the system under development. Different types of specifications are displayed as different types of nodes, e.g. abstract data types as hexagons, while the relations between the nodes are displayed as links in the development graph.

4.2.2 Abstract Data Types

Formal specification techniques treat data objects in computer science as mathematical objects of a certain domain. To get rid of the technical details of data types in real programming languages one either considers a single domain, which happens to be rich enough, as it is done in Z [94] or one abstracts away from incidental properties of a particular domain by considering whole *classes* of structures as it is done in the abstract data type approach [69, 26]. Here data objects are viewed as resulting from the nested application of certain functions and a (concrete) data type is given by a collection of domains (carriers) and functions on these domains. To introduce abstraction, one separates syntax from semantics and considers classes of *algebras* (or models) $\mathcal{A}$ which are interpretations of a fixed collection Σ of *function symbols* f as functions $f_{\mathcal{A}}$ on the carriers of $\mathcal{A}$. Classes of algebras are restricted by axioms of a logical language (over Σ) which is usually a sublanguage of first-order predicate logic. The various approaches to abstract data types differ in the classes of algebras that are considered and the corresponding description techniques that are used for specification.

In VSE-II [55] full first-order logic is used to specify data types. In general, all models $\mathcal{A}$ satisfying the axioms Ax, written $\mathcal{A} \models Ax$, are considered. Two models $\mathcal{A}_1$ and $\mathcal{A}_2$ of Ax will not necessarily be isomorphic, that is, they may differ not only in the concrete representation of data objects. This allows for a really abstract style of specification where one describes *what* a function does but not *how* this is realized. A well-known example is encoding and decoding. On the abstract level it might suffice to know that $dec(enc(v)) = v$ leaving open a wide range of perhaps highly sophisticated implementations for later refinements.

VSE-II also supports a more constructive style of specification, that allows the user to introduce recursive data structures like lists and trees. Classes of algebras are restricted by requiring that certain carriers are *generated* by constructors from some $\Sigma' \subset \Sigma$ which means that for each element a of the corresponding carrier A, there is a term τ over Σ' that denotes a. In particular, we consider *freely* generated structures where each element $a \in A$ has a *unique* representation in Σ'. Figure 4.2 shows the specification of lists and the enrichment of the list data type by a function and a predicate. Generated clauses bring about *induction principles* that have to be used if inductive theorems or lemmata have to be proven. The axiomatic counterpart of these clauses is generated by the system when the deduction unit belonging to the specification is generated.

So far, we have discussed elementary (unstructured) specifications. In VSE-II there are two ways of structuring data type specifications: *generic* specifications and the *import* of specifications. By importing (using) several specifications *enrichment* and (disjoint) *union* can be modelled in VSE-II. Generic specifications provide an additional slot (parameter part) to describe the formal parameter including axioms. Upon actualisation of a generic theory as part of the "using slot" of some other theory, proof obligations are generated for these axioms. In Figure

```
THEORY elems
 PURPOSE "type element with default constant"
 TYPES element
 FUNCTIONS default : element
THEORYEND

BASIC list_data
 PARAMS elems
 USING NATURAL
 list = nil WITH nilp |
        cons(first : element, rest : list) WITH consp
 VARS x, y, z : list
 SIZE FUNCTION length : list -> NAT
BASICEND

THEORY list
 PURPOSE "concatenation of lists and element-predicate"
 PARAMS elems
 USING list_data[element, default]
 FUNCTIONS  append   : list, list -> list
 PREDICATES _ ELEM _ : element, list
 VARS x, y, z : list;
      a       : element
 AXIOMS
 FOR append :
   append(nil, x) = x;
   append(cons(a, x), y) = cons(a, append(x, y))
 FOR ELEM :
   a ELEM x <-> (EX y, z : x = append(y, cons(a, z)))
THEORYEND
```

Figure 4.2: Freely Generated Data Types

4.2 there is an example for such a parameterised specification. The parameter theory `Elems` contains the definition for a type `element`. So `list_data` can be instantiated to form lists of elements of any kind. Furthermore, `list_data` is used in the theory `list` where additional functions (`append`) and predicates (`ELEM`) are defined.

Structured theories are not flattened in VSE-II. Each specification is an entity in its own right and linked to other specifications according to the used specification building operation. The semantic counterparts of these operations determine the translation into logical formulae (renaming) and also the flow of information between the units corresponding to the specification entities.

VSE-II implements an elaborated theory of data type refinements [86]. Operations of an (abstract) export algebra are implemented by programs that use operations from some (more concrete) import algebra. The axioms of the export specification give rise to proof obligations that are assertions about the implementing programs. Properties of the import specifications are used in the course of verifying the assertions in a programming logic.

4.2.3 Concurrent System Specifications

Large specifications are structured in several subcomponents which constitute the complete system specification including the environment specification. In the description of these units elementary specifications and different structuring operators for state-based systems are used.

4.2.3.1 Elementary Specifications

For the specification of state transition systems a specification language close to the specification language of TLA [7, 66, 9] (see also Section 4.1) is used. In addition to the theory of compositional development presented in [9], which covers the composition of systems using input and output variables, *shared variables* are supported by the structuring operators in VSE-II. Furthermore, the form of a specification of a component, also discussed in [9], is $\exists\, x_1, \ldots, x_n.(\mathit{Init} \wedge \Box[\text{SYS-STEPS}]_{\bar{v}} \wedge \text{FAIR})$, where

- SYS-STEPS are the *actions* (steps) made by the system,
- $\bar{v}$ is the stuttering index, which contains flexible variables of the system,
- *Init* is a predicate which holds initially,
- $x_1, \ldots, x_n$ are the internal variables, and
- FAIR stands for the fairness requirements of the system.

In addition to the normal form, the user can specify an elementary specification using a pseudo programming language which will be translated in subsequent steps into the normal form given above. Moreover, it is possible to specify a system's behaviour by an arbitrary temporal logic formula. This specification style is usually non-constructive and the proof support for such specifications is restricted to the usual temporal logic proof rules.

4.2.3.2 Structuring of Specifications

One of the main means to manage complexity in real world applications is to structure the development process. As a first step we have to provide operators to structure specifications. Based on one or more such operators we then look for a modular way of proving properties and of treating refinements. In the VSE-II approach refinement proofs are supported by a general modular proof method.

In specifying industrial sized systems, composition operators play a central role. Large specifications can only be handled in a safe and time saving way if the specification is structured. The search for suitable composition operators is connected with the problem of modular proving and modular refinement of specifications. Both the proving of properties of the composed system and the refinement proofs should be done in a (local) independent way, if possible. In this context a refinement proof appears to be a special case of the general methodology of modular proofs.

In VSE-II there are two operators to structure state-based specifications: **combine** and **include**. We focus on the **combine**-operator which models the concurrent execution of two components given by the specifications S_1 and S_2 but we mention also the **include** operator and its semantics.

Components communicate by input-output variables or by shared variables. In both cases large parts of the inner structure of the components, in particular the flow of control, is hidden to the outside world. To achieve a proper synchronisation with the environment, components may have to wait for a "response".

Concurrency is modelled by considering all possible *interleavings* of actions of the combined systems. Basically, a behaviour $\bar{s}$ is a behaviour of the combined systems if and only if it is a behaviour of both S_1 *and* S_2. However, in order to model the concurrent execution of S_1 and S_2 by conjunction, that is by $S_1 \wedge S_2$, we have to allow environment steps in the (local) specifications S_1 and S_2. In [9] environment steps are modelled by stuttering. This technique only works for communication by input-output variables. It does not work in the presence of shared variables. A more general approach [89, 54] is to record for each step the active component. For this, a special predicate $active_{spec}$ is used. By using $active_{spec}$ it becomes possible to distinguish steps of a component from steps of the environment which might consist of several other components. Let $A_1, \ldots, A_n$ be the elementary actions of a component given by S. The action formula POSSIBLE–STEPS has to be of the

form

$$((A_1 \vee \ldots \vee A_n) \wedge active_S) \vee (\neg active_S \wedge \bar{x} = \bar{x}' \wedge \bar{o} = \bar{o}') \ ,$$

where $\bar{x}$ and $\bar{o}$ are the internal and output variables of S. The second disjunct claims that environment steps do not affect the variables "owned" by S. The system builds this formula automatically from the list of actions for VSE-SL specifications in canonical form.

Given two elementary systems S_1 and S_2 as above the combined system is given by

$$S_1 \wedge S_2 \wedge \Box(\neg active_{S_1} \vee \neg active_{S_2}) \ .$$

For specifications in canonical form consistency is preserved by the **combine** operator. This does not mean that the combined system really exhibits the intended behaviour.

The second operator for the composition of systems, say $\mathcal{S}$ and $\mathcal{S}'$, is **include**. $\mathcal{S}$ **include** $\mathcal{S}'$ represents a mechanism to provide certain functionalities which are often used in the specification of systems without re-specifying them. It allows for a hierarchical composition of state machines. The semantics of $\mathcal{S}$ **include** $\mathcal{S}'$ is the conjunction of $\mathcal{S}$ and $\mathcal{S}'$. The steps of $\mathcal{S}'$ in the composed system are not left open but the actions of the system $\mathcal{S}'$ occur now in the system $\mathcal{S}$. In this way the externally visible behaviour of $\mathcal{S}$ is also determined by the system $\mathcal{S}'$. This composition style can end up in an inconsistent system. So proof obligations arise which have to be proved in order to assure the consistency of the composition.

4.2.4 Assumption Guarantee Specifications

One of the advantages in working with VSE-II is that systems can be specified and verified in a modular way. This means that a system needs not to be specified as a single monolthic block, but as an arbitrary number of components with well-defined interfaces for communication. Also the verification process supports the modular specification style by allowing to prove properties local to components using assumptions about the environment of the component. This style of specification and verification has several advantages:

- The specification and the verification is not as sensitive to changes as it would be in a monolthic specification style.
- Proofs become easier, since we do not always have to consider the whole specification, i. e. the whole system behaviour, as a precondition.
- The understanding of the specified system gets deeper, a point corroborated by the experience we made in specifying systems. For modular systems one has to define the assumptions of each component explicitly. Thus, the dependencies between the different system parts become obvious and explicit.

- Systems are often embedded in an environment within which they are expected to work. Refining a system does not mean to refine the system *and* the environment. Refinement of the system or part of the system separately is only possible in a modular specification.

The assumption guarantee theory in the presence of shared variables is explained in detail in [89, 88], where we used a distributed *Producer-Channel-Consumer* example to explain the theory and discuss the proof of the property that "no information is lost in the communication between the components".

4.2.5 Structured Deduction

Structuring specifications as described above supports readability and makes it easier to edit specifications by allowing the user to use local notions. However, the system exploits the structure beyond this purely syntactical level. Components of a combined system can be viewed as systems in their own right, where certain parts can be observed from the outside while most of the inner structure, including the flow of control and local program variables are hidden.

In particular we can prove properties of a combined system in a modular way. This means that we attach local *lemma bases* to components where local proofs are conducted and stored. Exchange of information between lemma bases is on demand. This approach has two main advantages: First, the given structure of the specification is used to reduce the search space in the sense that large parts of the overall system are not visible and, second, the *revision process* is supported by storing the proofs local to certain lemma bases, thus making the export and import of information (between lemma bases) explicit.

We now show some cases, the first two are simple real-time examples. After these we present the gasburner and a more complex real-time example specifying a control software for a storm surge barrier. These examples demonstrate how real-time can be specified and how real-time properties can be verified in VSE-II.

4.2.6 A Simple Real-Time Example in VSE-II

Real-time constraints are usually used to place an upper bound on how long it may take for a system to do something. In this sense real-time constraints can be considered as a strong form of liveness, specifying not only that something eventually happens but also when it must happen. In the specifications presented here, real-time constraints usually replace liveness properties.

The example is first presented in a monolthic or so-called closed specification. There is no environment and the system specification contains everything we need to know about its behaviour. This version of the example is called the "non-concurrent" version. After showing how a simple property of this specification can be proved, we transform the non-concurrent version into a concurrent one. This

version is examined in the same way as the non-concurrent one. Hereafter, we try to summarise the effects of this transformation and argue that the concurrent version has some advantages over the non-concurrent one.

4.2.6.1 The Non-Concurrent Version

The development graph of our example is shown as a VSE-II screen shot in Figure 4.3. In our model we use a theory named `Func` where some constants and a function

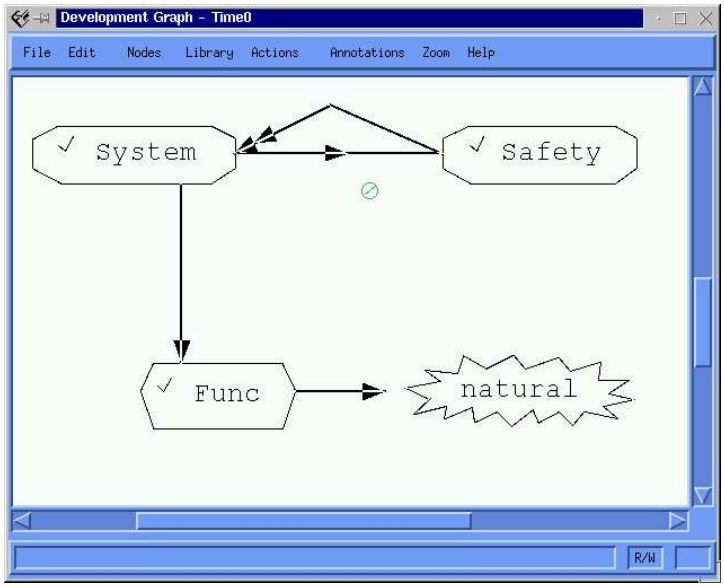

Figure 4.3: Development Graph

`rand` are specified. This theory can be seen in Figure 4.4. The meaning of the constants becomes clear when looking at the temporal logic part of the system. The constant `duration` is defined to be greater than zero and the function `rand` is defined as the sum of two natural numbers[10].

The temporal logic part is represented by the temporal logic specification (TL-SPEC) `System` shown in Figure 4.5. Within this component we have specified three interface variables of type natural: `x`, `timer` and `now`. The behaviour of the system is defined with the help of the action definitions of `tick` and `send`. The system starts in an initial state where `x` and `now` are equal to 0 and `timer` is equal to `duration`. The system can take a `tick` step, a `send` step or it can stutter which means that all the variables mentioned in the stuttering index remain unchanged. The effect of the `tick` action is that `now` gets incremented by one and `x` and `timer` remain unchanged. The `send` action sets `x` to some "random" value which depends

[10]Natural numbers are predefined data-types in VSE-II. This is graphically indicated by a zig-zag shape of the corresponding theory.

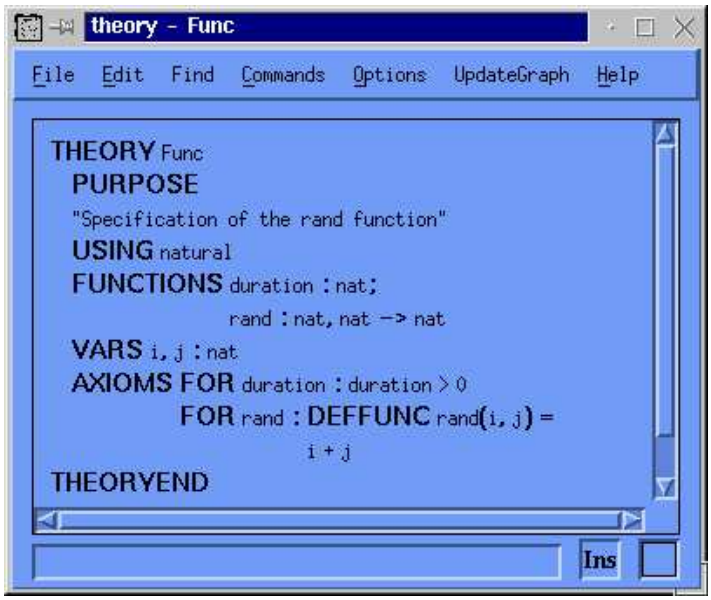

Figure 4.4: `Func` Theory

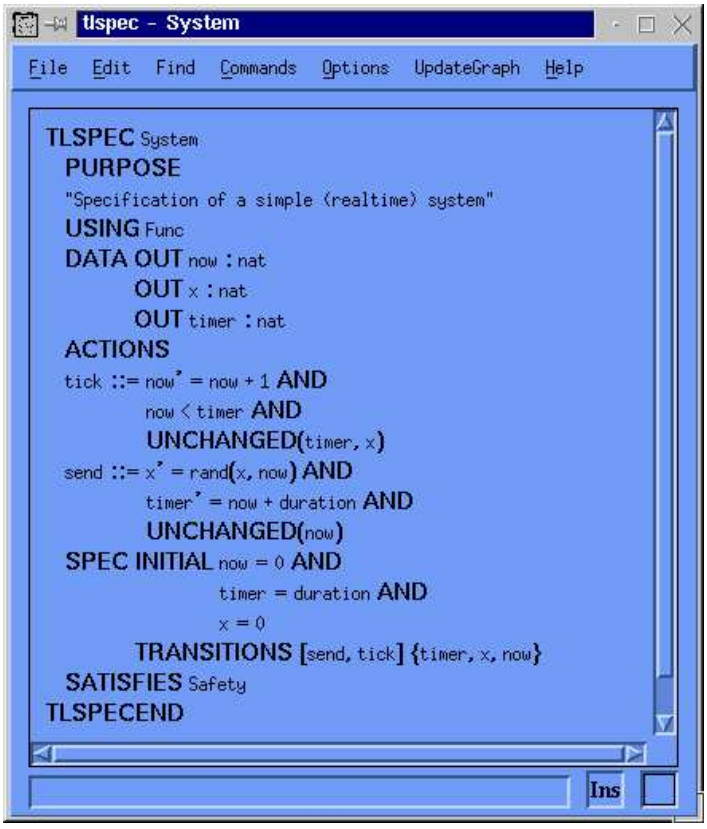

Figure 4.5: TLSPEC `System`

on the time represented by the variable `now` and the value of `x` itself[11]. The `timer` is set to the sum of `now` and `duration` and `now` remains unchanged.

This specification realizes an upper bound `timer` for the change of the variable `x`. This means that we can guarantee (under some assumptions concerning the function `rand`) that there are at least two different values for `x` in every interval of length greater than `duration`. In other words, this means that if we compare the value of the variable `x` at some time with the value it had at least `duration` time units before, then it will have changed. This property can be expressed with the following temporal logic formula:

$$\Box((now = t \wedge x = s) \rightarrow \Box(x = s \rightarrow now \leq t + duration))$$

We want to check whether this property holds for our specified system. Therefore, we insert this formula into the safety model represented by the temporal logic specification `Safety` (see Figure 4.6).

Figure 4.6: TLSPEC `Safety`

In order to start the proof of the mentioned property we have to translate the specifications given in Figure 4.3 to logic. This is done automatically in VSE-II by starting the verification process. In this process the logical representation of the specification of the system is generated in the deduction unit. Hereafter we can start proving the corresponding safety property (see Figure 4.6) that assumes the name `Export-axiom-1` (see Figure proof-structure) in the deduction unit.

[11] For simplicity we have taken `rand` to be a function that computes the sum of its arguments.

Proof of the Property in VSE-II Proofs are visualised in VSE-II by trees where nodes contain sequents and represent the result of rule applications. Part of the main proof of our property is shown in Figure 4.7, where the nodes of the proof tree can be "clicked on" to show the corresponding sequent . In order to close the

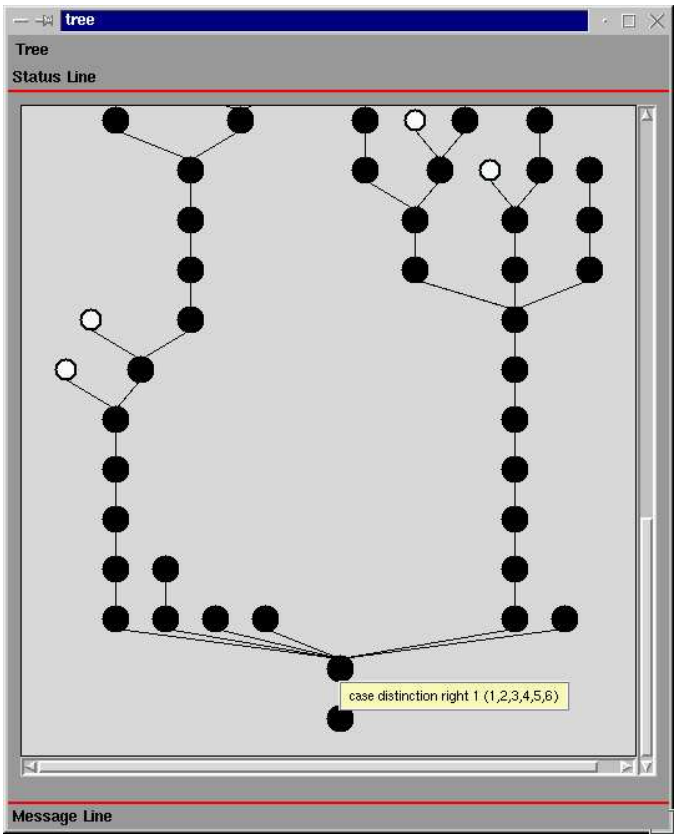

Figure 4.7: Proof of the Safety Property

proof of the property shown in Figure 4.6 we make use of two lemmata:

1. `help-inv1`: $\Box((x = s \wedge now = t) \rightarrow \Box(x = s \rightarrow timer \leq t + duration))$
2. `help-inv5`: $\Box(x = s \rightarrow \Box(x \geq s))$

To prove these two lemmata we need another three lemmata:

1. `help-inv4`: $\Box(now \geq 0)$
2. `help-inv6`: $\Box((x = s \wedge timer \leq u) \rightarrow \Box(x = s \rightarrow timer \leq u))$
3. `help-inv7`: $\Box(timer \leq now + duration)$

We show the lemmata in their originally specified form together with their dependencies. We do not tune the lemma-base in this small example[12].

[12]It is clear that this lemma base could be optimised as some lemmata could be joined together to one lemma or some of them might even be superfluous.

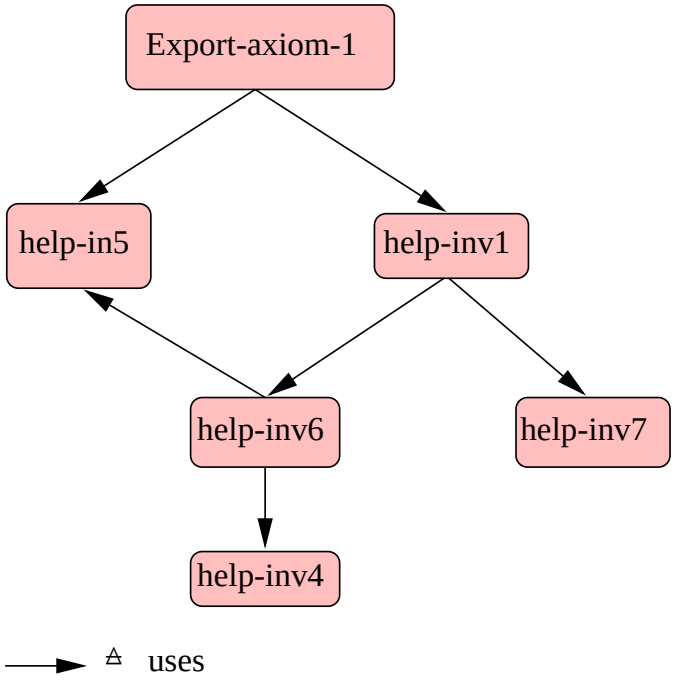

Figure 4.8: Proof Structure

As can be seen in Figure 4.8 there is no circle in the "using" structure. Since we have specified a closed system we do not need assumptions in the proofs that are made completely in VSE-II and we are done.

In the following we have separated our example into two components, a system and a clock component and we show the consequences which result from the new concurrent design of the system with respect to specification and verification.

4.2.6.2 The Concurrent Version

The development graph of the concurrent version can be seen in Figure 4.9. There we have a combine node (`CompSys`) that represents the composition of the temporal logic specifications of the `System` and the `Clock`.

The `System` component consists of a single action `send`. This action is basically the same as in the monolthic case (see Figure Figure 4.5). The same holds for the `tick` action in the `Clock` component.

Whereas in the non-concurrent version communication was implicitly possible because all the variables are visible to all the actions, communication has to be specified explicitly in the concurrent version. It is realized via the variables `timer` and `now` as indicated in Figure 4.9 by dotted lines between the `System` and `Clock` components. In Figure 4.10 the syntactical representation of this connection in VSE-II is given. Semantically the arrows indicate that

- the `System` component is only allowed to read the values of the variable `now`,

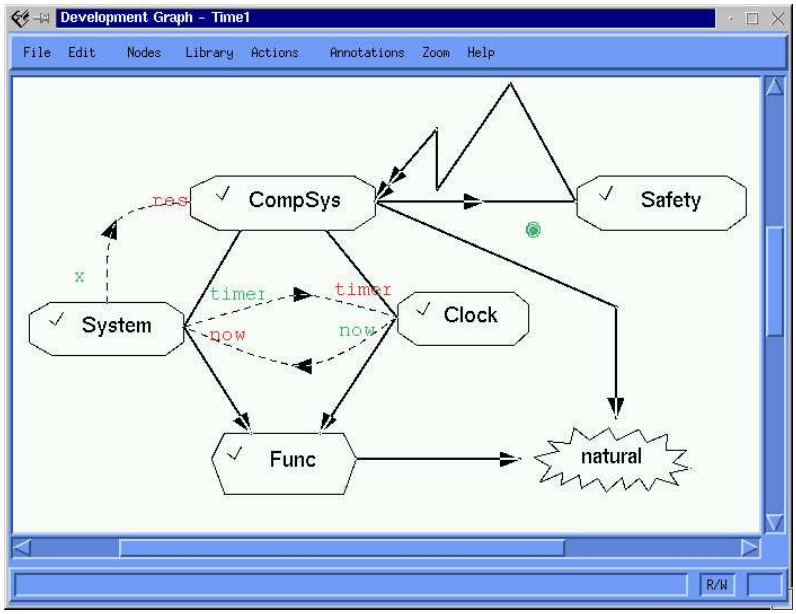

Figure 4.9: Development Graph: Concurrent Version

but not to change it, and

- the `Clock` component may only read the `timer` variable.

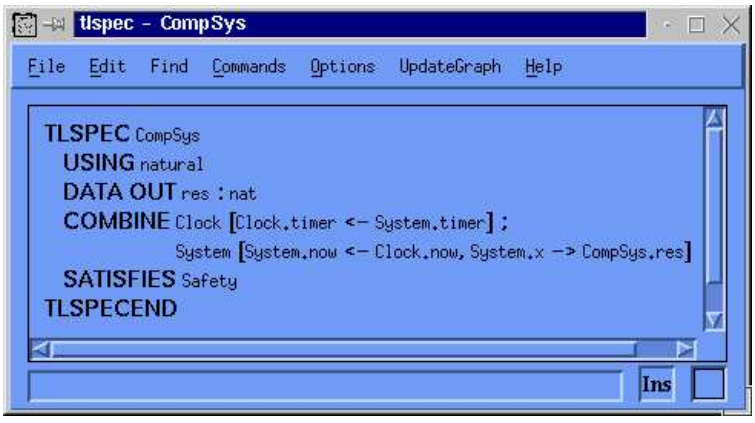

Figure 4.10: `CompSys` Specification

The composed specification is represented by the `CompSys` component in Figure 4.9. Semantically, this is the conjunction of the `System` and `Clock` components. The safety requirement shown in Figure 4.6 represents the property that this specification should satisfy. The underlying datatype specifications as well as the specification of the safety property are identical to the non-concurrent version.

Proofs in the Concurrent Version As in the monolthic version the proof obligation corresponding to the satisfies-link is generated automatically. Its proof can be done local to some component: `System` or `Clock`.

As indicated earlier in this work finding proofs local to components can have advantages over a monolthic specification and proving style. In our small example it is not a necessity to structure the specification, but it exhibits the general principle we follow throughout this work. Considering refinement it is often the case that only the system specification and not the clock specification has to be refined and this might simplify the refinement steps of such specifications.

Recall that searching for proofs local to the `System` component we know nothing about the environment (that consists of the `Clock` component in this case). All the knowledge about the environment has to be inserted via assumptions that are stored in the lemma base of the `System` component. The assumptions made about the environment are used in this proof. The proof tree in Figure 4.11 represents the proof of the proof obligation corresponding to the satisfies-link between the components `CompSys` and `Safety`. This proof uses the `CompSys-Export-axiom1` lemma[13] as a main lemma inserted in the lemma base of the `System` component. Changing the `Safety` component only affects this proof and not the proofs local to the `System` component in which the `CompSys-Export-axiom1` lemma was proved.

The proof of the main local `System` lemma `CompSys-Export-axiom1`, which is identical to the proof obligation `export-axiom1` (generated by the VSE-II system), uses the same lemmata as the corresponding proof in the monolthic version. But in this proof additional information is needed about the behaviour of the environment which was explicitly given in the monolthic version. We need two assumptions about the `Clock` in the `System` component:

1. `Clock-ass-1`: $\Box(\neg is\_active(system) \rightarrow (now' = now \vee now' = now + 1))$
2. `Clock-ass-2`: $\Box(is\_active(system) \rightarrow now' = now)$

By introducing assumptions we are able to insert the *proof relevant* part of the environment of the `System` component. We have to take care of the fact that the assumptions have to be discharged in later proof steps. This means that a proof has to be given that the environment really satisfies the assumptions the system has made. For example, in writing `Clock-ass-1` as $\Box(\neg is\_active(system) \rightarrow now' = now + 1)$ we are not able to discharge this assumption since it does not represent one of the possible behaviours of the environment of the `System`, namely the `Clock`.

The overall proof structure is illustrated in Figure 4.12. As can be seen there it is very similar to the structure in the monolthic case if we omit the assumptions and their proofs. The structure is not identical since some of the proofs are done in different deduction units. The proof of `Export-axiom-1` is done (virtually) in the satisfies-link. This proof uses the lemma `CompSys-Export-axiom1` which is

[13]This lemma is identical to the formula shown in Figure 4.6.

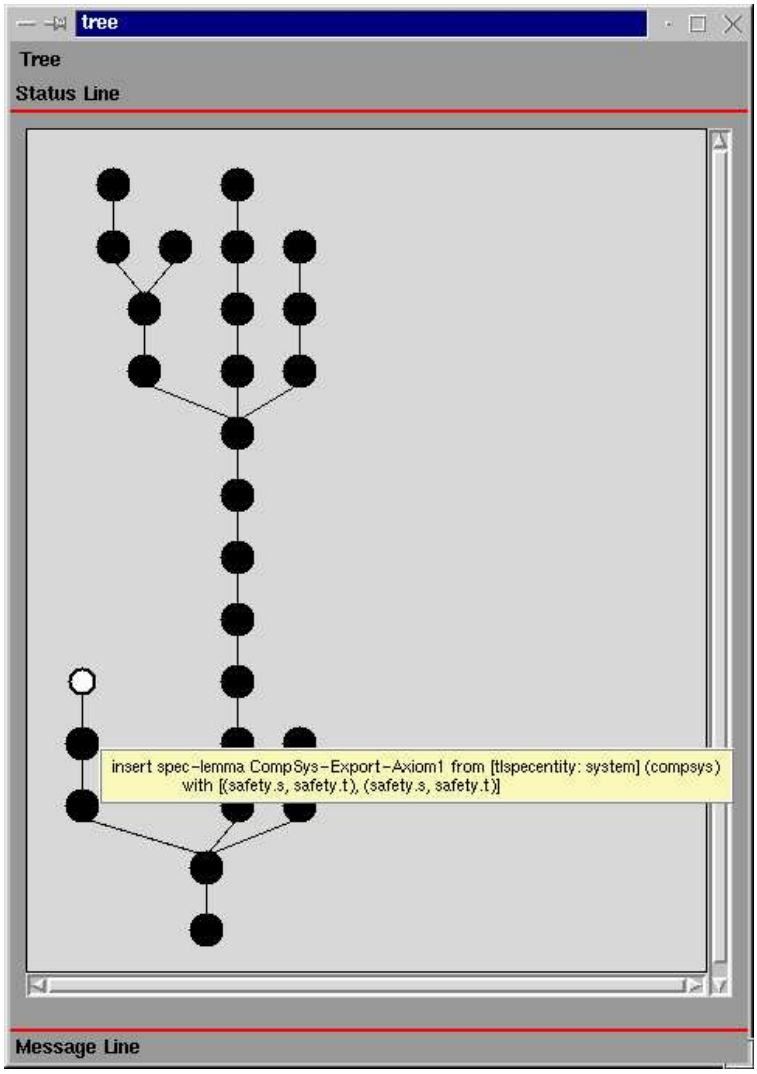

Figure 4.11: Proof of `export-axiom1`

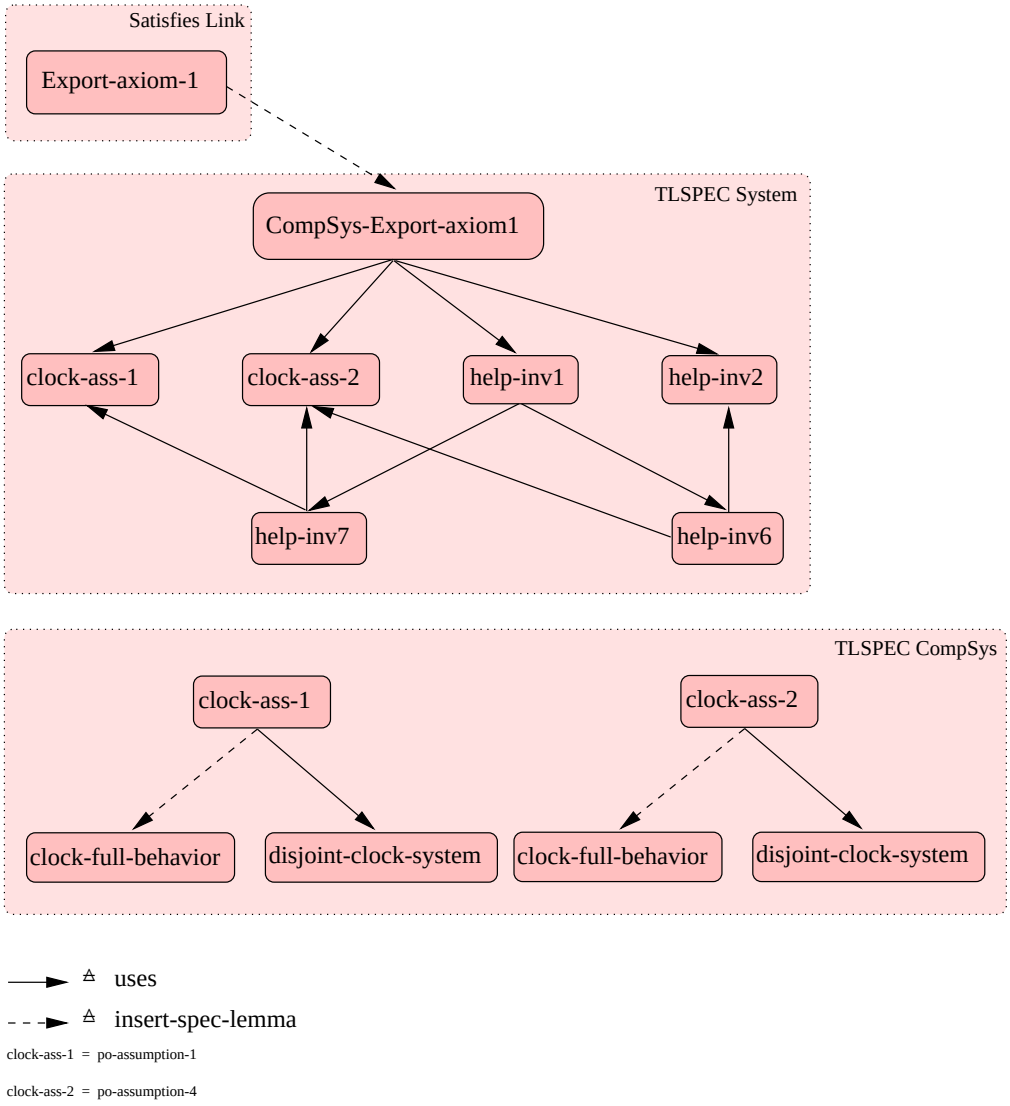

Figure 4.12: Proof Structure

a local lemma in the TLSPEC `System`. In order to prove this lemma we have used `help-inv1`, `help-inv2`, `help-inv6` and `help-inv7` which are known from the monolthic example. The assumptions `clock-ass-1` and `clock-ass-2` were needed for the proof as indicated in Figure 4.12. These assumptions are inserted since, by executing steps (actions) of the specified system within the process of proving, we have to know what happens to the variables that are not changed by the `System` component. Furthermore, we have to know how the `IN` variables (input variables) of the `System` component are changed, if the `System` is not active. All this has to be done since performing proofs local to a component means that our knowledge is mainly restricted to the facts known about the component itself. Thus, we know nothing about the behaviour of the environment. In the case of the monolthic system specification the use of assumptions was not needed since all proof-relevant information is available in the closed system specification. In the concurrent version this was simulated by doing the proofs local to the `CompSys` component.

Two interesting points concerning local and global proofs are:

- Since verification is not a one-way process it is often the case that one has to adjust the specification in order to get the desired behaviour or to fix specified bugs. Searching for proofs relative to the complete system means that these changes of the specification always result in redoing all the proofs unless they are not completely independent of the specification. Although the system supports this work by reuse and replay techniques, it often ends up in a far too big effort. In the other case the situation is usually[14] much better. Suppose for example, that we change the specification of the `Clock` component. Since we have not done any proof local to the `Clock` component only the proofs of the assumptions become invalid. So we only have to redo these proofs and all other work can be saved[15].

- In specifying a system concurrently and in structuring proofs by using lemmata it seems to be natural also to structure the proofs with respect to the specification structure. In many cases even where assumptions were needed it results in less proof work to do. Furthermore, it detects dependencies of the specified system parts which are not obvious. In searching for a proof local to a system component without using assumptions we definitely know that this theorem was independent of the behaviour of the environment, whereas if we do not succeed in closing the proof we realize at some point that we need at least some parts of the behaviour of the environment within this proof. With this "deep understanding" of the specified system we are often able

[14]It is clear that in the worst case where the changes of the specification are so profound that the lemmata are no longer satisfied, we have to adapt the lemmata and search for new proofs.

[15]If the changes of the `Clock` component have the effect that we cannot discharge the assumptions, we are not able to close the proof and we have to redo the proof of the `CompSys-Export-axiom1` (see Figure 4.12).

to insert only that part of the behaviour of the environment that is really needed. This again results in a much clearer and explicit understanding of the dependencies of the system components.

We present the gasburner example as it is specified and verified in VSE-II, followed by the presentation of a more complex real-time example introducing a general methodology to specify real-time in VSE-II.

4.2.7 The Gasburner in VSE-II

The development graph of the gasburner specified in VSE-II is shown in Figure 4.13. It consists of

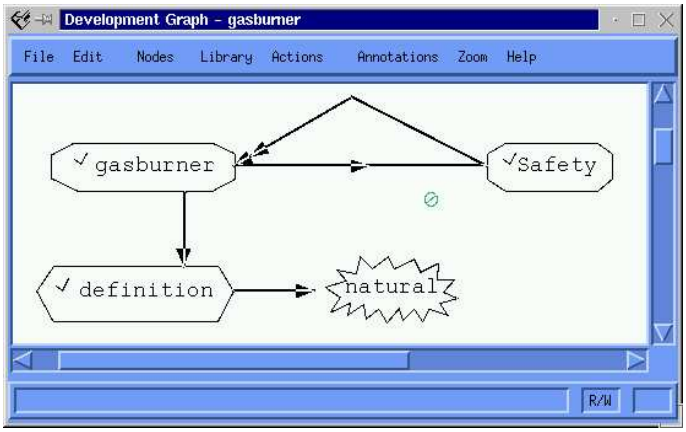

Figure 4.13: Development Draph of the Gasburner Specification

- the abstract data type definitions `natural` and `definition`, and
- the temporal logic specifications `gasburner` and `safety`.

The abstract datatype `natural` is a predefined datatype in VSE-II. Natural numbers are generated by `zero` and `succ`, i.e. every natural number is obtained by a finite number of applications of the successor function `succ` on `zero`[16].

The abstract datatype `definition` consists of the definition of a constant `c` of type natural which is greater than 0. It contains the type declaration `states` defined as an enumerated type. This type is used to model the safe (`non_leak`) and unsafe (`leak`) states of the gasburner. These states play an important role in the specification of the behaviour of the gasburner.

[16]This specification is not monomorphic since it does not determine what the predecessor of `zero` is. It can be any natural number.

The Temporal Logic Specification The TLSPEC `gasburner` describes the behaviour of the gasburner by declaring its interface and action definitions.

```
TLSPEC gasburner
 PURPOSE  "Specification of the gasburner."
 USING definition
 DATA OUT x, y, t : nat
      OUT state : state_t
 ACTIONS
 phi_1 ::= state = leaking AND
           state' = non_leaking AND
           x' = 0 AND
           UNCHANGED(y, t)
 phi_2 ::= state = non_leaking AND
           state' = leaking AND
           x >= 30 * c AND
           x' <= c AND x' = 0 AND
           UNCHANGED(y, t)
 phi_1_star ::= state = leaking AND
                state' = non_leaking AND
                NOT x + 1 <= c AND
                x' = 0 AND
                UNCHANGED(y, t)
 psi_1 ::= state = leaking AND
           state' = leaking AND
           x + 1 <= c AND
           y' = y + 1 AND
           x' = x + 1 AND
           t' = t + 1
 psi_2 ::= state = non_leaking AND
           state' = non_leaking AND
           x' = x + 1 AND
           y' = y + 1 AND
           UNCHANGED(t)
 SPEC INITIAL x = 0 AND y = 0 AND t = 0 AND state = leaking
      TRANSITIONS [phi_1, phi_2, phi_1_star, psi_1, psi_2]
                  {x, y, t, state}
      FAIRNESS WF(phi_1_star) {x, y, t, state},
               WF(psi_1) {x, y, t, state},
               WF(psi_2) {x, y, t, state}
 SATISFIES Safety
TLSPECEND
```

The specification of the gasburner makes use (`USING` slot) of the abstract datatype `definition`. In the following `DATA` slot the interfaces are defined using the flexible variables `x`, `y`, `t` and `state` with the following meaning:

- The variable `y` accumulates overall time.
- The variable `x` represents a control clock that guarantees that the system remains for at most c time units within the `leak` state and for at least $30 * c$ time units within *non_leak* state.
- The variable `t` counts leakage time, i.e. the amount of time the system resides in the `leak` state[17].
- The variable `state` can have the values `leaking` or `non_leaking`. It represents in some sense the state of the gasburner. If the variable has the value `leaking`, then unlit gas flows out of the nozzle of the gasburner. In the `non_leaking` state this is not the case. This means that the gas flows out and is burned or there is no gasflow at all. The specification of the gasburner given here is able to distinguish between a leaking and a non-leaking situation. But it cannot distinguish whether the gas is burned or there is gasflow at all in a non-leaking situation.

The mentioned variables occur in the specification within the five specified actions of the system: `phi_1`, `phi_2`, `phi_1_star`, `psi_1` and `psi_2`. Whereas the actions `phi_1`, `phi_2` and `phi_1_star` describe a change of the variable `state`, i.e. `state` is changed from `leaking` to `non_leaking` or vice versa, it remains the same in the actions `psi_1` and `psi_2`.

The specification uses symbolic values to express time bounds. For example, in action `phi_2` the term `x >= 30 * c` is used as a guard. The constant c makes the system independent of a chosen time unit. Let us assume, for example, the value of c is 10, then x has to be incremented 300 times, before the barrier is reached. If, however, c is 1000, x has to be incremented 30000 times to reach the barrier. That is, if we consider seconds as the basic time unit, then steps take $\frac{1}{10}$ of a second or a millisecond, respectively.

However, the real proof of the (safety) property is independent of specific values of the constant c. Therefore, it is valid for all possible integer values and thus for all granularities (even infinitesimal).

As you can see in the `SPEC` slot `phi_1_star`, `psi_1` and `psi_2` are assumed to be executed in a fair manner. This way we prohibit the gasburner from doing anything but stuttering.

[17]The variables `t` as well as `y` are needed to have the possibility to express properties like $safe$ and *round* (see Section 4.2.7.1). One of the first versions of the gasburner has used only the variable `x` in its description. In order to prove something interesting about this system, the property specification language was extended to introduce new clocks (so-called integrators). In later versions of the gasburner these clocks were added to the system description.

4.2.7.1 Proof in the Gasburner Scenario

In this section we present the proof of a theorem in the VSE-II system which treats a property of the gasburner specified already in Section 4.2.7 (see also Section 4.3). Besides others the following properties are proved with the VSE-II system:

1. $\Box(y \geq 60c \Rightarrow 20t < y)$ (*safe*)
2. $\Box(t \geq 3c \Rightarrow y \geq 60c)$ (*round*)

Whereas *safe* expresses something about the right functioning of the system, it is an objective of the system specification, the formula *round* expresses something like a requirement on the system's behaviour[18]. *round* guarantees that if the gasburner was at least $3c$ time units in the leaking state, then the total time measured by y is at least $60c$ time units.

In order to prove that *safe* and *round* are properties of the gasburner, we have to find a suitable invariant. Such invariants are used together with the system specification to prove properties like *safe* or *round*. Using such an invariant is correct if it is proved to be a consequence of the system specification.

Finding a suitable invariant is, in general, not a mechanical process, but an inventive process where the user must generate the appropriate idea.

In order to prove the *safe* condition we need to prove a proposition about the relation between the time t the system spends in the state *leak* and the total time y of observation. Considering the specification we can see that the `leak` state is changed to `non-leak` after at most c time units, whereas `non-leak` can only be left earliest after $30c$ time units. This amounts to a minimum ratio of 1 to 30 after each full cycle of the state-machine, i.e. at the moment when the state enters `leak`. Then the condition $30t \leq y - t$ holds. We cannot prove this condition directly, since it can only be proved as a consequence of the states visited before. Therefore, we will formulate an invariant that also covers all possible states in-between. While the automaton remains in one of the states `leak` or `non-leak` the ratio is changed in one of both direction depending on the state. During $state = leak$ the leak time t grows exactly as fast as the time x since entering the current state. This yields $30(t-x) \leq y-t$. When leaving and entering state `non_leak` the value x has reached at most c. Therefore, at this moment we have $30t \leq y-t+30c$. Again we have to subtract x for the other states after entering and before leaving `non_leak` resulting in $30t \leq y-t+30c-x$. Now we are ready to present the whole invariant I:

$$\begin{aligned}\Box(&state = leaking \Rightarrow 30(t-x) \leq y-t \wedge \\ &state = non\_leaking \Rightarrow 30t \leq y-t+30c-x)\end{aligned}$$

We are now going to show how this invariant can be used to prove the formulae *safe* and *round*. Both cases are very similar, so we concentrate on the proof of

[18] From a logical viewpoint, *safe* and *round* are invariants of the gasburner.

the property $safe$. We represent the proof by identifying sequents by numbers in brackets, (1) for example, and we use triples T consisting of

1. a number identifying a sequent,

2. a list of rules applied to that sequent where the application is in the order the rule is mentioned, and

3. a set of result sequents which are represented by a set of numbers identifying sequents.

The names of the sequents we are using and the corresponding sequences are given in the following table:

(1) $\widehat{=}$ $\Box(state = leaking \Rightarrow 30(t-x) \leq y - t \wedge$
$state = non\_leaking \Rightarrow 30t \leq y - t + 30c - x)$
$\vdash$
$\Box(y \geq 60c \Rightarrow 20t < y)$

(2) $\widehat{=}$ $state = leaking \Rightarrow 30(t-x) \leq y - t,$
$state = non\_leaking \Rightarrow 30t \leq y - t + 30c - x, y \geq 60c$
$\vdash$
$20t < y$

(3) $\widehat{=}$ $state = leaking, 30(t-x) \leq y - t, y \geq 60c \vdash 20t < y$

(4) $\widehat{=}$ $state = non\_leaking, 30t \leq y - t + 30c - x, y \geq 60c \vdash 20t < y$

(5) $\widehat{=}$ $state = leaking, x \leq c, 20t \leq y - 11t + 30x, y \geq 60c \vdash 20t < y$

(6) $\widehat{=}$ $t < 3c, y \geq 60c \vdash 20t < y$

(7) $\widehat{=}$ $t \geq 3c, x \leq c, y \geq 60c, 20t \leq y - 11t + 30x \vdash 20t < y$

The proof rules applied are described in detail in Appendix C. There we have listed a summary of the proof rules taken from the manuals of the VSE-II system description. The proof is represented by the following sequence of triples of type T[19]:

[19] We omit some steps for readability.

$[(1), (\text{always left}, \text{always right}), \{(2)\}]$

$[(2), (\text{case distinction on } state = leaking), \{(3), (4)\}]$

$[(3), (\text{Insert-lemma} : \Box(state = leaking \Rightarrow x \leq c),$

$\text{always left}, \text{case distinction on } state = leaking), \{(5)\}]$

$[(5), (\text{cut with } t < 3c), \{(6), (7)\}]$

$[(6), (\text{arithmetic}, \text{inequations}), \{\}]$

$[(7), (\text{arithmetic}, \text{inequations}), \{\}]$

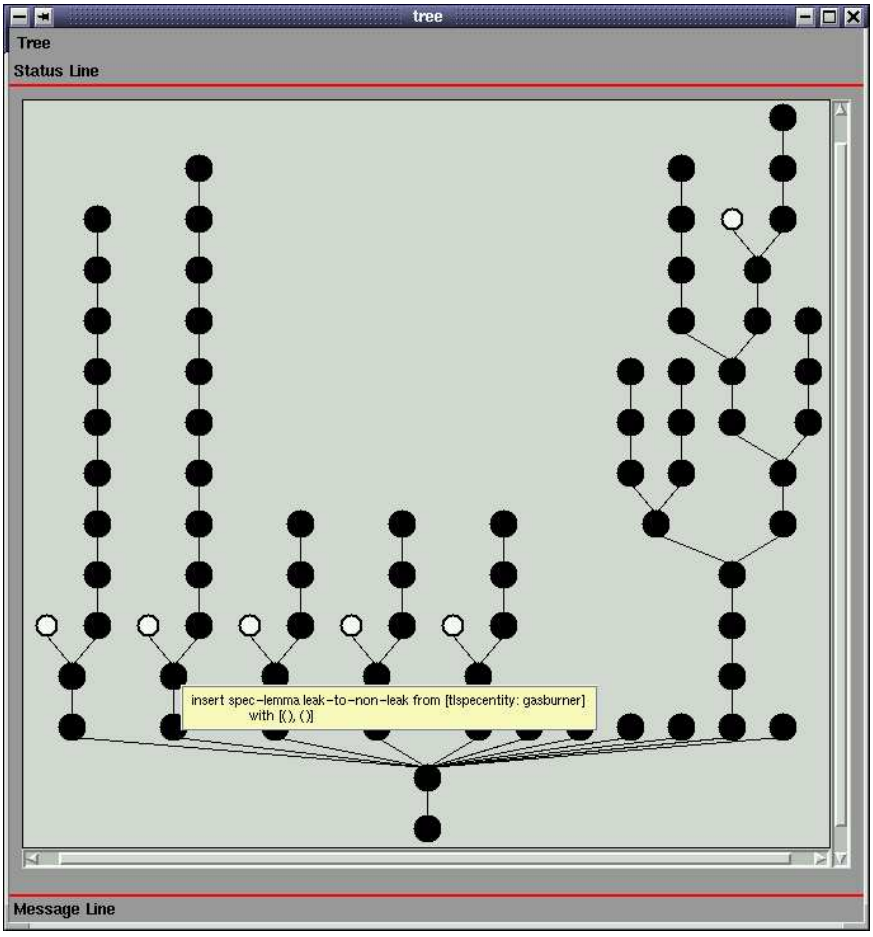

Figure 4.14: Proof Tree of the Gasburner Scenario

We do not present the details for the proof of subgoal (4) because it can be done analogously to the proof of subgoal (3). All the proofs including the lemmata mentioned have been found with the help of the VSE-II system.

The critical point in finding such a proof is to find the right invariant. This point is often neglected in the relevant literature, but it is obvious that finding the

right invariant is a very important, perhaps the most important, step in such a proof.

The proof of the safety property using all the introduced lemmata and invariants is shown as a VSE-II proof tree in Figure 4.14.

In examining finite systems we are normally not confronted with such problems. They have the advantage that we do not need to invent such an invariant because the whole behaviour of the system can be finitely computed. It might take a very long time but in the case of the gasburner it is a solvable task for many properties. How this can be done and how the gasburner is handled with the help of hybrid automata is shown in Section 4.3.

4.2.8 A more Complex Real-Time Example in VSE-II

In this section we present a real-time modelling technique in VSE-II which is based on a global clock architecture with a discrete time scale. We argue that this model can be applied to a wide range of scenarios. We illustrate the modelling technique with an example that describes an emergency closing system (ECS) that physically consists of several huge gates to isolate the North Sea from the Eastern Scheldt, a control system and several sensors to measure the water levels inside and outside the gates. The emergency control system which is part of the overall system keeps track of the changes of the water levels and closes the gates if the water level reaches a dangerous limit. The specification and the proofs are generated with the VSE-II tool [54, 87, 89].

A picture of the gates (see Figure 4.15) gives an impression of the ECS, actually the part of the ECS, the gates, that are controlled by the ECS control unit. Specification and verification of the ECS was part of an industrial project at the DFKI Safety and Security Department.

4.2.8.1 General Principles

In this section we discuss a general technique used to model real-time systems like the ECS using the VSE-II system. Since ECS is a very special system which in some aspects might not be considered typical for the kind of systems the methodology is designed for, the discussion in this section is more general than necessary for the formal model of the ECS.

Requirement Engineering A formal model like the one of the ECS is only one constituent of the requirements engineering process. Starting point in the case of the ECS is a document containing the (informal) requirements specification.

The requirements are given by verbal descriptions and an informal, mainly graphical design of the system. The most important requirements deal with the behaviour of the system in time. While the verbal description is not very precise

Figure 4.15: The gates of the ECS

and therefore not sufficient in itself, the graphical description contains many details of a particular solution, actually it is already close to a technical realization.

Our aim was to provide a formal requirement specification that does not refer to the internal structure of the system and then to prove that these requirements are satisfied by a system specification that is as close as possible to the design given in the document. In contrast to the technical description in the document, the formal system specification does not use concrete physical entities, like seconds, meters, and milliamperes. In a separate step one would have to choose concrete values that are in accordance with the constraints given by the formal specification.

Overall Structure of the Model The formal model consists of three components: the *system*, the *environment*, and a component containing a global *clock* (see Figure 4.16). The system takes as inputs values from sensors measuring various water levels and values from switches that are set by an operator. Basically, it computes two output signals, one for closing the barrier and one for opening it. The design is *fail safe* in the sense that the first signal going down means *close*.

Both, the environment and the clock are separated from the system to allow for a refinement of the abstract specification to the actual system. The environment records the changes of the water levels as well as the different stati of the switches,

i.e. in this case there are no complex assumptions about the possible behaviour of the environment. However, the steps of the three components have to be synchronised to model the assumptions about their behaviour with respect to time. In particular, not all states can be given a meaningful interpretation, Therefore, the component containing the clock *updates* certain *visible* variables upon each tick. All the remaining variables are *internal* and cannot be accessed by an observer. Note that the synchronisation among the components as well as the choice of observable variables implicitly formalises our way of looking at the system and is therefore relevant for the notion of correctness.

The requirements specification only refers to the visible variables and has access neither to the variables local to the system nor to the synchronisation mechanism. However, it uses a variable *time* that provides with the current time in each state.

All three components are needed to formalise the safety properties of the ECS, and in particular, the environment is modelled explicitly in a separate component that computes new values for the input variables. Although an approach based

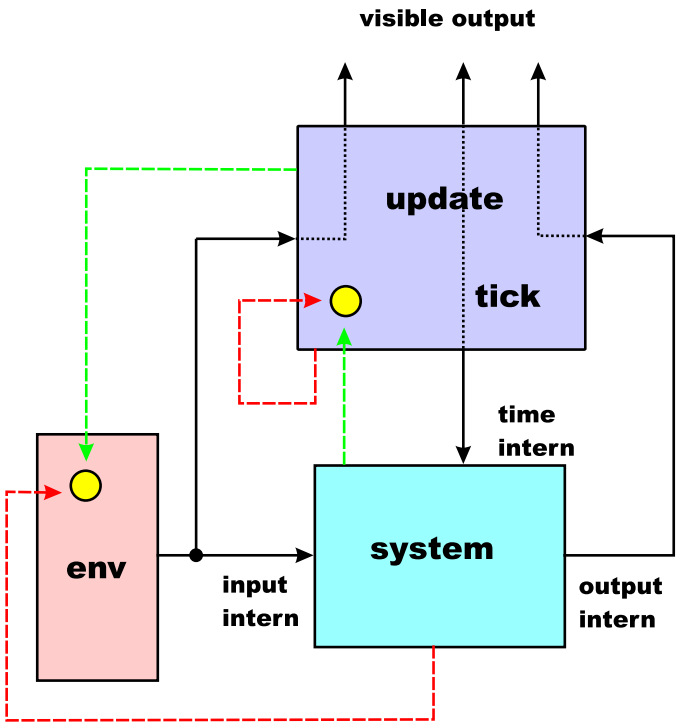

Figure 4.16: The Complete Scenario

on assumptions about the environment might be more abstract, the closed system approach proposed here makes it easier to impose "rules" on the interaction between the three components. This is done by realizing the flow of control with shared variables acting as guards (indicated by circles within the components in Figure 4.16). An action (step) of a component can only be executed if the guard's value is *true*. For ECS we have imposed the following rules: First, the system reacts

immediately (i.e. without any delay) to a change of the input variables caused by the environment or the clock. Second, whenever the system has updated its output values there has to be a tick of the clock. There is a problem with the first rule. Immediately after the environment has changed some input value, the output values of the system (in general) do not have the desired values. Moreover, since there was no tick of the clock either, we might have different input values at the same time. To solve this problem we distinguish between internal variables and variables visible to the outside. The environment, the system, and the clock (in the update component) change only internal variables. Whenever the clock ticks the corresponding visible variables are updated with the current values of the internal ones. With this in mind we can reformulate our rules: First, the system reacts immediately (without any delay) to a *visible* change of the input variables caused by the environment or the clock, and second, whenever the system has updated its output values there has to be a tick of the clock *and the new values of the internal variables become visible to the outside.* There are many ways of implementing these rules. We have chosen a liberal implementation illustrated in Figure 4.16. Less liberal but perhaps more systematic ones (excluding certain behaviours as "not meaningful") are obviously possible as well.

Environment actions Environment actions model changes in the state of the environment. In the simplified scenario there is only one action which models the change of the waterlevel (*chw*). In the general case there might be other actions. All of these might occur in an arbitrary order but at most once before the system updates its outputs and the next tick occurs. This behaviour is realized as follows. If the environment is enabled and there has been a step, i.e. the waterlevel has changed, it becomes disabled. The next step can only be done by the system[20] component after which the clock ticks and enables the environment again.

A very general description of the behaviour of the environment is as follows:

$$
\begin{aligned}
& enable - chw = true \quad \wedge \\
& (waterlevel' \geq waterlevel \vee waterlevel' < waterlevel) \quad \wedge \\
& enable - chw' = false \quad \wedge \\
& enable - clock' = false
\end{aligned}
$$

If there is more than one sensor for the waterlevel that provides an input to the system each of them is treated separately. Any number of sensors may change (at most) once between two updating steps of the system.

System The system itself is modelled as a single action which *updates* its outputs according to certain conditions. One of the outputs is to set the timer appropriately.

[20] In talking about the "system component" we mean, depending on the context, either the control part of the overall system or the whole system including all the subcomponents.

In general, the system can be modelled by several actions constituting its possible behaviour. But we have to take care of the fact that there should be no environment and no clock steps as long as the system computes its outputs. There are several solutions to this. In the ECS scenario we have chosen the simplest one, where the system component consists of only one action. Since actions are atomic there can be no intersection with the other components in this step.

Updates of the system are always possible. An update enables the clock. In the simple scenario the system is modelled by the following step:

$$
\begin{aligned}
& enable - clock' = true \quad \wedge \\
& waterlevel \leq triggerlevel \rightarrow timer' = timer \quad \wedge \\
& \neg waterlevel \leq triggerlevel \rightarrow timer' = time + duration \quad \wedge \\
& time < timer' \rightarrow out' = 0 \quad \wedge \\
& \neg time < timer' \rightarrow out' = 1
\end{aligned}
$$

In the first conjunct the system enables the clock, since the system has computed the new signals. The computation of the outputs of the system depends on the levels measured by the environment. If the waterlevel is less than the critical limit, *timer* remains unchanged. Otherwise the *timer* is started. If the actual time is less than the timer, the output `out` is 0 which in our system means that the gates should close. Otherwise the gates should open.

A perhaps more regular behaviour is obtained if the update disables (blocks) all environment actions.

Clock The clock component has to be specified as a single action. The clock gets the waterlevels from the environment and the open or close signals from the system as inputs. With every *tick* action the clock component makes these values visible to the outside world. This architecture has the consequence that changes of the environment, i.e. changes of the waterlevels, are immediately noticed by the system.

Fairness In order to model the scenario properly we need *fairness* constraints. As mentioned above, the update operation (made by the system) is always enabled and therefore *weak fairness* suffices. If the clock becomes enabled by the execution of an update[21], it remains so until it is executed, since the waterlevel cannot be changed again. In the general case where we have more than one environment action like *chw* (change waterlevel) it might happen that an action that has not yet been executed is taken. However, this can occur only finitely many times. Hence, also in the case of the clock we can use *weak fairness*.

It is not too difficult to prove that the clock ticks infinitely often, that changes of the environment lead to updates, and that changes of the waterlevel are enabled

[21] This does not hold for the first state.

infinitely often. Formally this is written as

$$\Box\Diamond\langle tick\rangle_{time}$$
$$\Box(enable - chw = true \rightarrow \Diamond\langle update\rangle_{enable-clock})$$
$$\Box\Diamond enabled - chw = true$$

Safety properties We consider the property: *When the waterlevel becomes higher than the triggerlevel, then the output signal is low for the time given by duration.* The formalisation of this assertion is

$$\Box((t_0 = time \wedge level > triggerlevel) \rightarrow$$
$$\Box((t_0 < time \leq t_0 + duration) \rightarrow out = 0))$$

Time We assume a global *discrete* time scale with a *clock* that increases the value of a flexible variable *time* (of type Nat) by one upon each tick. By a small example (different from the ECS) we illustrate the general approach and argue that it is applicable for many scenarios.

Figure 4.17 shows a possible behaviour of a system that reacts to an input signal (*signal*–1). A safety requirement could be the following: If *signal*–1 is high for at

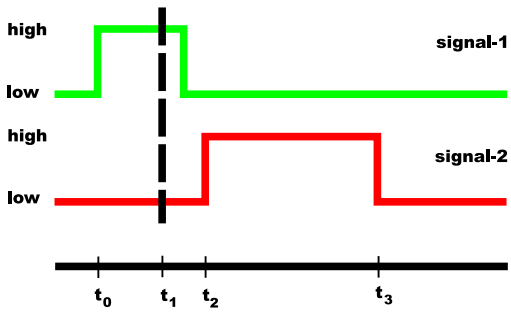

Figure 4.17: A possible Behaviour of a Real-Time System

least $\frac{1}{500}$ sec starting from t_0 and *signal*–2 is low at t_0, then after at most $\frac{1}{100}$ sec *signal*–2 will be high for at least $\frac{1}{4}$ sec. If we have the conditions

$$\begin{aligned} t_1 - t_0 &\geq \frac{1}{500} \text{ sec} \\ t_2 - t_0 &\leq \frac{1}{100} \text{ sec} \\ t_3 - t_2 &\geq \frac{1}{4} \text{ sec} \end{aligned}$$

for the time points t_0, t_1, t_2 and t_3, then this particular behaviour fulfils the requirement.

Our aim is to provide a formal specification of the system, (assumptions about) the environment, and the clock, where in more complicated cases the system specification might by structured into several components running in parallel. In a separate specification we formalise safety requirements like the one above. Since in the formal model we use a discrete time scale we have to replace concrete durations by constants that stand for an arbitrary but fixed number of time steps. The above mentioned requirement can be formulated as a temporal logic formula in VSE as follows:

$$\begin{aligned}
&\Box\forall t_0.((t_0 = time \wedge signal\text{–}2 = low \\
&\qquad \wedge \Box(t_0 \leq time \leq t_0 + d_1 \rightarrow signal\text{–}1 = high)) \\
&\rightarrow \\
&\exists t_2.((t_2 - t_0) \leq d_2 \wedge \Box((t_2 \leq time \leq t_2 + d_3) \rightarrow signal\text{–}2 = high)))
\end{aligned}$$

The premise that *signal*–1 is high for at least d_1 time units and that the allowed delay is d_2 time units take the fact into account that the system might need a certain time d_4 to react and also a certain time d_5 to compute new output values. In the case of ECS both reaction time and computation time are considered to be zero according to the given documents[22]. For d_3 it holds that $t_3 - t_2 \geq d_3$. In order to prove the timing requirements the specification has to make certain assumptions about relations between the durations mentioned. In our example the assumptions $d_4 \leq d_1$ and also $(d_4 + d_5) \leq d_1$ are required.

Having successfully proved the safety requirements we are free to choose concrete values for the durations according to the above mentioned constraints.

4.2.8.2 The Formal Model of the ECS in VSE-II

The specification of the ECS is modelled with VSE-II. The development graph of the specification is shown in Figure 4.18. The structure of the development graph is very similar to the description of the general scenario given in Figure 4.16. The three nodes `environment_data`, `SVKO_system` and `Update` from Figure 4.18 correspond to the `env`, `system` and `update` nodes of Figure 4.16, respectively. The node `SVKO_combine` in Figure 4.18 represents the composed system and consists of three concurrent components. The properties the system has to satisfy are specified in the temporal logic specification `SVKO_safety`. The whole VSE-II specification of the Emergency Closing System is shown in Appendix A.

Synchronisation According to the requirements[23] we started with, it can be assumed that the system immediately (i.e. without any delay) notices a change of

[22] These documents describe the desired behaviour of the ECS. They were written by the company that has implemented the first version of the control system.

[23] These requirements are stated in the original system description of the manufacturing company.

the input variables caused by the environment or the clock and instantaneously computes an output. Obviously, the concurrent execution of the three components has to be restricted appropriately to model this assumption. For example, if the environment changes the waterlevel and the clock ticks twice before the system has computed new output values, then there can be an intermediate state where a safety requirement is violated. So the clock has to be blocked until the system has finished its computation. Note that in cases where there are certain (restricted) reaction and computation times a similar, slightly more complicated scheduling regime is necessary to rule out behaviours where reaction or computation takes too much time. In all these situations *fairness*, which forces a step to be executed sometimes in the future, is not enough.

Technically the scheduling among the three components is realized by shared variables acting as guards. An action (step) of a component can only be executed if the guard is evaluated to *true*.

Unless there was a tick of the clock the variable *time* remains unchanged. When the environment has changed the input for the system and the system has computed the outputs, then the clock component should do a tick step. Otherwise there would be two different situations with the same time stamp.

But even if the clock ticks frequently enough for the intermediate states we may have different input values at the same time. Moreover, immediately after the environment has changed some input values, the output values of the system might not have the values requested by the requirements specification.

To overcome this problem we distinguish between internal variables and variables visible to the outside according to our general model. The environment, the system, and the clock change only internal variables. Whenever the clock ticks the corresponding visible variables are updated with the current values of the internal variables. The observable variables remain unchanged in the intermediate states mentioned above. If the internal variables are finally *hidden* (by existential quantification over flexible variables), no regular behaviour can be observed for them in the resulting system.

There are many ways of implementing these rules. We have chosen a liberal implementation shown in Figure 4.16. The underlying datatype definitions are made in the *theories* `StatFunctions` and `BasicDatas` (see Figure 4.18). The *theories* `natural` and `boolean` are predefined theories in the VSE-II system.

In the following we describe the specifications of the different components.

Environment Specification In our application scenario, the physical environment consists of the natural changes of waterlevels which have various complex causes. Since we do not want to specify these causes and since our main interest is in specifying the real-time behaviour of the `SVKO_system` we have specified the environment as abstract as possible. There is only one action which represents the change of the (inside and outside) waterlevel sensors. These changes are

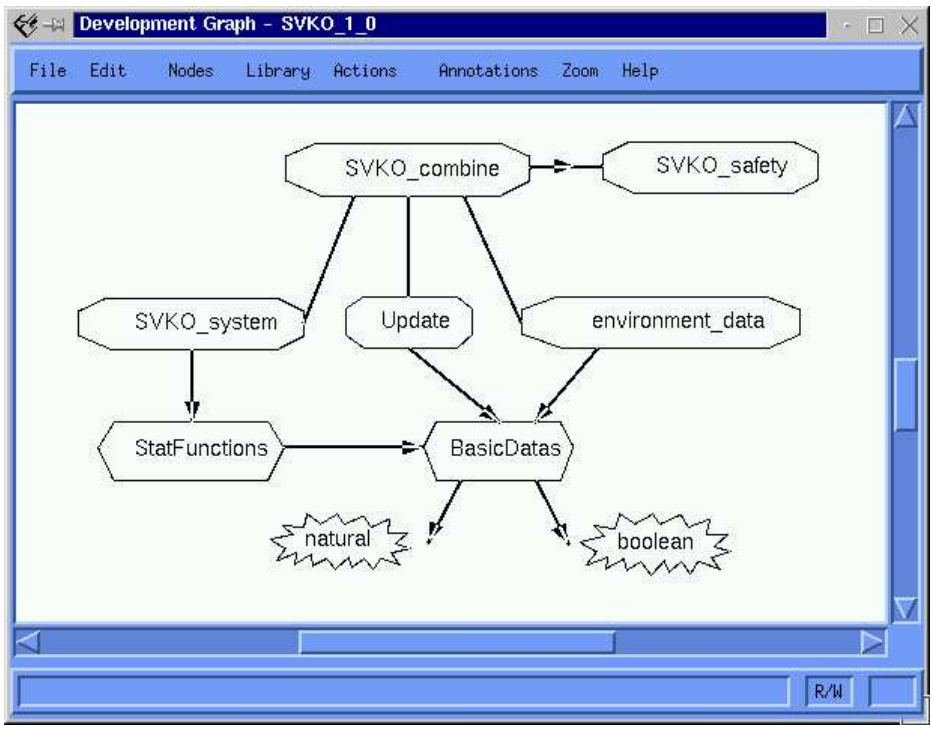

Figure 4.18: The VSE-II Development Graph of the ECS

transmitted to the `SVKO_system` and to the `Update` component.

System Specification The data delivered from the environment to the system are the values of the inside and outside waterlevel sensors. These data are used in the `SVKO_system` to compute the values of two signals: the `CLOSE` and the `OPEN` signal. If the `CLOSE` signal is true, then the system should close the gates and if the conditions for the `OPEN` signal are true, then the system should again open the gates.

The computation of the system is modelled in a single action and separated into two parts: a static and a dynamic part. In the static part we have modelled diagrams from the original (graphical) description of the ECS as abstract data-types and their corresponding functions. A very simple example is a 2/3 voter. It returns true if at least two of the three inputs are true and false otherwise. This voter is specified simply as a predicate `voter2from3` with the following axiomatic definition:

```
ALL sig1, sig2, sig3 :
    voter2from3(sig1,sig2,sig3) <->
    (sig1 = T AND sig2 = T) OR
    (sig2 = T AND sig3 = T) OR
    (sig1 = T AND sig3 = T)
```

The specification of the dynamic part is more complex. This is mainly concerned with the description of the timing behaviour of the system. To give an example think of a mono-stable multivibrator[24] which starts working by a high to low trigger and then keeps the output signal low for at least `d` time units and leaves the signal high in all other cases. To remember the values on the input line of the multivibrator and thus to determine whether such a trigger has happened, we have inserted a flexible variable (`CLOSE_TRIGGER`) which stores the "old" value on the line and the current value. Depending on the relationship between the old and the new value the multivibrator is active or not. This is specified by a timer (`timer_CLOSE`) which is set in case the multivibrator is active and which remains unchanged in all other cases. The following specification part models such a multivibrator:

```
IF (CLOSE_TRIGGER = T AND
    CLOSE_TRIGGER' = F)
THEN timer_CLOSE' = time + d
ELSE timer_CLOSE' = timer_CLOSE
```

In this formula we refer to a flexible variable `time` which represents the actual time in the system. This variable is sent from the `Update` component to the `SVKO_system` component.

Update Specification The specification of the `Update` component models mainly two properties of the whole system specification. First, it filters the signals visible to the outside world, and second, it represents the global clock and increments the *time* variable. The specification of the `update` component consists of a single action which increments the time in every step and makes the variables representing the `CLOSE` and the `OPEN` signal together with the variables representing the waterlevel sensor values visible to the outside world.

Property Specification The ECS should satisfy some safety properties which are important for the correct functioning of the system. We proved among others the following two properties:

1. $\Box(\neg(\texttt{OPEN} = \texttt{T} \wedge \texttt{CLOSE} = \texttt{T}))$
2. $\Box((time = t_0 \wedge Change\_Sensor\_Sig) \Longrightarrow$
   $\Box((t_0 < time \leq t_0 + d + 1) \Longrightarrow \texttt{CLOSE} = \texttt{T}))$

Property 1 says that the `OPEN` and the `CLOSE` signal are never true at the same time. Property 2 says that if the waterlevels get dangerous (expressed by the

[24] A monostable multivibrator knows exactly one stable state. If it is set to an unstable state, then it will turn back to the stable state after a certain amount of time except it is again set to the unstable state.

formula *Change_Sensor_Sig*) at time t_0, then the system reacts by setting the `CLOSE` signal to true for at least d time units beginning at time $t_0 + 1$.

The proofs of these properties are all local within the `SVKO_system` component. Property 1 can be be proved without assumptions whereas the proof of property 2 is more complex. It needs assumptions which have to be guaranteed by the environment of the `SVKO_system`.

4.2.8.3 Summary

We have presented a general methodology how to specify real-time systems in VSE-II and we have illustrated the methodology using a well-known example from the literature and also a "real life" scenario with a very high potential risk. The requirements engineering phase of the specification of the ECS used diagrams as e.g. Figure 4.17 to analyse the timing behaviour of the system. Although numerous tests have been performed, the formal analysis reveals some minor design and specification problems and in particular one severe specification error that results in the necessity to finally redesign the control unit.

Nevertheless, this way of analysing a real-time system is not very effective and satisfying, because we can only show one special property of such real-time systems. A more abstract and structured way would be to use timed automata [16] or hybrid automata [47] to specify the timing behaviour of a system. The integration of these automata into a formal software development process supported by VSE-II is shown in the following chapter 5.

4.3 Hybrid Automata

Hybrid Systems [16] are real-time systems which are embedded into an analog environment. They contain discrete and continuous components and interact with the physical world through sensors and actuators. Since they typically operate in safety-critical situations, rigorous techniques for analysis are of high importance.

A common model for hybrid systems can be found in *hybrid automata*. Briefly, such hybrid automata are finite graphs whose nodes correspond to global states. Such global states represent some sort of general observational situation, as, for instance, "the heater is on" or the "the heater is off". During these global states some continuous activity takes place. For example, coming back to the heater from above, depending on the global state, the temperature rises or falls continuously according to a dynamical law until there is a transition from one node to another.

These transitions are usually guarded with some constraint formula. that is required to hold if the transition is supposed to be taken. Similarly, nodes have some attached constraint formula that describes an invariant for this very node, i.e., some property that has to be true while the system resides within this node. The dynamics of the system's behaviour, on the other hand, is given by a description of how the data changes with time.

Additionally, transitions are annotated with a general assignment that is responsible for the discrete action to be performed by taking the transition.

In what follows we formally describe hybrid automata, i.e., we define their syntax as well as their semantics.

4.3.1 Syntax

As noted earlier, transition guards and node invariants are formulated by constraint formulae. These are defined as follows.

Definition 4.3.1 (Constraint Terms, Constraint Formulae)
The set CT of Constraint Terms *over a fixed variable set X is defined as the smallest set containing X, and a set RC of real-valued constants, and, moreover, it is closed under addition, subtraction, and multiplication with real-valued constants.*

The set CF of Constraint Formulae *(over the variable set X) is defined as the smallest set that is closed under conjunction and contains $\top$ (truth) and $\bot$ (falsity) as well as all atoms of the form $t_1 > t_2$, $t_1 \geq t_2$, $t_1 < t_2$, $t_1 \leq t_2$, and $t_1 = t_2$, where t_1 and t_2 are constraint terms taken from CT.*

Such constraint terms and formulae are a prerequisite for the formal description of the hybrid automata which are as follows.

Definition 4.3.2 (Hybrid Automata)
Hybrid Automata *are tuples of the form $(X, \mathcal{L}, \mathcal{E}, dif, inv, guard, act)$, where*

- X *is a finite set of real-valued* data variables,
- $\mathcal{L}$ *is a finite set of* locations, *i.e. nodes of a graph,*
- $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ *is a finite (multi)set of* transitions, *i.e. edges of the graph with nodes from* $\mathcal{L}$,
- $dif\colon \mathcal{L} \times X \mapsto CT$ *is a mapping that associates with each location and each data variable a constraint term (with free variables taken from* X*), representing the change of the data variable within this location over time,*
- $\text{inv}\colon \mathcal{L} \mapsto CF$ *is a mapping that associates with each location a constraint formula (with free variables taken from* X*), representing the location invariant,*
- $guard\colon \mathcal{E} \mapsto CF$ *is a mapping that associates with each edge a constraint formula (with free variables taken from* X*), representing the condition that has to hold in order to travel along the edge, and*
- $act\colon \mathcal{E} \times X \mapsto CT$ *is a mapping that associates with each edge and each data variable a constraint term (with free variables taken from* X*), representing the value of the variable after travelling along the edge.*

As already noted earlier, we illustrate hybrid automata as graphs. As an example we consider the gasburner example (see section 3).

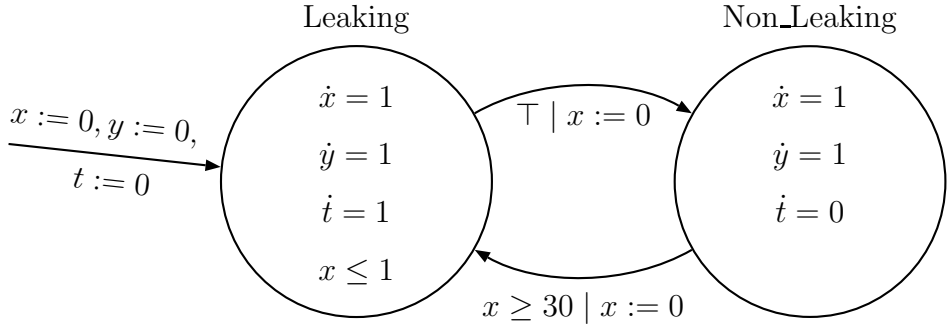

Figure 4.19: Gasburner as Hybrid Automaton

The graphical representation of the gasburner in Figure 4.19 describes its formal realisation with a Hybrid Automaton.

The nodes Leaking and Non_Leaking represent discrete locations, whereas x, y and t are data variables. Within each location we describe the location invariant ($x \leq 1$ in location Leaking) and the continuous activity which describes how the values of the data variables change in time. In the example above the value of x increases by 1 per time unit (say, second), i.e., the first derivative of the function describing the behaviour of x over time is the constant 1.

Edges are annotated with guards and discrete actions. Guards form a constraint on the data variables to hold if a transition via the corresponding edge is to be performed. The discrete action specifies how the data variables are to be changed after taking the transition. In the above example the guard of the edge from Non_Leaking to Leaking is $x \geq 30$ and the corresponding action is to set x to 0.

The above hybrid automaton thus describes the following behaviour: it starts at location Leaking with data variables all set to 0. Within these locations the value of the data variables increases by 1 every second. However, there is one exception. The clock t has slope 0 in location Non_Leaking, therefore it stopped there.

The automaton leaves location Leaking the latest after one second and resets x to 0. Similarly, it remains within Non_Leaking for at least 30 seconds and reenters Leaking after having reset x to 0 again.

The clock x, therefore, acts as a control variable, y accumulates overall time and t counts leakage time (the amount of time that the system resides within Leaking).

4.3.2 Semantics

The semantics of hybrid automata is defined by a formal description of all possible "computations" of the automaton. These are sequences of states that represent the temporal behaviour.

Definition 4.3.3 (State)
We define a state *of a hybrid system as a pair* (L, ϕ)*, where* $L \in \mathcal{L}$ *is a location and* $\phi\colon X \mapsto \mathbb{R}$ *is a valuation of the data variables.*

ϕ extends to (constraint) terms and (constraint) formulae as follows.

Definition 4.3.4
Let $x \in X$ *and* $c \in RC$*, where* RC *is a set of real-valued constants. Then*

- *If* $x \in X$*, then* $\phi(x)$ *is defined as in Definition 4.3.3.*
- *If* $c \in RC$*, then* $\phi(c) = c$*.*
- *If* $t_1, t_2 \in CT$*, then* $\phi(t_1 \circ t_2) = \phi(t_1) \circ \phi(t_2)$*, where* $\circ \in \{+, -\}$*.*
- *If* $c \in RC$ *and* $x \in X$*, then* $\phi(c * x) = c * \phi(x)$*.*

Definition 4.3.5
Let P_1 *and* P_2 *be constraint formulae and* t_1*,* t_2 *be constraint terms, then* ϕ *is extended to constraint formulae as follows:*

- $\phi(\top)$ *equals true and* $\phi(\bot)$ *equals false.*
- $\phi(t_1 \circ t_2)$ *is true if and only if* $\phi(t_1) \circ \phi(t_2)$ *holds, where* $\circ \in \{<, \leq, >, \geq, =\}$*.*

- $\phi(P_1 \wedge P_2)$ *is true if and only if* $\phi(P_1)$ *and* $\phi(P_2)$ *hold.*

DEFINITION 4.3.6 (ADMISSIBLE)
A state (L, ϕ) *is called* admissible *if* $\phi(\text{inv}(L))$ *holds, i.e. if the valuation of the data variables does not violate the location invariant.*

DEFINITION 4.3.7 (TRANSITION-REACHABLE)
Given two admissible states $\sigma = (L, \phi)$ *and* $\sigma' = (L', \phi')$ *we say that* σ' *is* transition-reachable *from* σ *– denoted by* $\sigma \overset{\text{tr}}{\mapsto} \sigma'$ *– if there exists a transition* $T = (L, L') \in \mathcal{E}$ *with source* L *and target* L'*, and both* $\phi(\text{guard}(T))$ *and* $\phi'(x) = \phi(\text{act}(T, x))$ *for each* $x \in X$*.*

In other words, σ' is transition-reachable from σ (via transition T) if the corresponding guard is true and the valuation of the data variables in σ' is changed according to the discrete transition action.

DEFINITION 4.3.8 (TIMELY-REACHABLE)
We call $\sigma' = (L', \phi')$ timely-reachable *from* $\sigma = (L, \phi)$ *with delay* δ *– denoted by* $\sigma \overset{\delta}{\mapsto} \sigma'$*, where* δ *is a non-negative real number – if*

1. $L = L'$ *and*
2. *for each* $x \in X$ *there exists a differentiable function* $f_x \colon [0, \delta] \mapsto \mathbb{R}$*, with the first derivative* $\dot{f}_x \colon (0, \delta) \mapsto \mathbb{R}$*, such that*
   (a) $f_x(0) = \phi(x)$ *and* $f_x(\delta) = \phi'(x)$ *and*
   (b) *for all* $\epsilon \in \mathbb{R}$ *with* $0 < \epsilon < \delta$*: both* $\text{inv}(L)[x_1/f_{x_1}(\epsilon), \ldots, x_n/f_{x_n}(\epsilon)]$ *and* $\dot{f}_x(\epsilon) = \text{dif}(L, x)[x_1/f_{x_1}(\epsilon), \ldots, x_n/f_{x_n}(\epsilon)]$ *are true.*

σ' *is* timely-reachable *from* σ *– denoted by* $\sigma \overset{\star}{\mapsto} \sigma'$ *– if there exists a non-negative* $\delta \in \mathbb{R}$ *such that* $\sigma \overset{\delta}{\mapsto} \sigma'$*.*

DEFINITION 4.3.9 (REACHABLE)
σ' *is said to be* reachable *from* σ *with respect to* $\mathcal{H}$*,* $\sigma \underset{\mathcal{H}}{\mapsto} \sigma'$*, if* $(\sigma, \sigma') \in (\overset{\star}{\mapsto} \cup \overset{\text{tr}}{\mapsto})^*$*.*

DEFINITION 4.3.10 (RUN)
A run ρ *of* $\mathcal{H}$ *with initial state* $\sigma_0 = (L_0, \phi_0)$ *is a sequence of states represented as*

$$\rho = \sigma_0 \overset{t_0}{\underset{f_0}{\mapsto}} \sigma_1 \overset{t_1}{\underset{f_1}{\mapsto}} \sigma_2 \overset{t_2}{\underset{f_2}{\mapsto}} \sigma_3 \overset{t_3}{\underset{f_3}{\mapsto}} \cdots$$

where $t_i \in \mathbb{R}^{\geq 0}$ *and* $f_i \colon [0, t_i] \mapsto (X \mapsto \mathbb{R})$*, such that*

1. $f_i(0) = \phi_i$ *and*

2. *inv*$(L_i)[x_j/f_i(t)(x_j)]$ *holds for all* $0 \leq t \leq t_i$, $x_j \in X$, $(L_i, f_i(t_i)) \overset{\text{tr}}{\mapsto} \sigma_{i+1}$ *and for all* $0 \leq t' \leq t' + \delta \leq t_i : (L_i, f_i(t')) \overset{\delta}{\mapsto} (L_i, f_i(t' + \delta))$.

Given such a run ρ, we call the state σ_0 the *starting state* of ρ and denote it by start(ρ). The set of states contained in ρ is given as States(ρ) = $\{(L_i, f_i(t)) \mid t \in \mathbb{R}, 0 \leq t \leq t_i\}$. The set of all runs of a hybrid system $\mathcal{H}$ with initial state σ is denoted by runs($\mathcal{H}, \sigma$).

Definition 4.3.11 (position)
A position π *of a run* $\rho = \sigma_0 \mapsto^{t_0}_{f_0} \sigma_1 \mapsto^{t_1}_{f_1} \sigma_2 \mapsto^{t_2}_{f_2} \sigma_3 \mapsto^{t_3}_{f_3} \cdots$ *is a pair* $\pi = (i, r) \in \mathbb{N} \times \mathbb{R}$ *such that* $0 \leq r \leq t_i$.

We denote the set of positions of a run ρ as pos(ρ). Positions are ordered lexicographically, i.e. $(i, r) < (j, s)$ if and only if $i < j$ or ($i = j$ and $r < s$). Also, $(i, r) \leq (j, s)$ if and only if $(i, r) < (j, s)$ or ($i = j$ and $r = s$). By $\rho(\pi)$ with $\pi = (i, r)$ we denote the state $(L_i, f_i(r))$. Thus, States(ρ) = $\{\rho(\pi) \mid \pi \in \text{pos}(\rho)\}$.

Definition 4.3.12 (suffix)
Given a run $\rho = (L_0, \phi_0) \mapsto^{t_0}_{f_0} (L_1, \phi_1) \mapsto^{t_1}_{f_1} \cdots$, *an* $i \geq 0$, *and a* t *with* $0 \leq t \leq t_i$, *we define the* suffix *of* ρ, *starting at* $(L_i, f_i(t))$ *as*

$$(L_i, f_i(t)) \mapsto^{t_i - t}_{f'_i} (L_{i+1}, \phi_{i+1}) \mapsto^{t_{i+1}}_{f_{i+1}} (L_{i+2}, \phi_{i+2}) \mapsto^{t_{i+2}}_{f_{i+2}} \cdots$$

where $f'_i(t') = f_i(t + t')$. *By* suf($\rho\prime, \rho$) *we indicate that* $\rho\prime$ *is a suffix of* ρ.

Lemma 4.3.1 *Let* ρ_1, ρ_2 *be two runs with* suf(ρ_1, ρ_2). *Then there exists a position* (i, r) *with* $(i, r) \in \text{pos}(\rho_2)$ *such that*

$$\begin{aligned} \text{start}(\rho_1) &= \rho_2((i, r)) \\ \rho_1((i', r')) &= \rho_2((i + i', r')) \text{ for each } (i', r') \in \text{pos}(\rho_1), i' \geq 1 \\ \rho_1((0, r')) &= \rho_2((i, r + r')) \text{ for each } (0, r') \in \text{pos}(\rho_1) \end{aligned}$$

Definition 4.3.13 (non-zeno)
A run is said to be non-zeno *if* $\sum_{i=0}^{n} t_i$ *diverges.*

In general, it is assumed that the runs of the hybrid system under consideration are all non-zeno.

Within hybrid automata data variables change both continuously and discretely. The change is given by some rational number where the underlying base unit is in general neglected. Nevertheless, we usually have certain base units in mind when we specify systems, be it seconds, meters, or what have you. However, there might be a need for switching the base unit, for example from seconds to milliseconds. One reason for such a need will be found in Section 5 where we present the final translation step from Hybrid Automata to VSE-II specifications. In the following we define such a change of the base units formally.

DEFINITION 4.3.14 (GRANULARITY CHANGE)
Given a Constraint Term $\mathcal{T}$ we define the granularity change *of $\mathcal{T}$ by Δ as*

$$\Delta \star x = x \text{ for } x \in X$$
$$\Delta \star (i * x) = i * x \text{ for } x \in X, i \in \mathbb{Q}$$
$$\Delta \star i = \Delta * i \text{ for } i \in \mathbb{Q}$$
$$\Delta \star (t_1 + t_2) = \Delta \star t_1 + \Delta \star t_2$$

where $\mathbb{Q}$ represents the rationals.

For constraint formulae $\mathcal{F}$ "$\star$" distributes in an natural way.

For some given hybrid automaton, $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, inv, guard, act)$, we define

$$\Delta \star \mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, inv', guard', act')$$

where

- $inv'(L) = \Delta \star inv(L)$ for all $L \in \mathcal{L}$,
- $guard'((L_1, L_2)) = \Delta \star guard((L_1, L_2))$ for all $(L_1, L_2) \in \mathcal{E}$, and
- $act'((L_1, L_2), x) = \Delta \star act((L_1, L_2), x)$ for all $(L_1, L_2) \in \mathcal{E}$, $x \in X$.

Moreover, we extend the notion of granularity change to runs as follows: Given

$$\rho = (L_0, \phi_0) \mapsto_{f_0}^{t_0} (L_1, \phi_1) \mapsto_{f_1}^{t_1} (L_2, \phi_2) \cdots$$

then $\Delta \star \rho$ is defined as

$$(L_0, \Delta \star \phi_0) \mapsto_{\Delta \star f_0}^{\Delta * t_0} (L_1, \Delta \star \phi_1) \mapsto_{\Delta \star f_1}^{\Delta * t_1} (L_2, \Delta \star \phi_2) \cdots$$

where $(\Delta \star f_i)(\Delta * t) = \Delta \star f_i(t)$ and $(\Delta \star \phi_i)(x) = \Delta * \phi_i(x)$.

This change of the granularity is used within the proof of the correctness of the translation of hybrid automata to VSE-II.

A typical procedure to prove temporal safety properties of hybrid automata is to compute all the states that are reachable from the initial state and to check the temporal property against this set of reachable states. This obviously requires that this set of reachable states is finitely computable and representable[25].

[25] In this work we shall not go into details what this verification approach is concerned. We rather emphasise on how to use the results of the hybrid automata verification in the VSE-II system and vice versa.

4.3.3 Property Specifications

In the following we describe the syntax and the semantics of the (linear temporal logic) property specification language (PSL) we utilise in this work. In chapter 5 the general scenario illustrated in Figure 5.1 indicates the usage of PSL and it shows that the specification languages PSL and VSE-SL correspond in some sense.

Syntax of PSL We assume a set CT of constraint terms and a set CF of constraint formulae as defined in Definition 4.3.2. We add to the set CF constraint formulae of the form $state = L$ with $L \in \mathcal{L}$ where *state* is a reserved keyword of PSL. Furthermore, we extend the definition of the set CF by assuming a variable set $X \cup \{state\}$ where *state* is new to X. The rest of the definition of the PSL syntax is as follows:

Definition 4.3.15 (Syntax of PSL)

1. *Any constraint formula $P \in CF$ is a formula of PSL.*
2. *If P and Q are formulae of PSL, then so are $P \wedge Q$ and $P \vee Q$.*
3. *If P is a formula of PSL, then so are $\Box P$ and $\Diamond P$.*
4. *The expression $state = L$ is a PSL formula, where $L \in \mathcal{L}$ and state is a special variable not occurring in X.*

We extend the notion of granularity change to PSL formulae as follows[26]:

$$\Delta \star (P_1 \vee P_2) = \Delta \star P_1 \vee \Delta \star P_2,$$
$$\Delta \star (P_1 \wedge P_2) = \Delta \star P_1 \wedge \Delta \star P_2,$$
$$\Delta \star (\Diamond P) = \Diamond(\Delta \star P)$$
$$\Delta \star (\Box P) = \Box(\Delta \star P)$$
$$\Delta \star (state = L) = (state = L)$$

After the definition of the syntax of PSL we come to the definition of the semantics.

Semantics of PSL The semantics of PSL formulae are defined with respect to a hybrid automaton $\mathcal{H}$ and a run ρ of that automaton and in accordance with Definition 4.3.2.

[26] Note that the case of a constraint formula is already covered by Definition 4.3.14.

DEFINITION 4.3.16 (SEMANTICS OF PSL)
Let $\mathcal{H}$ be a hybrid automaton and σ be its initial state, then the semantics of PSL formulae is defined as follows:

1. *($\mathcal{H}$, σ)* $\models P$ **iff** $(\mathcal{H}, \rho) \models P$ *for all* $\rho \in \text{runs}(\mathcal{H}, \sigma)$

2. *($\mathcal{H}$, ρ)* $\models state = L$ **iff** $\text{start}(\rho) = (L, \phi)$ *for some* ϕ.

3. *($\mathcal{H}$, ρ)* $\models t_1 \circ t_2$ **iff** $\text{start}(\rho) = (L_1, \phi_1)$ *with* t_1, $t_2 \in \mathcal{CT}$, $\circ \in \{<, \leq, >, \geq, =\}$ *and* $\phi_1(t_1) \circ \phi_1(t_2)$.

4. *($\mathcal{H}$, ρ)* $\models P \wedge Q$ **iff** *($\mathcal{H}$, ρ)* $\models P$ *and ($\mathcal{H}$, ρ)* $\models Q$

5. *($\mathcal{H}$, ρ)* $\models P \vee Q$ **iff** *($\mathcal{H}$, ρ)* $\models P$ *or ($\mathcal{H}$, ρ)* $\models Q$

6. *($\mathcal{H}$, ρ)* $\models \Box P$ **iff** *($\mathcal{H}$, $\bar{\rho}$)* $\models P$ *for every* $\bar{\rho}$ *with* $\text{suf}(\bar{\rho}, \rho)$

7. *($\mathcal{H}$, ρ)* $\models \Diamond P$ **iff** *($\mathcal{H}$, $\bar{\rho}$)* $\models P$ *for some* $\bar{\rho}$ *with* $\text{suf}(\bar{\rho}, \rho)$

We have defined a linear time semantics for PSL formulae according to the linear time semantics which is given to VSE-II specifications. In the literature, for example in [23], property specification languages for hybrid automata are often given a branching time semantics. There are two more temporal operators with the following (informal) semantics:

- $\exists\Box P$ holds **iff** P holds in any state of some run.

- $\exists\Diamond P$ holds **iff** P holds in some state of some run.

Because of the fact that we use linear time semantics in VSE-II, it is more convenient to have also linear time semantics on the hybrid systems side in order to define a translation function from one to the other.

Our experience in the field of formal specification and verification made by specifying and verifying safety and security critical systems shows us that many properties are of the kind $\Box\,\phi$, $\Diamond\,\phi$ or $\Box\Diamond\,\phi$. Therefore, we have chosen to take a linear time semantics for hybrid automata. Nevertheless, we know that we lose the ability to express properties of the kind that there is a state in some behaviour where a property does not hold (description of a counterexample). This point is related to further work to be done in this field. It is concerned with the extension of the VSE-II semantics to branching time which includes the insertion of new temporal operators and the extension of the current calculus available in the VSE-II deduction unit.

4.3.4 Example Proof with Hybrid Automata

In Section 4.2.7 we have presented how the property *safe* $\Box(y \geq 60c \Rightarrow 20t < y)$ of the gasburner can be specified and verified using the VSE-II system. In this section we are going to present how it is proved using hybrid automata. However, the property we are going to prove here is syntactically not the same:

$$\Box(y \geq 60 \Rightarrow 20t < y)$$

The only difference is that there is no multiplication of the number 60 with the constant c in this formula[27]. Here we assume some ground unit for constants like 60 (for example seconds). Although it is of no importance for the sequel, let us assume for the moment that we mean sixty seconds in writing 60 in formulae like above.

The informal description of the gasburner given as a graph in Figure 4.19 is now converted to a formal description according to Definition 4.3.2:

Let

- $X = \{x, y, t\}$,
- $L = \{Leaking, Non\_Leaking\}$,
- $E = \{(Leaking, Non\_Leaking), (Non\_Leaking, Leaking)\}$, and
- *init* be defined as $\{(Leaking, (x := 0, y := 0, t := 0))\}$

then *dif*, *inv*, *guard* and *act* are defined as follows:

$dif(Leaking, x) = 1$	$dif(Leaking, y) = 1$
$dif(Leaking, t) = 1$	$dif(Non\_Leaking, x) = 1$
$dif(Non\_Leaking, y) = 1$	$dif(Non\_Leaking, t) = 0$
$inv(Leaking) = x \leq 1$	$inv(Non\_Leaking) = \top$
$guard(Leaking, Non\_Leaking) = \top$	$guard(Non\_Leaking, Leaking) = x \geq 30$
$act((Leaking, Non\_Leaking), x) = 0$	$act((Leaking, Non\_Leaking), y) = y$
$act((Leaking, Non\_Leaking), t) = t$	$act((Non\_Leaking, Leaking), x) = 0$
$act((Non\_Leaking, Leaking), y) = y$	$act((Non\_Leaking, Leaking), t) = t$

To analyse whether the property *safe* holds for this hybrid automaton, we may utilise all the proof procedures which are in some sense "compatible" to our semantics of hybrid automata. One of the best known proof procedures is HyTech

[27] Why there is a c in the formula *safe* is explained in detail in Section 5.

[48]. There are other techniques presented in the literature as for example in [79] which could have been taken as well.

In the following we first present the HyTech input file of the gasburner, then we explain the input language according to this example[28].

```
-- leaking gasburner
var x,           -- time spent in current location
    y: clock;    -- total elapsed time
    t: stopwatch -- leakage time

automaton gasburner
synclabs:;
initially leaking & t=0 & x=0 & y=0;
loc leaking: while x<=1 wait {dt=1 & dx=1 & dy=1}
             when True do {x'=0} goto non_leaking;
loc non_leaking: while True wait {dt=0 & dx=1 & dy=1}
             when x>=30 do {x'=0} goto leaking;
end

var init_reg, final_reg,b_reachable: region;

init_reg := loc[gasburner] = leaking & x=0 & t=0 & y=0;
final_reg := y>=60 & t>=1/20 y;
b_reachable := reach backward from final_reg endreach;

If empty(b_reachable & init_reg)
   then prints ``Non_leaking duration requirement satisfied'';
   else prints ``Non_leaking duration requirement not satisfied'';
endif;
```

Figure 4.20: Gasburner as HyTech Specification

The explanations we give here are mainly taken from the HyTech user guide [48]. For a detailed description on how to use and how to write input files the reader is best referred to this guide. The description of our automaton is separated into the description of the system and the analysis commands:

- Input language: System description:
  - Variables: In the `var` slot the variables are declared. The variable types used here are `clock` and `stopwatch` (see the variable declarations for

[28]Note that there are many other possibilities and language constructs in the HyTech input language which are not used in this example and therefore not presented here.

`x`, `y` and `t`). A `clock` variable always has rate 1 whereas the rate of a stopwatch must be 1 or 0.

- Linear constraints: A linear constraint is an inequality (`<, <=, >, >=`) or equality (`=`) between linear expressions. Note that rational coefficients must either be an integer, have an integer as numerator and a nonzero integer as denominator or be omitted, in which case it is understood to be 1. An example for such a linear constraint could be found in the definition of the location `Leaking`.
- Automaton components: Each automaton is given a name which may be used later in the specification. Its synchronisation labels[29] are declared. Its location and the initial condition on its variables must also be provided. In our example the name of the automaton is `gasburner`.

  Each automaton component (there is only one in the gasburner definition) includes a list of locations that is described in Figure 4.20 and terminated by the keyword `end`.
- Locations: Each location is named and labelled with its invariant. Rate conditions may also be provided where the term `dx` is used to denote $\dot{x}$.

  Invariants are conjunctions of linear constraints. Each location is associated with a list of transitions originating from it.
- Transitions: Each transition lists a guard on enablement and the successor location. Both the synchronisation label and the update information are optional.

- Input language: Analysis commands:
  - Region expressions: Region expressions in our example are built from linear inequalities, constraints on locations, and region names, reachability, and conjunction.
    - Linear inequalities: The most basic region expression is a linear inequality. For example `y>=60` is a region expression defining the set of all states where the variable `y` has a value greater than or equal to 60.
    - Location constraints: For example `loc[gasburner]=leaking` is a location constraint consisting of the location name `leaking` in the automaton `gasburner`.
    - Boolean combinations: The disjunction and conjunction (`&`) of region expressions is a region expression.
    - Reachability: There are two specialised expressions for returning the set of states reachable from any arbitrary region: one for forward reachability and one for backward reachability. In the gasburner example backward reachability is used.

[29] These labels are used to compute the composition out of the automaton components.

- Boolean expressions: The unary predicate `empty` applied to a region expression evaluates to true if and only if its argument contains no states.
- Command statements: There are commands to perform common tasks such as error-trace. Command statements are built from primitives for printing and assigning errors. The only command statement used in the gasburner is `prints` which prints its argument.

Now we are ready to give the gasburner specification as input to HyTech. It computes the following output:

```
Non_leaking duration requirement satisfied.
```

Looking at the temporal logic representation of the property we want to prove

$$\Box(y \geq 60 \Rightarrow 20t < y)$$

we can see that it differs from the definition of the `final_reg` region. That is because the applied strategy was backward reachability. This means that we start in some arbitrary state where `final_reg` holds. From this state we go back and try to reach a state in which `init_reg` holds. If this succeeds we know that there is at least one trace of the automaton which reaches a state where `final_reg` holds. Then the evaluation of `empty` on `b_reachable` and `init_reg` yields `True`. However, the formula used to define `final_reg` was the negated original formula. What we have shown now is that from a state in the region `init_reg` there is no way to a state in the region `final_reg`. It follows that in all states reachable from `init_reg` the negated `final_reg` formula holds and we are done.

Now, as can be seen in the next chapter, our concern is to connect such a technique with the VSE-II system in an integrated way. Such a connection would have advantages for both sides. On the one hand, the use of techniques as model checking in VSE-II means that we introduce automation into an interactive verification system. On the other hand, we give a tool like HyTech the possibility to use the results of interactive verification techniques where induction plays an important role. In the sequel we describe the theoretical aspects of this connection and how it is achieved.

5

Integration of Hybrid Automata and VSE-II

So far we introduced VSE-II and (linear) hybrid automata using the specification of the gasburner as an example. Our aim now is to embed linear hybrid automata into VSE-II.

Integrating Hybrid Automata into VSE-II is an interesting theoretical task. But there is also a practical perspective. Applying formal methods in industrial sized systems often results in complicated specifications and large proofs and the specification of real-time constraints does not ease the task of the software engineer either. So we need a methodology which helps to control this complexity. The VSE-II system supports the whole formal software development process, but it does not explicitly support the specification of real-time systems. Hence the need for an integration and our methodology is illustrated in Figure 5.1. Starting point is the specification of the behaviour of a system with special emphasis on real-time using a hybrid automaton `Hybrid-AutSpec` as indicated in Figure 5.1. Properties that should be satisfied by this automaton are specified in `Hybrid-PropSpec`. The idea is now to translate the hybrid automaton specification into a VSE-SL specification. From this translated specification, we want to prove that the properties described in `VSE-PropSpec` hold. A part of these properties is created by translating the

properties in `Hybrid-PropSpec` into VSE-SL properties. The properties can be proved in the VSE-II tool. The proof can equally be done with a tool supporting hybrid automata. In that case we can guarantee (see Theorem 5.2.18) that the translated VSE-SL specification also fulfils these properties.

The advantages of such a method are:

1. A better support for the specification and verification of real-time systems and properties.
2. A more adequate choice of means to specify and verify systems.
3. Integration of a fully automated technique into the VSE-II system.

Coming back to the gasburner example given in Section 3 and its realisation with VSE-II and Hybrid Automata in sections 4.2.7 and 4.3.4 respectively, we make a very simple observation. Whereas in the proof of the property *safe* in VSE-II we have to invent an invariant, in the Hybrid Automata approach we have to decide which proof strategy (forward or backward reachability analysis) is to be applied. In such a "push button technology" the proof procedure automatically computes the answer, provided it can handle the problem at all[1]. For problems where the procedure does not succeed, techniques like abstraction, as for example presented in [14], have to be applied. Such techniques need in general an interactive proof system in order to show the correctness of the abstraction. Our method gives a user the possibility to decide which technique to apply or to use both techniques in an interleaved way. A scenario for this is the following. There are problems where model checkers are not able to prove the desired property without using additional lemmata or abstractions. In our approach we can first translate the hybrid automaton into VSE-SL. There we can prove a lemma that helps to find a proof for the property under consideration since VSE-II is not limited in the way model checkers are. It is clear that this process needs user interaction because already finding such a lemma (invariant) is not a trivial task in general. This lemma then may help the Hybrid Automata tool to prove the more general property. The scenario also works the other way round. The proof results for a hybrid automaton using a tool like HyTech can be used in VSE-II without additional proof work. In this way it is possible to work in an interleaved way where both sides gain their benefits.

It is clear that we inherit the usual restrictions on automated methods as model checking (usually they use a restricted form of induction, for example fixpoint computations).

As indicated in Figure 5.1 we have to define the "satisfies"- and the "translation"-relation between the given specifications. "Satisfies" in both cases means

[1]Usually model checking strategies are limited to finite state problems in the sense that the underlying fixpoint computation is guaranteed to terminate. Hybrid Systems are not finite state systems in this strict sense.

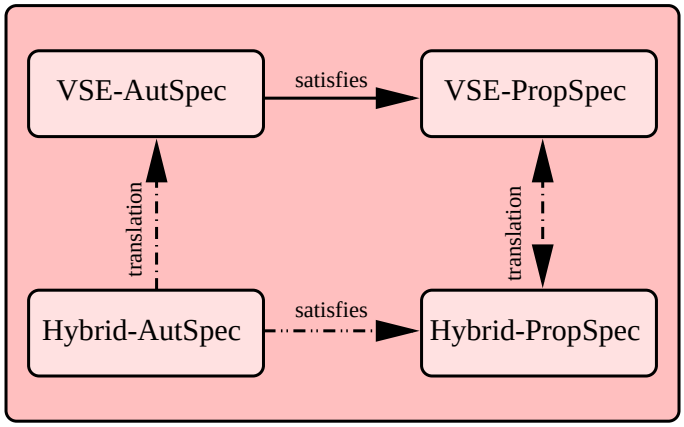

Figure 5.1: Relation: VSE-II- Hybrid Automata- Specification

model inclusion. In the VSE-II setting this is expressed by a satisfies-link in the real specification as can be seen for example in Figure 4.13. In the hybrid automaton setting the satisfies-link indicates that the hybrid automaton fulfils the specified property. How this is implemented ([48, 78, 79]) is not our main concern at this point, though.

5.1 Translating Hybrid Automata to VSE-II

The system we are considering is a hybrid automaton together with a definition of an initial state and a description of properties of that automaton. The translation has to be defined for all of these three parts.

5.1.1 Translation Function

Since hybrid automata as introduced in Section 4.3 do not have an explicit initial location, we have to extend the definition accordingly. We assume that there is a set *init* of pairs of type (CF, L) where every pair marks an admissible initial location L with a predefined variable valuation represented by a constraint formula CF. In our applications the set *init* consists of only one element. The definition of the translation is divided into the translation of *init*, the translation of the continuous actions and the translation of the discrete actions.

We use π as the name of the translation function. As can be seen in the next definitions we have overloaded π in order to make the definitions more readable. It will always be clear from the context which π is meant.

We first define π on constraint terms and constraint formulae. This definition includes the main ideas of the translation of hybrid systems into VSE-SL specifications. It is the technical kernel of the translation. As we shall see later there will

be a quantification over the (rigid) variables c and n, occurring in the translation of constraint formulae, depending on the property to be proved.

DEFINITION 5.1.1 (TRANSLATION OF CONSTRAINT FORMULAE)
Let $f_1, f_2 \in CF$, $t_1, t_2 \in CT$, $x \in X$, $e \in \mathbb{Q}$ and c, n be rigid variables of type $\mathbb{N}$. Then π is defined as follows:

$$\begin{aligned}
\pi(f_1 \wedge f_2) &\doteq \pi(f_1) \wedge \pi(f_2) \\
\pi(t_1 \circ t_2) &\doteq \pi(f_1) \circ \pi(f_2), \text{ with } \circ \in \{=, <, >, \leq, \geq\} \\
\pi(\top) &\doteq TRUE \\
\pi(\bot) &\doteq FALSE \\
\pi(t_1 \circ t_2) &\doteq \pi(t_1) \circ \pi(t_2), \text{ with } \circ \in \{+, -\} \\
\pi(-t_1) &\doteq -\pi(t_1) \\
\pi(e * x) &\doteq e * x \\
\pi(x) &\doteq x \\
\pi(e) &\doteq c * n * e
\end{aligned}$$

EXAMPLE 5.1.1 *Let $x, y \in X$ be variables. Then $\pi(x \leq 1) \doteq x \leq c * n$ and $\pi(3 * x + 2 \leq y) \doteq 3 * x + 2 * c * n \leq y$.*

DEFINITION 5.1.2 (TRANSLATION OF INITIAL STATES AND AUTOMATA)
Let init be a one element set $\{(f, L)\}$ representing the initial state of an automaton with $f \in CF$ and $L \in \mathcal{L}$, and let $\{x, x_1, \ldots, x_n\}$ be data variables and L_1, L_2 locations. Then we define π using the following formulae:

$$\pi(\{(f, L)\}) \doteq \pi(f) \wedge state = L$$

$$\phi \equiv \bigvee_{(L_1, L_2) \in \mathcal{E}} \Big(state = L_1 \wedge state' = L_2 \wedge \pi(guard((L_1, L_2))) \wedge \bigwedge_{x \in X} (x' = \pi(act((L_1, L_2), x))) \wedge (\pi(inv(L_2)))[x_1/x_1', \ldots, x_n/x_n'] \Big)$$

$$\phi^* \equiv \bigvee_{(L_1, L_2) \in \mathcal{E}} \Big(state = L_1 \wedge state' = L_2 \wedge \pi(guard((L_1, L_2))) \wedge \neg(\pi(inv(L_1)))[x_1/x_1 + dif(L_1, x_1), \ldots, x_n/x_n + dif(L_1, x_n)] \wedge \bigwedge_{x \in X} (x' = \pi(act((L_1, L_2), x))) \wedge (\pi(inv(L_2)))[x_1/x_1', \ldots, x_n/x_n'] \Big)$$

$$\psi \equiv \bigvee_{L \in \mathcal{L}} \Big(state = L \wedge state' = L \wedge \bigwedge_{x \in X} (x' = x + dif(L, x)) \wedge (\pi(inv(L)))[x_1/x_1 + dif(L, x_1), \ldots, x_n/x_n + dif(L, x_n)] \Big)$$

$$stuttering \equiv \bigwedge_{x \in X} x' = x$$

The translation of a hybrid automaton including an initial condition (and treating the safety part of the translation) is given by

$$\pi(\mathcal{H}, init) \doteq \pi(init) \wedge \Box(\phi \vee \phi^* \vee \psi \vee stuttering)$$

a formula in VSE-SL normal form.

An initial state of a hybrid automaton is translated by setting the variable *state* to the initial location and applying π to the constraint formula describing the initial conditions of the hybrid automaton. This results in the initial state of the VSE-II temporal logic specification.

Additionally, π is divided into three cases:

ϕ: This formula represents the translation of the discrete actions taken by the hybrid automaton. The result of the translation is a disjunction with one disjunct for every edge in the hybrid automaton. These disjunctive parts are built from conjunctions representing the translation of locations, of the guard of this edge, and of the action the hybrid automaton would take in travelling along that edge. A further conjunct that acts as an enabling condition is added to this formula. It represents the translation of the invariant of location L_2 where (L_1, L_2) is the edge under consideration. The variables in this invariant are replaced by the values that the variables have after travelling along the edge.

ϕ^*: The formula ϕ^* is very similar to ϕ. It has an additional conjunct that describes the case where the invariant of the location would be violated if the system would make the next ψ step. As can be seen in the translation for ψ, all actions represented by ψ are disabled. In such a situation the system has to leave the location. This is ensured by requiring weak fairness of ϕ^*. Together with the enabling condition of ϕ^* it is assured that a ϕ^* step will be performed.

Removing ϕ^* from the translation and requiring weak fairness of ϕ would result in a behaviour violating the presented semantics of hybrid automata, since not all discrete actions that can be taken have to be performed in a fair manner.

ψ: Whereas ϕ and ϕ^* represent the translation of the discrete part of the hybrid automaton, ψ represents the translation of the continuous part. ψ is a disjunction of conjunctions. For every location in the hybrid automaton there is a disjunct consisting of

- a conjunction representing the translation of the location,
- the constraint term describing the change of the data variables in this location, and
- the translation of the location invariant.

Translation of Locations In the translation definition, the locations of the hybrid system are used as enabling conditions for the VSE-SL actions. They are used to indicate the change of a location.

Treatment of Fairness Since hybrid automata are assumed to behave fair in the sense that they do not stop[2], we have to add fairness constraints in a generic way. There are some special situations which have to be analysed.

The treatment of fairness is divided into two cases. First, all the translations of continuous actions from a hybrid system have to be fair. This is because time is assumed not to stop in a hybrid automaton. Every disjunctive part (action) of ψ is assumed to be executed in a fair manner. Therefore, we impose weak fairness on every disjunct of ψ.

The second case is concerned with discrete actions. In this situation, which is somehow more difficult, we can not impose fairness on every discrete action in the VSE-SL translation since not every discrete action in a hybrid system is executed fairly. A hybrid system which is in location L_i and has no location invariant can stay in this location forever executing the continuous actions without taking a discrete action even if some guard is forever satisfied. In the *Non_Leaking* location of the gasburner there is exactly this scenario. We are not allowed to require fairness of the discrete action in this situation. We are also not allowed to omit the fairness constraints on the discrete actions in general. To illustrate this, think of a hybrid automaton that is located in location L_i and has a location invariant $x \leq 1$ (see the leaking state in the gasburner example). The automaton has only one discrete action with source L_i with a true guard. When the variable x is about to violate the location invariant, the semantics of a hybrid automaton requires that the discrete action has to be executed. The same has to take place in the translated version and this can only be achieved by imposing fairness on the discrete action.

The solution we suggest[3] deals with the insertion of actions in the translation which are always executed in a fair manner, but which are only enabled if the action must take place. This is realized by the formula ϕ^* (see Definition 5.1.2). The structure of ϕ^* is close to that of ϕ with the exception that all disjunctive parts of ϕ^* have an additional enabling condition guaranteeing that these actions

[2]This holds for the non-zeno automata that are considered in this work.

[3]We know that there are other solutions to this problem. One that is also presented in this section is for example to collect all the actions of a system in one action and to require weak fairness for this very action.

are enabled if and only if a discrete action has to be taken. Because of fairness they will be executed.

The translation of fairness as described above assures that a discrete action must be taken if the local invariant requires it. Furthermore, if there is no invariant, then the system can stay in this location forever. This behaviour of the translated VSE-SL specification represents the behaviour of a hybrid automaton.

Thus, the translation of a hybrid automaton is represented by the following formula:

$$\pi(\mathcal{H}, init) \doteq \pi(init) \wedge \Box(\phi \vee \phi^* \vee \psi \vee stuttering) \wedge FAIR(\phi^*) \wedge FAIR(\psi)$$

where $FAIR(\ldots)$ stands for the application of weak fairness as defined in Section 4.1.3.

Before presenting an example translation of a hybrid automaton we give another possibility to handle fairness constraints. As explained before, we have to impose fairness restrictions on the actions resulting from the translation process. In the translation function π we have explicitly handled fairness by inserting the formula ϕ^* and impose weak fairness on it. Another, perhaps on the specification side more elegant possibility is the following. We redefine π as follows.

Definition 5.1.3 (Translation of initial states and automata)
Let init be a one element set $\{(f, L)\}$ representing the initial state of an automaton with $f \in CF$ and $L \in \mathcal{L}$, and let $\{x, x_1, \ldots, x_n\}$ be data variables and L_1, L_2 locations. Then we define π using the following formulae:

$$\pi(\{(f, L)\}) \;\doteq\; \pi(f) \wedge state = L$$

$$\begin{aligned}
\phi \;\equiv\; & \bigvee_{(L_1,L_2)\in\mathcal{E}} \Big(state = L_1 \wedge state' = L_2 \wedge \pi(guard((L_1, L_2)) \wedge \\
& \qquad \bigwedge_{x\in X} (x' = \pi(act((L_1, L_2), x))) \wedge (\pi(inv\,(L_2)))[x_1/x'_1, \ldots, x_n/x'_n] \Big) \\
\psi \;\equiv\; & \bigvee_{L\in\mathcal{L}} \Big(state = L \wedge state' = L \wedge \bigwedge_{x\in X} (x' = x + dif(L, x)) \wedge \\
& \qquad (\pi(inv\,(L)))[x_1/x_1 + dif(L, x_1), \ldots, x_n/x_n + dif(L, x_n)] \Big) \\
stuttering \;\equiv\; & \bigwedge_{x\in X} x' = x
\end{aligned}$$

The only difference to the Definition 5.1.2 is that the formula ϕ^* is removed and we impose weak fairness on the disjunction of ϕ and ψ. The result is shown in the following behaviour description:

$$\pi(\mathcal{H}, init) \doteq \pi(init) \wedge \Box(\phi \vee \psi \vee stuttering) \wedge FAIR(\phi \vee \psi)$$

Both translations[4] are behaviour equivalent, this can be seen by the following (informal) argument. The safety part of both translations is identical except the formula ϕ^*. It is clear that ϕ^* implies ϕ. Furthermore, we know that the only situation in which ϕ^* is enabled is reached, if and only if the next step of a ψ-action would violate the location invariant. In this situation the corresponding ϕ-action is enabled too. This results in the identical successor state compared to the ϕ^*-action. We further know that in such a situation all the ψ-actions are not enabled. With this we can see that ϕ^* does not contribute anything new to the safety part of the behaviour of the translated automaton.

Concerning the fairness part we examine the accepted behaviours of both variants of the translation. Both specifications, resulting from the different translations, require that if a ψ-action is enabled it has to be taken in a fair manner. In the π_2-translation this fairness is not as explicit as in the π_1-translation but has the same effect on the behaviour. Fairness of the disjunction of all ϕ- and ψ-actions is required in the result of the π_2-translation. In the following we investigate the situations in which fairness influences the behaviour of the translated specifications and compare the results of both translations.

- First, we reside within a location and only the continuous actions are enabled, i.e. all the ϕ or ϕ^* actions are disabled[5]. It is easy to see that both translations require that the next step to be taken is a ψ-step.

- Second, we are in a situation where the next ψ-step would violate the location invariant and there are some ϕ-steps enabled[6]. This situation is already analysed with respect to the translation π_1. It results in a specification where the next action that has to be taken is a ϕ^* action[7]. Considering the specification resulting from π_2 we know that all the ψ-actions are disabled in such a situation and we can reduce ψ to $false$ in the formula $FAIR(\phi \vee \psi)$ and get $FAIR(\phi)$. Because of the construction of the translations we can see that the resulting behaviours are identical.

- The third case does not deal with situations as described in the first two cases. It is more concerned with the fact that the translations should not impose too many restrictions via fairness on the behaviours. In this case at least two discrete actions are enabled simultaneously. As explained in the introduction of the translation π_1, the specification should not require that every ϕ-action is executed fairly. We have already looked at this problem during the introduction of ϕ^*. Also for π_2 the requirement is fulfilled since

[4] In the following we will use π_1 (see Definition 5.1.2) and π_2 (see Definition 5.1.3) to name the different translations. Where only π is mentioned the results apply to both.

[5] If actions are enabled, they can take place but they need not take place. This holds for both translations.

[6] If there are no further enabled ϕ-steps, then we have a deadlock and thus a zeno automaton.

[7] The difference between a ϕ^*- and a ϕ-action in this case is that a ϕ-action can take place but if no ϕ-action takes place, then a ϕ^*-action must take place because of the fairness constraints.

imposing fairness on the disjunction of all ψ- and ϕ-actions does not mean to impose fairness on every disjunctive part. In the result of the translation π_2, it is only required that one of the ϕ-actions has to be taken if no other action is enabled.

Thus, both translations result in behaviour equivalent specifications, i.e. they accept the same behaviours. Therefore, the advantages of one over the other can only lie in proof theoretic arguments. Taking this as a measure, it depends on the proof technique to be used during a proof. With the deduction unit of VSE-II as an exemplary proof environment we made the following observations. In proving safety properties the only difference between the results of both translations is that for π_1 there is an additional case to be handled namely the one that is concerned with ϕ^*. Because of the fact that this part of the proof can be handled identically to the corresponding ϕ case, proofs do not become more difficult. In VSE-II there is a reuse/replay technique implemented that can be used to do the part of the proof concerned with ϕ^* automatically. However, the fairness part changes. At least in VSE-II it seems to be more convenient to have explicit fairness constraints on the different actions constituting the behaviour of a system. The reason is, that looking at the fairness of all the actions of a translated system at once results in more complicated proofs than having to incorporate the fairness constraints of single actions into a proof. This argumentation cannot be generalised however for all VSE-II specifications containing fairness constraints.

Translation of PSL Formulae The translation function π distributes over boolean and temporal operators in PSL formulae. A small example illustrates this. The formulae $\Box(y \geq 60 \Rightarrow 20t < y)$ and $\Diamond(x \geq \frac{3}{2} \Rightarrow y \geq \frac{9}{10})$ are translated into $\Box(y \geq 60cn \Rightarrow 20t < y)$ and $\Diamond(2x \geq 3cn \Rightarrow 10y \geq 9cn)$ respectively. In this example we assumed a normalisation procedure applied to formulae of PSL.

Normalisation of a Hybrid Automaton We assume a certain kind of normalisation of hybrid automata that results in a hybrid automaton with no fractions either in the guard, the initial condition, nor in the discrete or continuous action definitions. A simple example will clarify the normalisation procedure.

Consider the hybrid automaton given in Figure 5.2. The normalised version of this automaton is given in Figure 5.3.

Within this normalisation process the initial conditions, the guards and the discrete changes are converted in a straightforward way by simply multiplying each constraint formula with the denominators occurring in this formula. The result[8] is a constraint formula with no denominators as shown in Figure 5.3.

[8]This is done for cosmetic reasons here. Within the translation of a hybrid automaton a granularity change is done that results in an automaton without fractions.

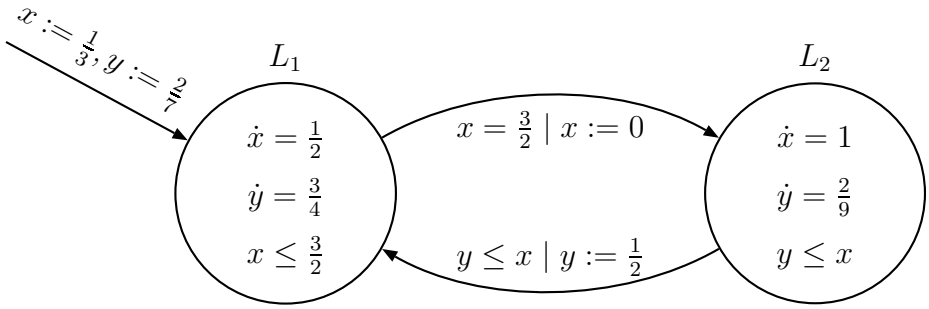

Figure 5.2: Non-Normalised Hybrid Automaton

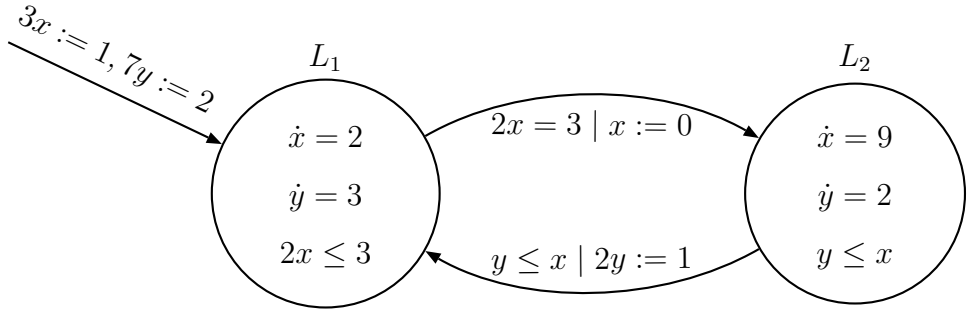

Figure 5.3: Normalised Hybrid Automaton

The continuous part is handled differently. Here it is important to take care of the rates of the variables within one location. This relation has to stay constant. We achieve this by computing the least common multiplier of the denominator of the slopes of all the variables in one location and multiply these slopes with it. In our example these were $\dot{y} = \frac{3}{4}$ and $\dot{x} = \frac{1}{2}$ in location L_1 and $\dot{y} = \frac{2}{9}$ in location L_2. For location L_1 the least common multiplier is 4 and for location L_2 it is 9. Performing the normalisation results in the automaton shown in Figure 5.3. The comparison of the relation of the rates of the variable x and y in location L_1 reveals that it is identical to this relation in the original non-normalised automaton.

Translation of an Example We are now ready to give the translation of a special hybrid automaton to demonstrate the translation.

EXAMPLE 5.1.2 *The hybrid automaton described here is an artificial example to illustrate the translation π. The hybrid automaton exhibits a faulty or at least undesired property of the specification; a zeno behaviour*[9]*. Without the zeno part the automaton fulfils simple properties as for example $\Box(y \leq x)$.*

The automaton is illustrated in Figure 5.4 and formally completely defined as follows:

[9]In this work we usually do not consider zeno automata, but even then the translation would work correct.

Let $X = \{x, y\}$, $L = \{L_1, L_2, L_3\}$, $E = \{(L_1, L_2), (L_2, L_1), (L_1, L_3), (L_3, L_1)\}$ *and init be defined as* $\{((x := 0, y := 0), L_1)\}$ *then dif, inv, guard and act are built as follows:*

$dif(L_1, x) = 1$ $dif(L_1, y) = 0$ $dif(L_2, x) = 0$
$dif(L_2, y) = 2$ $dif(L_3, x) = 1$ $dif(L_3, y) = 0$

$inv(L_1) = (x \leq 2)$ $inv(L_2) = (y \leq x)$ $inv(L_3) = (x \leq 1)$
$guard(L_1, L_2) = (x = 2)$ $guard(L_2, L_1) = (y \leq x)$ $guard(L_1, L_3) = (x \leq 1)$
$guard(L_3, L_1) = (x \geq 2)$

$act((L_1, L_2), x) = 1$ $act((L_1, L_2), y) = y$ $act((L_2, L_1), x) = x$
$act((L_2, L_1), y) = 0$ $act((L_1, L_3), x) = 0$ $act((L_1, L_3), y) = y$
$act((L_3, L_1), x) = 0$ $act((L_3, L_1), y) = y$

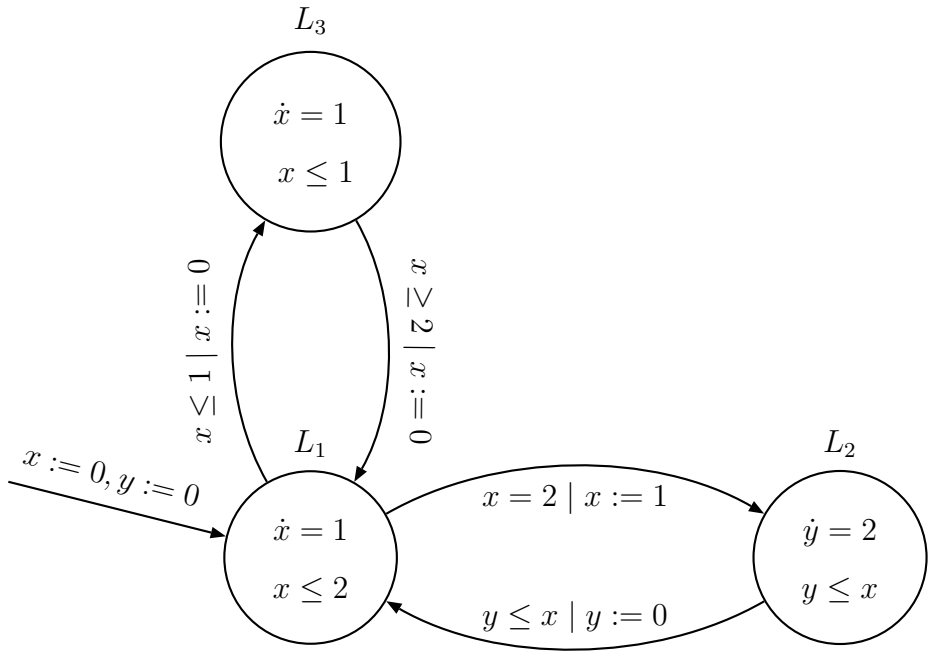

Figure 5.4: Graphical Representation: Hybrid Automaton

The π*-translation of this automaton is as follows:*
$\pi(init) \doteq x = 0 \wedge y = 0 \wedge state = L_1$
$stuttering \equiv x' = x \wedge y' = y$

$$
\begin{aligned}
\phi \equiv\ & (state = L_1 \wedge state' = L_2 \wedge x = 2 * c * n \wedge x' = c * n \wedge y' = y \wedge \\
& y' \leq x') && (\phi_1) \\
& \vee \\
& (state = L_2 \wedge state' = L_1 \wedge y \leq x \wedge x' = x \wedge y' = 0 \wedge x' \leq 2 * c * n) && (\phi_2) \\
& \vee \\
& (state = L_1 \wedge state' = L_3 \wedge x \leq c * n \wedge x' = 0 \wedge y' = y \wedge x' \leq c * n) && (\phi_3) \\
& \vee \\
& (state = L_3 \wedge state' = L_1 \wedge x \geq 2 * c * n \wedge x' = 0 \wedge y' = y \wedge \\
& x' \leq 2 * c * n) && (\phi_4)
\end{aligned}
$$

$$
\begin{aligned}
\psi \equiv\ & (state = L_1 \wedge state' = L_1 \wedge x' = x + 1 \wedge y' = y \wedge x + 1 \leq 2 * c * n) && (\psi_1) \\
& \vee \\
& (state = L_2 \wedge state' = L_2 \wedge x' = x \wedge y' = y + 2 \wedge y + 2 \leq x) && (\psi_2) \\
& \vee \\
& (state = L_3 \wedge state' = L_3 \wedge x' = x + 1 \wedge y' = y \wedge x + 1 \leq c * n) && (\psi_3)
\end{aligned}
$$

$$
\begin{aligned}
\phi^* \equiv\ & (state = L_1 \wedge state' = L_2 \wedge \neg(x + 1 \leq 2 * c * n) \wedge x = 2 * c * n \wedge \\
& x' = c * n \wedge y' = y \wedge y' \leq x') && (\phi_1^*) \\
& \vee \\
& (state = L_2 \wedge state' = L_1 \wedge \neg(y + 1 \leq x) \wedge y \leq x \wedge x' = x \wedge \\
& y' = 0 \wedge x' \leq 2 * c * n) && (\phi_2^*) \\
& \vee \\
& (state = L_1 \wedge state' = L_3 \wedge \neg(x + 1 \leq 2 * c * n) \wedge x \leq c * n \wedge \\
& x' = 0 \wedge y' = y \wedge x' \leq c * n) && (\phi_3^*) \\
& \vee \\
& (state = L_3 \wedge state' = L_1 \wedge \neg(x + 1 \leq c * n) \wedge x \geq 2 * c * n \wedge \\
& x' = 0 \wedge y' = y \wedge x' \leq 2 * c * n) && (\phi_4^*)
\end{aligned}
$$

The translation has the following form:

$$(x = 0 \wedge y = 0 \wedge state = L_1) \wedge$$
$$\Box(\phi_1 \vee \ldots \vee \phi_4 \vee \phi_1^* \vee \ldots \vee \phi_4^* \vee \psi_1 \vee \ldots \vee \psi_3 \vee stuttering) \wedge$$
$$\mathcal{WF}_{x,y}(\phi_1^*) \wedge \ldots \wedge \mathcal{WF}_{x,y}(\phi_4^*) \wedge \mathcal{WF}_{x,y}(\psi_1) \wedge \ldots \wedge \mathcal{WF}_{x,y}(\psi_3)$$

This form is equivalent to the normal form:

$$(x = 0 \wedge y = 0 \wedge state = L_1) \wedge$$
$$\Box[\phi_1, \ldots, \phi_4, \phi_1^*, \ldots, \phi_4^*, \psi_1, \ldots, \psi_3]_{x,y} \wedge$$
$$\mathcal{WF}_{x,y}(\phi_1^*) \wedge \ldots \wedge \mathcal{WF}_{x,y}(\phi_4^*) \wedge \mathcal{WF}_{x,y}(\psi_1) \wedge \ldots \wedge \mathcal{WF}_{x,y}(\psi_3)$$

Applying π_2 from Definition 5.1.3 to the hybrid automaton results in the formula:

$$(x = 0 \wedge y = 0 \wedge state = L_1) \wedge$$
$$\Box[\phi_1, \ldots, \phi_4, \psi_1, \ldots, \psi_3]_{x,y} \wedge$$
$$\mathcal{WF}_{x,y}(\phi_1 \vee \ldots \vee \phi_4 \vee \psi_1 \vee \ldots \vee \psi_3)$$

The formulae ϕ_3, ϕ_4 and ψ_3 show the "undesired behaviour" of the automaton. From the formal description and from Figure 5.4 we see that if we are in location L_3, we can never leave it because the location invariant and the guard on the edge (L_3, L_1) do not "fit". The same happens in the translated version of the automaton. Let us assume that we are located in state L_3, then the only action which leads to another state is ϕ_4. This action is never enabled since being in state L_3 means that x is always less or equal to $c * n$ and in the enabled condition of ϕ_4 x has to be greater or equal to $2 * c * n$[10]. In such a situation the translated automaton can only do stuttering steps forever.

The following situation is another source for errors or design flaws in the specification of an automaton. Imagine a variant of the described automaton where the action on the edge (L_3, L_1) is changed to $x = 1 \mid x := 3$. We have constructed a situation where performing the discrete action would result in a situation where the location invariant in L_1 would be immediately violated. The π-translation solves this problem like this. It adds an additional enabling condition to discrete actions ensuring that these actions are only enabled if they do not lead to an inconsistent state.

Other systems processing hybrid automata suggest different solutions to such a situation. In [79] $false$ would be the result in such a situation which indicates that the system is not consistent.

[10]Remember that c and n are both greater than zero.

There are also other possibilities for the translation π, for example an exception handling could be incorporated. This would be achieved by testing whether the location invariant and the action would lead to an inconsistent state. Only in cases where the location invariant is not violated the action can be taken. In the other cases the action would lead to an exception state which can not be left. In this way we can mark such an error in the translation of such an automaton.

It is clear that in both cases an indicator to show when the situation can appear is needed. This indicator is already built into the π-translation.

The translation of the changed automaton is different in the action definition of ϕ_4. It is changed to ϕ_4':

$$\phi_4' \equiv (state = L_3 \wedge state' = L_1 \wedge x = c * n \wedge x' = 3 * c * n \wedge y' = y \wedge x' \leq 2 * c * n)$$

Because of the fact that the formulae $x' = 3 * c * n$ and $x' \leq 2 * c * n$ cannot be satisfied in a behaviour ($c * n$ is assumed to be greater than zero), the action ϕ_4' is never enabled. This implies that the state L_3 can never be left in the VSE-SL specification.

5.2 Main Theorem

Up to now we defined the translation function π and what remains to be shown is that π behaves as desired, i.e. the diagram shown in Figure 5.1 commutes taking π as the indicated translation. Before stating the theorem we first give some preliminaries needed for the proof. We present the discretisation technique which is followed by some lemmata concerning the granularity change (see Definition 4.3.14) and the discretisation applied to hybrid automata.

5.2.1 Discretisation

The discretisation of a hybrid automaton $\mathcal{H}$ that is performed by Γ_δ results in a parameterised (δ) and discretised version of $\mathcal{H}$. Later on we shall see that the concatenation of the discretisation and the granularity change of a hybrid automaton is implemented by the π-translation.

DEFINITION 5.2.1 (DISCRETISATION)
Given a hybrid automaton $\mathcal{H}$ with $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, inv, guard, act)$ and $X = \{x_1, \ldots, x_n\}$ we define

$$\Gamma_\delta(\mathcal{H}) = (X, \mathcal{L}, \mathcal{E} \cup \mathcal{E}', dif', inv', guard \cup guard' \cup guard'', act \cup act')$$

where

- $\mathcal{E}' = \bigcup_{L \in \mathcal{L}} L \times L$,

- $dif'(L,x) = 0$ *for all* $L \in \mathcal{L}$, $x \in X$,
- $inv'(L) = \top$ *for all* $L \in \mathcal{L}$,
- $act'((L,L),x) = x + dif(L,x) * \delta$ *for all new* $(L,L) \in \mathcal{E}'$, $x \in X$ *and*
- $guard'((L,L)) = inv(L)[x_1/act'((L,L),x_1), \ldots, x_n/act'((L,L),x_n)]$ *for all new* $(L,L) \in \mathcal{E}'$
- $guard''((L_1,L_2)) = inv(L_2)[x_1/act((L_1,L_2),x_1), \ldots, x_n/act((L_1,L_2),x_n)]$ *for all* $(L_1,L_2) \in \mathcal{E}$

Thus, $\Gamma_\delta(\mathcal{H})$ *increases the data variables by multiples of* δ.

Note that $\Gamma_\delta(\mathcal{H})$ can hardly be called a hybrid automaton anymore. It is rather a standard (infinite state) automaton of the form $(X, \mathcal{L}, \mathcal{E} \cup \mathcal{E}', guard \cup guard' \cup guard'', act \cup act')$ that is parameterised with δ. An example for such a discretisation is the following.

Example 5.2.1 (Discretisation) *As a starting point we take the graphical representation of the gasburner presented in Figure 4.19. The discretisation computed by* Γ_δ *results in the automaton shown in Figure 5.5.*

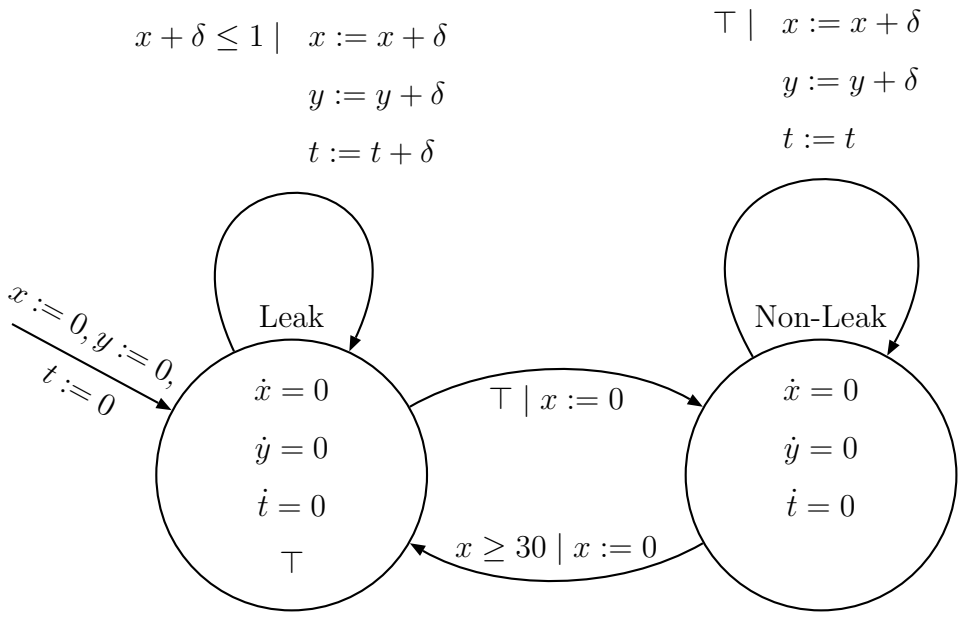

Figure 5.5: Leaking Gas Burner as (δ) Discretised (Hybrid) Automaton

As can be seen in Figure 5.5 all the continuous changes are set to 0 and only discrete steps remain. Within these steps the variables are incremented by the

rational number δ which is a free parameter. δ can be interpreted as any rational number, especially it can be arbitrarily small and Corollary 5.2.7 will show that it can be reduced to have the form $\frac{1}{m}$ with $m \in \mathbb{N}$.

Further important properties of a discretised automaton are given in the lemmata 5.2.2, 5.2.4, and 5.2.6.

LEMMA 5.2.2 *Let $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, \mathrm{inv}, guard, act)$ be a hybrid automaton with initial state $init_{\mathcal{H}} = (L, \phi)$. Then it holds:*

$$\sigma_1 \underset{\mathcal{H}}{\mapsto} \sigma_2 \text{ if and only if } \sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_2 \text{ for some } \delta \in \mathbb{Q}^+ \text{ with } \delta = \tfrac{n}{m},\ m, n \in \mathbb{N}.$$

Proof: "$\Leftarrow$": This direction follows obviously because we are considering only linear hybrid automata.
"$\Rightarrow$": The proof is by induction on i, where a position in a run ρ of $\mathcal{H}$ is given by $p = (i, r)$ with $i \in \mathbb{N}$, $r \in \mathbb{Q}$ according to Definition 4.3.11.

Base case 1: $p = (0, 0)$: Since the initial states of $\mathcal{H}$ and of $\Gamma_\delta(\mathcal{H})$ do not differ because it is not changed during the discretisation process, we are done with this case.

Base case 2: $p = (0, r)$ with $r \in \mathbb{Q}$: Informally this means that the hybrid automaton $\mathcal{H}$ is still in its initial location, but the automaton has made some time steps. This means that $\sigma_1 \overset{r}{\mapsto} \sigma_2$ with $\sigma_1 = (L, \phi)$, $\sigma_2 = (L, \phi')$, $f_x(0) = \phi$ and $f_x(r) = \phi'$ for all $x \in X$ where X is the set of variables occurring in $\mathcal{H}$.

From $r \in \mathbb{Q}$ it follows that $r = \frac{n}{m}$ with m, $n \in \mathbb{N}$. Choosing $\delta = \frac{n}{m}$, we can conclude as follows: Since $\sigma_1 \underset{\mathcal{H}}{\mapsto} \sigma_2$ the location invariant of location L is never violated during the time-transition steps done by $\mathcal{H}$ to reach state σ_2. During the discretisation process this location invariant is changed to a guard of the discretised automaton $\Gamma_\delta(\mathcal{H})$. So we can conclude that the corresponding discrete action in the discretised automaton is enabled in the initial state and can be executed. Thus, it follows that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\mapsto} \sigma_2$ where the term $x + dif(L, x) * \frac{n}{m}$ is identical to $f_x(r)$ for all $x \in X$.

Induction steps:

1. Timely reachable case: $p = (i, \delta_1)$:

   The last step done was a change of the data-variables in some location. We consider the run: $\sigma_1 \underset{\mathcal{H}}{\mapsto} \sigma_1' \overset{\delta_1}{\mapsto} \sigma_2$, where the position of σ_1' is $(i, 0)$ and the position of σ_2 is (i, δ_1).

   By the induction hypothesis there exists a $\delta \in \mathbb{Q}$ with $\delta = \frac{n}{m}$, n, $m \in \mathbb{N}$, such that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_1'$. Since $\delta_1 \in \mathbb{Q}$ it follows that $\delta_1 = \frac{n'}{m'}$ with n', $m' \in \mathbb{N}$. Choosing $\delta' = \frac{1}{m * m'}$ we can conclude as follows: We see that σ_1' is reachable from σ_1 in the automaton $\Gamma_{\delta'}(\mathcal{H})$, i.e. $\sigma_1 \underset{\Gamma_{\delta'}(\mathcal{H})}{\longmapsto} \sigma_1'$, since each run ρ with

$\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_1'$ can be extended to a run ρ' with $\sigma_1 \underset{\Gamma_{\delta'}(\mathcal{H})}{\longmapsto} \sigma_1'$ by replacing each new discrete step, the steps introduced by the discretisation of $\Gamma_\delta(\mathcal{H})$, by taking $(n * m')$ times the corresponding discrete steps of $\Gamma_{\delta'}(\mathcal{H})$. Thus, we divided one macro step of $\Gamma_\delta(\mathcal{H})$ into $n * m'$ micro steps of $\Gamma_{\delta'}(\mathcal{H})$. All of these micro steps can be extended. Otherwise, if one of the discrete steps of $\Gamma_{\delta'}(\mathcal{H})$ cannot be executed, then with Lemma 5.2.3 this must have been the case for $\Gamma_\delta(\mathcal{H})$ which is a contradiction to the precondition $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_1'$.

Extending $\sigma_1 \underset{\Gamma_{\delta'}(\mathcal{H})}{\longmapsto} \sigma_1'$ by $\sigma_1' \overset{\text{tr}}{\mapsto} \sigma_{11}' \overset{\text{tr}}{\mapsto} \ldots \overset{\text{tr}}{\mapsto} \sigma_{1(n'*m)}' = \sigma_2$, i.e. we extend the run by $n' * m$ discrete steps where in each of these steps the change of the variables is computed by $x + dif(L, x) * \delta' = x + dif(L, x) * \frac{1}{m*m'}$ for all $x \in X$. Executing this discrete action $n' * m$ times the incrementation of the variable x is given by

$$x + dif(L, x) * \frac{n' * m}{m' * m} = x + dif(L, x) * \frac{n'}{m'} = x + dif(L, x) * \delta_1 = f_x(\delta_1)$$

for all $x \in X$. Again all of these steps can be executed, since otherwise there must have been a state in the run of the original automaton $\mathcal{H}$ such that σ_2 is not timely reachable from σ_1' which is a contradiction to the precondition. From this it follows that the constructed run is a run of $\Gamma_{\delta'}(\mathcal{H})$ and that $\sigma_1 \underset{\Gamma_{\delta'}(\mathcal{H})}{\longmapsto} \sigma_1' \underset{\Gamma_{\delta'}(\mathcal{H})}{\longmapsto} \sigma_2$ holds. This concludes this case.

2. Transition reachable case: $p = (i + 1, 0)$:

   We are in a situation where the last step done by the automaton $\mathcal{H}$ was a discrete one, i.e. $\sigma_1 \underset{\mathcal{H}}{\mapsto} \sigma_1' \overset{\text{tr}}{\mapsto} \sigma_2$ where the position of σ_1' is (i, r) with $r \in \mathbb{Q}$ and the position of σ_2 is $(i + 1, 0)$.

   By the induction hypothesis there exists a $\delta = \frac{n}{m}$ with n, $m \in \mathbb{N}$ such that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_1'$. Because of the fact that we are considering only admissible states in the run of $\mathcal{H}$ and since the discrete actions of $\Gamma_\delta(\mathcal{H})$ differ from the discrete actions of $\mathcal{H}$ only with respect to an additional constraint in the guard of the discrete action that expresses exactly the admissibility of the next state to be reached executing this discrete step, we can conclude that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_1'$ and thus that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_2$.

This concludes the proof. □

The following lemma makes use of the linearity of the constraints used in linear hybrid automata. It states that a constraint formula that holds at some point in time and that still holds δ time units later must be true at every time instant inbetween.

LEMMA 5.2.3 *Let C be a constraint formula and let x_i, $1 \leq i \leq n$, be the variables occurring in C. If C and $C[x_i/x_i + k_i * \delta]$ hold then it follows that $C[x_i/x_i + k_i * \delta']$ holds with $\delta' \in \{0 \leq \delta' \leq \delta\}$.*

Proof: The proof is by structural induction on constraint formulae.
Base case 1: $C \mathrel{\widehat{=}} a_1x_1 + \ldots + a_nx_n = b$ with a, $b \in RC$.
From the precondition we know that (1) $a_1x_1 + \ldots + a_nx_n = b$ and that (2) $a_1(x_1 + k_1\delta) + \ldots a_n(x_n + k_n\delta) = b$. Using (1) and (2) we can conclude the following:

$$\begin{aligned} a_1(x_1 + k_1\delta) + \ldots + a_n(x_n + k_n\delta) &= b \\ a_1x_1 + \ldots + a_nx_n + a_1k_1\delta + \ldots + a_nk_n\delta &= b \\ b + a_1k_1\delta + \ldots + a_nk_n\delta &= b \\ a_1k_1\delta + \ldots + a_nk_n\delta &= 0 \\ \delta(a_1k_1 + \ldots + a_nk_n) &= 0 \end{aligned}$$

From this we can infer that $\delta = 0$ or that $(a_1k_1 + \ldots + a_nk_n) = 0$. Thus, we can conclude as follows:

$$\begin{aligned} a_1(x_1 + k_1\delta') + \ldots a_n(x_n + k_n\delta') &= \\ a_1x_1 + \ldots + a_nx_n + a_1k_1\delta' + \ldots + a_nk_n\delta' &= \\ b + a_1k_1\delta' + \ldots + a_nk_n\delta' &= \\ b + \delta'(a_1k_1 + \ldots + a_nk_n) & \qquad (5.1) \end{aligned}$$

We know that $\delta = 0$ or that $(a_1k_1 + \ldots + a_nk_n) = 0$. The result of 5.1 is in both cases b and we are done.
Base case 2: $C \mathrel{\widehat{=}} a_1x_1 + \ldots + a_nx_n < b$ with a, $b \in RC$.
From the preconditions we can infer the following:

$$\begin{aligned} a_1(x_1 + k_1\delta) + \ldots + a_n(x_n + k_n\delta) &< b \\ a_1x_1 + \ldots + a_nx_n + a_1k_1\delta + \ldots + a_nk_n\delta &< b \\ \delta(a_1k_1 + \ldots + a_nk_n) &< b - (a_1x_1 + \ldots + a_nx_n) \qquad (5.2) \end{aligned}$$

Now we have to prove that

$$a_1x_1 + \ldots + a_nx_n + \delta'(a_1k_1 + \ldots + a_nk_n) < b \qquad (5.3)$$

For the sequel we assume that $\delta \neq 0$ since otherwise we can conclude that $\delta' = 0$ and from the precondition we know that $a_1x_1 + \ldots + a_nx_n < b$ holds. Thus, we can conclude that 5.3 holds and we are done in this case.

Now we consider three cases for the term $a_1k_1 + \ldots + a_nk_n$:
Case $a_1k_1 + \ldots + a_nk_n = 0$: In this case we can immediately conclude that 5.3 is less than b because of the precondition that $a_1x_1 + \ldots + a_nx_n < b$ holds.
Case $a_1k_1 + \ldots + a_nk_n < 0$: It is evident that 5.3 is less than b under this condition.

Case $a_1k_1 + \ldots + a_nk_n > 0$: With 5.2 we get the following:

$$\begin{aligned} \delta'(a_1k_1 + \ldots + a_nk_n) &\leq \delta(a_1k_1 + \ldots + a_nk_n) \\ &< b - (a_1x_1 + \ldots a_nx_n) \end{aligned} \tag{5.4}$$

With 5.4 we can infer:

$$\begin{aligned} & a_1x_1 + \ldots + a_nx_n + \delta'(a_1k_1 + \ldots a_nk_n) \\ < \; & a_1x_1 + \ldots + a_nx_n + b - (a_1x_1 + \ldots + a_nx_n) \\ = \; & b \end{aligned}$$

Thus, it follows that (5.3) holds and we are done.

Induction step: $C \mathrel{\widehat{=}} C_1 \wedge C_2$
We know that

$$C_1 \wedge C_2 \wedge (C_1 \wedge C_2)[x_i/x_i + k_i\delta]) \tag{5.5}$$

holds. From 5.5 we can conclude that

$$C_1 \wedge C_1[x_i/x_i + k_i\delta] \wedge C_2 \wedge C_2[x_i/x_i + k_i\delta]) \tag{5.6}$$

holds. By the induction hypothesis we get the following

$$C_1[x_i/x_i + k_i\delta'] \wedge C_2[x_i/x_i + k_i\delta']) \text{ for } \delta' \in \{0 \leq \delta' \leq \delta\} \tag{5.7}$$

From 5.7 we know that

$$(C_1 \wedge C_2)[x_i/x_i + k_i\delta'] \text{ for } \delta' \in \{0 \leq \delta' \leq \delta\} \tag{5.8}$$

follows and this concludes the proof. □

Lemma 5.2.4 *Let $\mathcal{H}$ be a hybrid automaton, then for all $\delta \in \mathbb{Q}^+$:*

$$\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_2 \text{ if and only if } \forall n \in \mathbb{N} : \; \sigma_1 \underset{\Gamma_{\frac{\delta}{n}}(\mathcal{H})}{\longmapsto} \sigma_2$$

Proof: "⇐": Choosing n to be 1 we are done.
"⇒":
We are considering only admissible states in a run of $\mathcal{H}$. Therefore, we can confine ourselves to consider only the new discrete actions of $\Gamma_\delta(\mathcal{H})$ and $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ inserted within the discretisation process.

The proof is by induction on the position $p = (i, r)$ in the run ρ of $\Gamma_\delta(\mathcal{H})$ with $\text{start}(\rho) = \sigma_1$. Because of the discretisation of $\Gamma_\delta(\mathcal{H})$ the positions p in the run ρ are of the form $p = (i, 0)$.

Base case: $p = (0, 0)$:
Since the initial states of $\Gamma_\delta(\mathcal{H})$ and $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ are not changed by the discretisation process, we are done with the base case.

Induction step: $p = (n, 0)$:
We assume that $\sigma_1 \underset{\Gamma_\delta(\mathcal{H})}{\longmapsto} \sigma_2$ and that $\rho((n+1, 0))$, which means that the position of σ_2 in the run ρ is $(n+1, 0)$. Let σ_2' be the state of the run ρ of $\Gamma_\delta(\mathcal{H})$ with position $(n, 0)$. By the induction hypothesis we know that $\sigma_1 \underset{\Gamma_{\frac{\delta}{n}}(\mathcal{H})}{\longmapsto} \sigma_2'$ holds. As argued before we have to consider only the new discrete actions of $\Gamma_\delta(\mathcal{H})$ ($\Gamma_{\frac{\delta}{n}}(\mathcal{H})$) respectively, since in case that σ_2 is reached from σ_2' by an originally discrete step of $\mathcal{H}$, then this step is the same for $\Gamma_\delta(\mathcal{H})$ and $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ because the discretisation does not change these steps and we are done.

Let C be the guard of the discrete action of $\Gamma_\delta(\mathcal{H})$ taken in state σ_2' that leads to state σ_2: Then we know that $C[x_i/x_i + k_i * \delta]$ holds in state σ_2' where x_i are the variables occurring in the discrete action and the k_i represent the corresponding growth rates.

By Lemma 5.2.3 we can conclude that $C[x_i/x_i + k_i * \delta']$ with $\delta' \in \{0 \leq \delta' \leq \delta\}$ holds in state σ_2'. From this we can deduce that $\forall n \in \mathbb{N} : C[x_i/x_i + k_i * \frac{\delta}{n}]$ holds. It follows that the corresponding guard in the automaton $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ is true throughout the n-times execution of the discrete action. Thus, we have that $\sigma_2' \underset{\Gamma_{\frac{\delta}{n}}(\mathcal{H})}{\longmapsto} \sigma_2$ and together with the induction hypothesis it follows that $\sigma_1 \underset{\Gamma_{\frac{\delta}{n}}(\mathcal{H})}{\longmapsto} \sigma_2$ and we are done.
□

LEMMA 5.2.5 *For any atemporal PSL-formula* Φ *we have*

$$\mathcal{H}, \sigma \models \Box\Phi \quad \textit{if and only if} \quad \forall \delta \in \mathbb{Q}^+ \; \Gamma_\delta(\mathcal{H}), \sigma \models \Box\Phi$$

Proof: "⇒": obvious
"⇐":
Assume that $\mathcal{H}, \sigma \models \Box\Phi$ does not hold. Then we can conclude as follows:
$\neg(\mathcal{H}, \sigma \models \Box\Phi)$ if and only if
$\neg(\mathcal{H}, \rho \models \Box\Phi$ for all $\rho \in \text{runs}(\mathcal{H}, \sigma))$ if and only if
$\neg(\mathcal{H}, \bar{\rho} \models \Phi$ for all $\bar{\rho}$ with $\text{suf}(\bar{\rho}, \rho)$ for all $\rho \in \text{runs}(\mathcal{H}, \sigma))$ if and only if
$\mathcal{H}, \bar{\rho} \models \neg\Phi$ for some $\bar{\rho}$ with $\text{suf}(\bar{\rho}, \rho)$ for some $\rho \in \text{runs}(\mathcal{H}, \sigma)$
Let $\tau = \text{start}(\bar{\rho})$ and $\sigma \underset{\mathcal{H}}{\mapsto} \tau$ with Φ does not hold in τ. With Lemma 5.2.2 we have:
$\sigma \underset{\Gamma_\delta(\mathcal{H})}{\mapsto} \tau$ for some $\delta \in \mathbb{Q}^+$ and Φ does not hold in τ. It follows that there exists a $\delta \in \mathbb{Q}^+$, such that $\Gamma_\delta(\mathcal{H}), \sigma \not\models \Box\Phi$ which is a contradiction to the precondition. □

The following theorem, which expresses that a hybrid automaton and the Γ_δ-transformed version of it exhibit the same PSL-properties, is one of the most important theorems needed in Theorem 5.2.18, the main theorem.

Theorem 5.2.6 *Let Φ be an atemporal PSL-formula, i.e. Φ contains no temporal operator. Then*

$$\mathcal{H}, \sigma \models \Box\Phi \quad \textit{if and only if} \quad \forall \delta \in \mathbb{Q}^+ \; \exists n \in \mathbb{N} : \; \Gamma_{\delta/n}(\mathcal{H}), \sigma \models \Box\Phi$$
$$\mathcal{H}, \sigma \models \Diamond\Phi \quad \textit{if and only if} \quad \exists \delta \in \mathbb{Q}^+ \; \forall n \in \mathbb{N} : \; \Gamma_{\delta/n}(\mathcal{H}), \sigma \models \Diamond\Phi$$

Proof: Proof of the first case:
"$\Rightarrow$":
From $\mathcal{H}, \sigma \models \Box\Phi$ it follows that $\forall \delta \in \mathbb{Q}^+ : \Gamma_\delta(\mathcal{H}), \sigma \models \Box\Phi$ holds by Lemma 5.2.5. Choosing n to be 1 we are done.
"$\Leftarrow$":
Assume that $\mathcal{H}, \sigma \not\models \Box\Phi$. With Lemma 5.2.5 we can conclude that $\exists \delta \in \mathbb{Q}^+ : \Gamma_\delta(\mathcal{H}), \sigma \not\models \Box\Phi$ holds. From this we know that on every run ρ of $\Gamma_\delta(\mathcal{H})$ with start state σ there is a state where ϕ does not hold. From Lemma 5.2.4 it follows that this state is also reachable for $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ for all $n \in \mathbb{N}$. From this we have that $\exists \delta \in \mathbb{Q}^+ : \forall n \in \mathbb{N} : \Gamma_{\frac{\delta}{n}}(\mathcal{H}), \sigma \models \Diamond\neg\Phi$ that is a contradiction to the precondition.

Proof of the second case:
Let us first consider the direction from left to right, i.e. suppose $\mathcal{H}, \sigma \models \Diamond\Phi$ holds. This is equivalent according to Definition 4.3.16 to $\forall \rho \in \text{runs}(\mathcal{H}, \sigma) : \; \mathcal{H}, \rho \models \Diamond\Phi$. Now assume that $\forall \delta \; \exists n \; \exists \rho \in \text{runs}(\Gamma_{\delta/n}(\mathcal{H}), \sigma) : \; \rho \models \Box\neg\Phi$, i.e. for each $\delta \in \mathbb{Q}^+$ there exists a run of the form

$$\sigma = \sigma_0 \mapsto \sigma_1 \mapsto \sigma_2 \mapsto \sigma_3 \mapsto$$

where we assume that it takes k_i δ/n-steps to go from σ_i to σ_{i+1}. Evidently, each such run can easily be extended to a (prefix of a) run for $\mathcal{H}$[11] and this can only prevent an immediate contradiction if all these runs would stop before they reach the Φ-moment. However, in this case we can try another (smaller) fraction of δ to proceed for the discretised automaton. If it really were the case that for no fraction of δ there is a further step, then the original hybrid automaton already would have no possibility to proceed as well, which contradicts our preliminaries. Therefore, the assumption cannot be true and we are done with this direction.

For the direction from right to left assume that $\exists \rho \in \text{runs}(\mathcal{H}, \sigma) : \; \rho \models \Box\neg\Phi$. Then, given δ, we can find for each finite prefix of ρ a suitable n such that σ_{i+1} is reachable from σ_i in k_i δ/n steps. It would thus be possible to construct a discrete run ρ' such that $\rho' \models \Box\neg\Phi$ which contradicts the precondition. $\Box$

[11] Recall that we are dealing with linear constraints.

COROLLARY 5.2.7 *The above lemmata and theorems hold also in case we consider special δ's, namely those of the form $1/m$ where m is a positive integer.*

Proof: This is due to the fact that for any δ of the form l/m we can equivalently consider a $\delta' = 1/m$ but repeat the steps l times. With Lemma 5.2.4 it follows that every state reachable before with δ would also be reachable with δ' used in the discretisation process. □

At first glance it might be surprising that Theorem 5.2.6 works only in case there are no nested temporal operators. In fact, the theorem would not hold otherwise, at least in general. If, however, one insists on nested temporal operators – for example to be able to express certain types of liveness properties – then some additional machinery is necessary. We do not go into details here and refer the interested reader to Section 5.3.

5.2.2 Granularity Change

The formal definition of a granularity change is given in Section 4.3.2 in Definition 4.3.14. We motivated the introduction of the granularity change by the need for switching the base units in some cases. In this section we present the granularity changed version (see Figure 5.6) of the (δ) discretised gasburner (see Figure 5.5) and a theorem which gives the correlation between a hybrid automaton $\mathcal{H}$ and its granularity changed version with respect to a PSL formula P. To this end, we make use of the lemmas 5.2.9 and 5.2.10 given below.

EXAMPLE 5.2.8 *As can be seen in Figure 5.6 the change of the granularity of a discretised hybrid automaton results in an automaton where*

- *variables are only changed in discrete actions,*
- *the change of the variables is always a natural number, and*
- *all sole occurrences of rational numbers are replaced by some constant $c * n$ or multiples of it.*

This translation from a hybrid automaton to a VSE-II specification is rather simple and all safety properties of the original automaton hold also for the changed automaton.

Lemma 5.2.9 expresses that the set of suffixes of a run ρ with changed granularity is equal to the set of granularity changed suffixes of ρ.

LEMMA 5.2.9 *For any given run ρ:* $\{\rho_1 \mid \mathrm{suf}(\rho_1, \Delta \star \rho)\} = \{\Delta \star \rho_2 \mid \mathrm{suf}(\rho_2, \rho)\}$

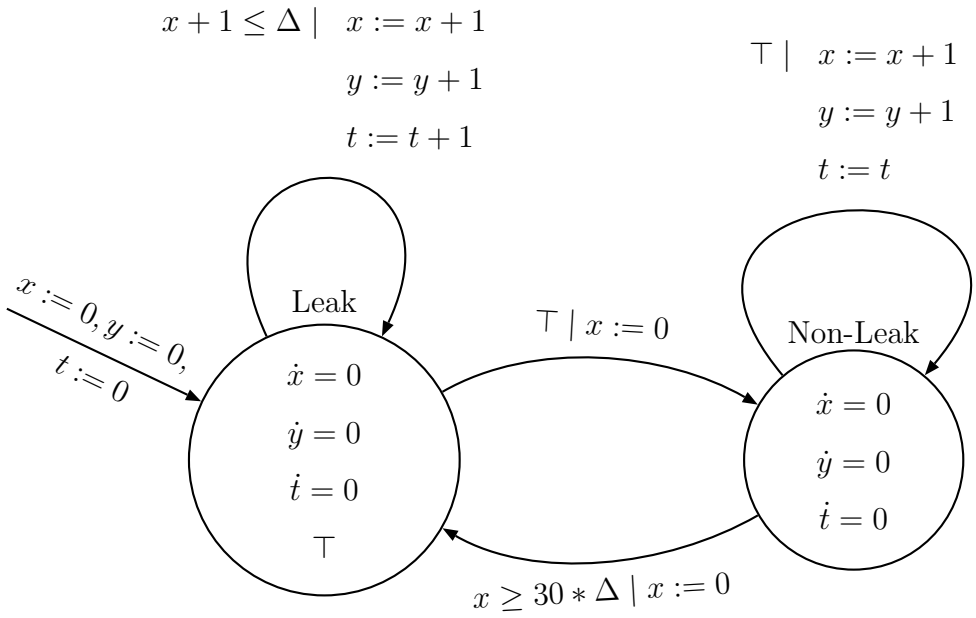

Figure 5.6: Discretised and Granularity Changed Leaking Gas Burner

Proof: Let $\rho = (l_0, \phi_0) \mapsto_{f_0}^{t_0} (l_1, \phi_1) \mapsto_{f_1}^{t_1} (l_2, \phi_2) \cdots$ and fix any $\Delta \in \mathbb{N}(\mathbb{Q}^+)$.
$\Rightarrow$: For some position (i, q) we then get

$$\rho_1 = (l_i, (\Delta \star f_i)(q)) \mapsto_{(\Delta \star f_i)'}^{\Delta * t_i - q} (l_{i+1}, \phi_{i+1}) \mapsto_{\Delta \star f_{i+1}}^{\Delta * t_{i+1}} \cdots$$

where $0 \leq q \leq \Delta * t_i$ and

$$(\Delta \star f_i)'(t) = (\Delta \star f_i)(q + t) = \Delta \star f_i(\frac{q+t}{\Delta}).$$

To show that $\rho_1 \in \{\Delta \star \rho_2 \mid \mathrm{suf}(\rho_2, \rho)\}$. To this end, let ρ_2 be the suffix of ρ that starts at position $(i, \frac{q}{\Delta})$, i.e.,

$$\rho_2 = (l_i, f_i(\frac{q}{\Delta})) \mapsto_{f_i'}^{t_i - \frac{q}{\Delta}} (l_{i+1}, \phi_{i+1}) \mapsto_{f_{i+1}}^{t_{i+1}} \cdots$$

where $f_i'(t) = f_i(\frac{q}{\Delta} + t)$. Then

$$\Delta \star \rho_2 = (l_i, \Delta \star f_i(\frac{q}{\Delta})) \mapsto_{\Delta \star f_i'}^{\Delta * t_i - q} (l_{i+1}, \Delta \star \phi_{i+1}) \mapsto_{\Delta \star f_{i+1}}^{\Delta * t_{i+1}} \cdots$$

Note that

$$(\Delta \star f_i')(t) = \Delta \star f_i'(\frac{t}{\Delta}) = \Delta \star f_i(\frac{q}{\Delta} + \frac{t}{\Delta}) = \Delta \star f_i(\frac{q+t}{\Delta}) = (\Delta \star f_i)'(t).$$

Therefore

$$\Delta \star \rho_2 = (l_i, (\Delta \star f_i)(q)) \mapsto_{(\Delta \star f_i)'}^{\Delta * t_i - q} (l_{i+1}, \phi_{i+1}) \mapsto_{\Delta \star f_{i+1}}^{\Delta * t_{i+1}} \cdots = \rho_1$$

and we are done.
$\Leftarrow$: This is tantamount to showing that if $\mathrm{suf}(\rho_2, \rho)$ holds, then $\mathrm{suf}(\Delta \star \rho_2, \Delta \star \rho)$ holds too. Therefore, take any suffix ρ_2 of ρ, say at position (i, q), and consider $\Delta \star \rho_2$, i.e.

$$\Delta \star \rho_2 = (l_i, \Delta \star f_i(q)) \mapsto^{\Delta * (t_i - q)}_{\Delta \star f_i'} (l_{i+1}, \Delta \star \phi_{i+1}) \mapsto^{\Delta * t_{i+1}}_{\Delta \star f_{i+1}} \cdots$$

where $f_i'(t) = f_i(q + t)$. For $\Delta \star \rho$ we know that

$$\Delta \star \rho = (l_0, \Delta \star \phi_0) \mapsto^{\Delta * t_0}_{\Delta \star f_0} (l_1, \Delta \star \phi_1) \mapsto^{\Delta * t_1}_{\Delta \star f_1} \cdots .$$

Now consider the suffix of $\Delta \star \rho$ starting at position $(i, \Delta * q)$, i.e. we consider the run

$$(l_i, (\Delta \star f_i)(\Delta * q)) \mapsto^{\Delta * t_i - \Delta * q}_{(\Delta \star f_i)'} (l_{i+1}, \Delta \star \phi_{i+1}) \mapsto^{\Delta * t_{i+1}}_{\Delta \star f_{i+1}} \cdots .$$

As mentioned above, we know that $(\Delta \star f_i')(t) = (\Delta \star f_i)'(t)$ and, moreover (by Definition 4.3.14), that $(\Delta \star f_i)(\Delta * q) = \Delta \star f_i(q)$. Hence, the above suffix is exactly $\Delta \star \rho_2$ and we are done. □

LEMMA 5.2.10 *For any valuation ϕ and term $t \in \mathcal{CT}$: $(\Delta \star \phi)(\Delta \star t) = \Delta * \phi(t)$.*

Proof: The proof is by induction on the structure of the constraint term t.
Base case 1: Let $t \widehat{=} x$ with $x \in X$, then with Definition 4.3.14 we have
$(\Delta \star \phi)(\Delta \star t) = (\Delta \star \phi)(\Delta \star x) = (\Delta \star \phi)(x) = \Delta * \phi(x)$
Base case 2: Let $t \widehat{=} d$ with $d \in RC$, then with Definition 4.3.4 we have
$(\Delta \star \phi)(\Delta \star d) = (\Delta \star \phi)(\Delta * d) = \Delta * d = \Delta * \phi(d)$
Induction step:
Let $t = d * x$ with $d \in RC$ and $x \in X$, then we have

$(\Delta \star \phi)(\Delta \star t) = (\Delta \star \phi)(\Delta \star (d * x))$	=	[by Definition 4.3.14]
$(\Delta \star \phi)(d * x)$	=	[by Definition 4.3.4]
$d * (\Delta \star \phi)(x)$	=	[by Definition 4.3.14]
$d * (\Delta \star \phi)(\Delta \star x)$	=	[by ind. hypothesis]
$d * \Delta * \phi(x)$	=	[by Definition 4.3.4]
$\Delta * \phi(d * x)$	=	
$\Delta * \phi(t)$		

Let $t = t_1 \circ t_2$ with $\circ \in \{+, -\}$, then

$$
\begin{array}{lll}
(\Delta \star \phi)(\Delta \star t) & = & \\
(\Delta \star \phi)(\Delta \star (t_1 \circ t_2)) & = & \text{[by Definition 4.3.14]} \\
(\Delta \star \phi)(\Delta \star t_1) \circ (\Delta \star \phi)(\Delta \star t_2) & = & \text{[by ind. hypothesis]} \\
\Delta * \phi(t_1) \circ \Delta * \phi(t_2) & = & \\
\Delta * (\phi(t_1) \circ \phi(t_2)) & = & \\
\Delta * \phi(t_1 \circ t_2) = \Delta * \phi(t) & &
\end{array}
$$

which concludes the proof. □

Together with Lemma 5.2.10 and Lemma 5.2.9 we are in a position to prove Theorem 5.2.11 which specifies the relation of a hybrid automaton and its granularity changed version with regard to a property of the hybrid automaton.

Theorem 5.2.11 *Let $\mathcal{H}$ be a hybrid automaton, $\mathcal{F}$ a PSL formula then*

$$\mathcal{H}, \rho \models \mathcal{F} \text{ if and only if } \Delta \star \mathcal{H}, \Delta \star \rho \models \Delta \star \mathcal{F}$$

Proof: The proof is by induction on the structure of $\mathcal{F}$. We consider the cases $\mathcal{F} = \Box\mathcal{F}_1$, $\mathcal{F} = t_1 \circ t_2$ and $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$ with $t_1, t_2 \in \mathcal{CT}$. The remaining cases can be proved analogously.

$$
\begin{array}{lll}
(\mathcal{H}, \rho) \models \Box\mathcal{F}_1 & \text{iff} & (\mathcal{H}, \rho') \models \mathcal{F}_1 \text{ for all } \rho' \text{ with } \mathrm{suf}(\rho', \rho) \text{ [by Definition 4.3.16]} \\
& \text{iff} & (\Delta \star \mathcal{H}, \Delta \star \rho') \models \Delta \star \mathcal{F}_1 \text{ for all } \rho' \text{ with } \mathrm{suf}(\rho', \rho) \text{ [ind. hyp.]} \\
& \text{iff} & (\Delta \star \mathcal{H}, \rho') \models \Delta \star \mathcal{F}_1 \text{ for all } \rho' \text{ with } \mathrm{suf}(\rho', \Delta \star \rho) \text{ [L.5.2.9]} \\
& \text{iff} & (\Delta \star \mathcal{H}, \Delta \star \rho) \models \Box(\Delta \star \mathcal{F}_1) \text{ [by Definition 4.3.16]} \\
& \text{iff} & (\Delta \star \mathcal{H}, \Delta \star \rho) \models \Delta \star (\Box\mathcal{F}_1) \text{ [by Definition 4.3.14]}
\end{array}
$$

$$
\begin{aligned}
(\mathcal{H},\rho) \models \mathcal{F}_1 \wedge \mathcal{F}_2 \quad & \text{iff} \quad (\mathcal{H},\rho) \models \mathcal{F}_1 \text{ and } (\mathcal{H},\rho) \models \mathcal{F}_2 \text{ [by Definition 4.3.16]} \\
& \text{iff} \quad (\Delta \star \mathcal{H}, \Delta \star \rho) \models \Delta \star \mathcal{F}_1 \text{ and} \\
& \qquad (\Delta \star \mathcal{H}, \Delta \star \rho) \models \Delta \star \mathcal{F}_2 \text{ [by ind. hyp.]} \\
& \text{iff} \quad (\Delta \star \mathcal{H}, \rho) \models \Delta \star \mathcal{F}_1 \wedge \Delta \star \mathcal{F}_2 \text{ [by Definition 4.3.16]} \\
& \text{iff} \quad (\Delta \star \mathcal{H}, \rho) \models \Delta \star (\mathcal{F}_1 \wedge \mathcal{F}_2) \text{ [by Definition 4.3.14]}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{H},\rho) \models t_1 \circ t_2 \quad & \text{iff} \quad \phi(t_1) \circ \phi(t_2) \text{ where } \mathrm{start}(\rho) = (L,\phi) \text{ [by Def. 4.3.16]} \\
& \text{iff} \quad \Delta * \phi(t_1) \circ \Delta * \phi(t_2) \text{ [by arithmetic]} \\
& \text{iff} \quad (\Delta \star \phi)(\Delta \star t_1) \circ (\Delta \star \phi)(\Delta \star t_2) \text{ [by Lemma 5.2.10]} \\
& \text{iff} \quad \phi'(\Delta \star t_1) \circ \phi'(\Delta \star t_2) \text{ with } \phi' = \Delta \star \phi \\
& \text{iff} \quad (\Delta \star \mathcal{H}, \Delta \star \rho) \models (\Delta \star t_1) \circ (\Delta \star t_2) \\
& \qquad \text{with } \mathrm{start}(\Delta \star \rho) = (L,\phi') \text{ and } \phi' = \Delta \star \phi \\
& \text{iff} \quad (\Delta \star \mathcal{H}, \Delta \star \rho) \models \Delta \star (t_1 \circ t_2) \text{ [by Definition 4.3.14]}
\end{aligned}
$$

This completes the proof. □

Before proving the main theorem we define a relation between states of a hybrid automaton and states of a temporal logic specification[12].

DEFINITION 5.2.2 ($\tilde{=}$)
A state $\sigma_j = (L_j, \phi_j)$ of a run σ and a state t_i of a behaviour t are in the relation $\tilde{=}$, i.e. $\sigma_j \tilde{=} t_i$ if and only if $t_i(state) = L_j$ and $t_i(x) = \phi_j(x)$ for all $x \in X$, where X is the set of variables occurring in σ respectively t and "state" is a special variable according to Definition 5.1.2.

In the following lemma we use an assumption about the growth rate of the data-variables in linear hybrid automata. Such a growth rate is normally given by a rational number, say $q \in \mathbb{Q}$. We can assume without loss of generality that $q \in \mathbb{N}$ holds, since only the relation of the growth rates of the variables within one location is of interest. The growth rates q_i, with $q_i \in \mathbb{Q}$ within one location can be normalised so that all q_i are natural numbers by simply multiplying all $q_i = \frac{n_i}{m_i}$ with the least common multiplier of all the m_i as described in Section 5.1.1 in the normalisation of a hybrid automaton.

[12]Although the notion of a state is different in the context of hybrid automata and temporal logic specifications, we have overloaded it. However, from the context it should be clear which one is meant.

In the discretisation step of hybrid automata we use δ to prescribe a discrete, but arbitrarily small step of the discretised automaton. According to Corollary 5.2.7 we can assume without loss of generality that all the δ have the form $\delta = \frac{1}{m}$ where $m \in \mathbb{N}$.

The following lemma tells us something about the relation between a discretised and granularity changed automaton and the π-transformed version of it. In a sense, Lemma 5.2.12 and Lemma 5.2.13 prove some kind of bi-simulation between the two transformations of a hybrid automaton.

Lemma 5.2.12 *Let $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, inv, guard, act)$ be a hybrid automaton and let $init_{\mathcal{H}}$ be a representative for a possible initial state of $\mathcal{H}$ with L_0 as its initial location. Let $\tilde{\mathcal{H}} = (X, \mathcal{L}, \mathcal{E} \cup \mathcal{E}', dif', inv', guard \cup guard' \cup guard'', act \cup act') = \Delta \star (\Gamma_{\frac{\delta}{n}}(\mathcal{H}))$, $init_{\tilde{\mathcal{H}}} = \Delta \star init_{\mathcal{H}}$ and $\hat{\mathcal{H}} = \pi(\mathcal{H}, init_{\mathcal{H}})$. Furthermore, let $\rho \in runs(\tilde{\mathcal{H}})$, σ_i, σ_j be two states of the run ρ with $\sigma_i \overset{\text{tr}}{\mapsto} \sigma_j$ and $\sigma_i \tilde{=} t_i$ with t_i is a state in the behaviour t of $\hat{\mathcal{H}}$. Then there exists an action A in the description of $\hat{\mathcal{H}}$ where $[\![A]\!]_{t_i,t_j}$ holds, with $\sigma_j \tilde{=} t_j$.*

Proof: The proof is by case analysis:
Case initial state: Initially in $\mathcal{H}$ the constraint formula $init_{\mathcal{H}}$ holds. The definition of $\Gamma_{\frac{\delta}{n}}$ given in 5.2.1 shows that $\Gamma_{\frac{\delta}{n}}$ leaves $init_{\mathcal{H}}$ unchanged. The definitions of π and $\star$ show that they transform $init_{\mathcal{H}}$ identically where we assume that $\Delta = \frac{n}{\delta}$ and $c = \frac{1}{\delta}$[13]. Furthermore, initially $state = L_0$ holds in the description of $\hat{\mathcal{H}}$ which fits the initial state of $\tilde{\mathcal{H}}$. This concludes the initial case.

Considering an arbitrary step $\sigma_i \overset{\text{tr}}{\mapsto} \sigma_j$ in $\tilde{\mathcal{H}}$ with $\sigma_i = (L_i, \phi_i)$ and $\sigma_j = (L_j, \phi_j)$. This step results from the translation of either a transition or a time step of $\mathcal{H}$. We will consider these two cases separately.
Case of a transition step of $\mathcal{H}$: We know the following facts:

- The discretisation of $\mathcal{H}$ has no effect on the guards or on the actions of $\mathcal{H}$.
- $\phi_i(\Delta \star guard((L_i, L_j))$ holds
- $\phi_j(x) = \phi_i(\Delta \star act((L_i, L_j), x))$
- Since σ_j is taken to be a successor state of σ_i we know that this step can also be taken by $\tilde{\mathcal{H}}$. Therefore, the corresponding guard added by the discretisation step cannot be violated. Formally this means that the following formula holds:
$$\phi_i(\Delta \star inv(L_j)[x_1/\Delta \star act((L_i, L_j), x_1), \ldots x_n/\Delta \star act((L_i, L_j), x_n)])$$

[13]Recall that according to Corollary 5.2.7 we can assume that δ is of the form $\frac{1}{m}$ with $m \in \mathbb{N}$.

The π-translation of $\mathcal{H}$ results in the description of the temporal logic specification $\hat{\mathcal{H}}$ that contains the following part:

$$\phi \quad \equiv \quad \ldots \Big(state = L_i \wedge state' = L_j \wedge \tag{5.9}$$

$$\pi(guard((L_i, L_j)) \wedge \tag{5.10}$$

$$\bigwedge_{x \in X} (x' = \pi(act((L_i, L_j), x))) \wedge \tag{5.11}$$

$$(\pi(inv\,(L_j)))[x_1/x'_1, \ldots, x_n/x'_n] \Big) \ldots \tag{5.12}$$

- From $t_i(state) = L_i$ and $t_j(state) = L_j$ it follows that the first part of ϕ, 5.9, fits to the pair of states (t_i, t_j).
- From

  $\phi_i(\Delta \star guard(L_i, L_j)) \quad = \quad$ [by Def. 4.3.14, def. of π and with $\Delta = \frac{n}{\delta}$, $c = \frac{1}{\delta}$]
  $\phi_i(\pi(guard(L_i, L_j))) \quad = \quad$ [by $\sigma_i \tilde{=} t_i$]
  $t_i(\pi(guard(L_i, L_j)))$

  it follows that formula 5.10 is satisfied in state t_i.
- From

  $t_j(x) = \phi_j(x) \quad = \quad$ [because of the preconditions]
  $\phi_i(\Delta \star act((L_i, L_j), x)) \quad = \quad$ [by Def. 4.3.14, def. of π, $\Delta = \frac{n}{\delta}$ and $c = \frac{1}{\delta}$]
  $\phi_i(\pi(act((L_i, L_j), x))) \quad = \quad$ [by $\sigma_i \tilde{=} t_i$]
  $t_i(\pi(act((L_i, L_j), x)))$

  it follows that formula 5.11 fits to the pair of states (t_i, t_j).
- From

  $\phi_i(guard''((L_i, L_j))$
  $=$ [by def. of $\Gamma_{\frac{\delta}{n}}$ and Def. 4.3.14]
  $\phi_i((\Delta \star inv\,(L_j))[x_1/\Delta \star act((L_i, L_j), x_1), \ldots, x_n/\Delta \star act((L_i, L_j), x_n)]$
  $=$ [by Def. 4.3.14 and def. of π]
  $\phi_i(((\pi(inv\,(L_j)))[x_1/\pi(act((L_i, L_j), x_1)), \ldots, x_n/\pi(act((L_i, L_j), x_n))]$
  $=$ [by $\sigma_i \tilde{=} t_i$ and $x'_k = \pi(act((L_i, L_j), x_k))$ for $k = 1, \ldots, n$]
  $t_i((\pi(inv\,(L_j)))[x_1/x'_1, \ldots, x_n/x'_n])$

  it follows that formula 5.12 fits to the pair of states (t_i, t_j) that concludes the transition step.

Case of a time step of $\mathcal{H}$:
Looking at a time step of $\mathcal{H}$, $\sigma_i \overset{\xi}{\mapsto} \sigma_j$ that is transformed to a discrete step in $\tilde{\mathcal{H}}$ by $\Gamma_{\frac{\delta}{n}}$ we know the following facts about the definition and the behaviour of $\tilde{\mathcal{H}}$:

1. For a run ρ of $\tilde{\mathcal{H}}$ it holds that $f_i : [0, t_i] \mapsto (X \mapsto \mathbb{Q})$ is constantly ϕ_i.
2. ξ is arbitrary but finite.
3. The source location and the target location are identical: $L_i = L_j$
4. For each $x \in X$:
   (a) $\dot{f}_x : (0, \xi) \mapsto 0$
   (b) $f_x(0) = \phi_i(x)$
   (c) $f_x(\xi) = \phi_j(x)$
5. The new discrete actions in the definition of $\tilde{\mathcal{H}}$ according to the definition of $\Gamma_{\frac{\delta}{n}}$ are:
   For all $(L, L) \in \mathcal{E}', x \in X : (act'((L, L), x) = \Delta \star (x + dif(L, x) * \frac{\delta}{n})$, where Δ can be chosen to be $\frac{n}{\delta}$.
6. The new discrete actions get the new additional guard:
   $guard'((L, L)) = \Delta \star inv(L)[x_1/act'((L, L), x_1), \ldots, x_n/act'((L, L), x_n)]$

Considering the π-translation of $\mathcal{H}$, ψ has the following formula as a disjunctive part:

$$\psi \equiv \ldots \Big(state = L_i \wedge state' = L_i \wedge \tag{5.13}$$

$$\bigwedge_{x \in X} (x' = x + dif(L_i, x)) \wedge \tag{5.14}$$

$$(\pi(inv(L_i)))[x_1/x_1 + dif(L_i, x_1), \ldots, x_n/x_n + dif(L_i, x_n)] \Big) \tag{5.15}$$

Now we check whether this action description fits to the pair of states (t_i, t_j).

- From $t_i(state) = L_i$ and with $state' = L_i$ from equation (5.13), it follows that $t_j(state) = L_i$. This shows that the first part of ψ fits to the pair of states (t_i, t_j).
- Let $x \in X$ with $dif(L_i, x) \neq 0$, then
  $\phi_i(guard'((L_i, L_i)))$
  $=$ [by definition of $guard'((L_i, L_i))$]
  $\phi_i((\Delta \star inv(L_i))[x_1/x_1 + dif(L_i, x_1) * \frac{\delta}{n}, \ldots, x_n/x_n + dif(L_i, x_n) * \frac{\delta}{n}])$
  $=$ [by $\sigma_i \tilde{=} t_i$, def. of π and with $\Delta = \frac{n}{\delta}$, $c = \frac{1}{\delta}$]

$t_i(\pi(inv(L_i))[x_1/x_1 + dif(L_i, x_1), \ldots, x_n/x_n + dif(L_i, x_n)])$

From this it follows that formula 5.15 is satisfied in state t_i and the corresponding action is enabled.

- From the definition of $\tilde{\mathcal{H}}$ in the time transition case we know for all x_i with $i = 1, \ldots, n$ that

$$\begin{array}{lcl} \phi_j(x_i) & = & \\ \phi_i(act'((L_i, L_j), x_i)) & = & \\ \phi_i(\Delta \star (x_i + dif(L_i, x_i) * \frac{\delta}{n})) & = & [\text{by } \Delta = \frac{n}{\delta}] \\ \phi_i(x + dif(L_i, x_i) & = & [\text{with } \sigma_i \tilde{=} t_i] \\ t_i(x + dif(L_i, x_i)) & = & [\text{with formula (5.14)}] \\ t_j(x_i) & & \end{array}$$

It follows that the formula 5.14 fits to the pair of states (t_i, t_j).

This concludes the transition part and thus the proof. □

The above lemma says that for every action taken in a discretised granularity changed hybrid automaton $\mathcal{H}$, there is a corresponding action that can be executed in the π-translated version of $\mathcal{H}$. The following lemma treats the reverse case. It says that for every action in the π-translated version of $\mathcal{H}$ there is a corresponding one in the discretised granularity changed version of $\mathcal{H}$.

LEMMA 5.2.13 *Let $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, inv, guard, act)$ be a hybrid automaton and let the constraint formula $init_{\mathcal{H}}$ be a representative for a possible initial state of $\mathcal{H}$ with L_0 as its initial location.*

Let $\tilde{\mathcal{H}} = (X, \mathcal{L}, \mathcal{E} \cup \mathcal{E}', dif', inv', guard \cup guard' \cup guard'', act \cup act') = \Delta\star(\Gamma_{\frac{\delta}{n}}(\mathcal{H}))$, $init_{\tilde{\mathcal{H}}} = \Delta\star init_{\mathcal{H}}$ and $\hat{\mathcal{H}} = \pi(\mathcal{H}, init)$. Furthermore, let $t = t_0, t_1, \ldots$ be a behaviour of $\hat{\mathcal{H}}$, let t_i be an arbitrary state of t and t_{i+1} be its successor state and let σ_i be a state of the run ρ of $\tilde{\mathcal{H}}$ with $\sigma_i \tilde{=} t_i$. Then there exists a discrete action in the description of $\tilde{\mathcal{H}}$ that can be taken in state σ_i resulting in a state σ_j with $\sigma_j \tilde{=} t_j$.

Proof: The proof of this lemma is by case analysis similar to Lemma 5.2.12. □

After the examination of the relation between a discretised hybrid automaton with changed granularity and its π-translated version, we have to analyse the relation of the properties of a hybrid automaton expressed in PSL. We consider in the following lemma the granularity changed version and the π-translated version of such a property.

Lemma 5.2.14 *Let $\mathcal{H}$ be a hybrid automaton and let $\tilde{\mathcal{H}}$ be the discretised, granularity changed version of $\mathcal{H}$, i.e. $\tilde{\mathcal{H}} = \Delta \star (\Gamma_\delta(\mathcal{H}))$. Furthermore, let $\rho = \sigma_0, \sigma_1, \sigma_2, \ldots$ be a run of $\tilde{\mathcal{H}}$ and let $\tau = s_0, s_1, \ldots$ be constructed according to Lemma 5.2.12 and Definition 5.2.2, i.e. $\forall s_i : \sigma_i \tilde{=} s_i$. Then*

$$\phi_i(\Delta \star t) = s_i(\pi(t))$$

with $\sigma_i = (L_i, \phi_i)$, t is a constraint term and $\Delta = c * n$, $c, n \in \mathbb{N}$.

Proof: The proof is by induction on the structure of term t.

Base case 1: Let $t \mathrel{\widehat{=}} state = L$ with $L \in \mathcal{L}$, then

$$\begin{array}{lcl} \phi_i(\Delta \star (state = L)) & = & \text{[by Definition 4.3.14]} \\ \phi_i(state = L) & = & \text{[by } \sigma_i \tilde{=} s_i\text{]} \\ s_i(state = L) & = & \text{[by Definition 5.1.2]} \\ s_i(\pi(state = L)) & & \end{array}$$

Base case 2: Let $t \mathrel{\widehat{=}} x$ with $x \in X$, then
$\phi_i(\Delta \star x) = \phi_i(x) = s_i(x) = s_i(\pi(x))$ by Definition 4.3.14 and 5.1.1.

Base case 3: Let $t \mathrel{\widehat{=}} r$ with $r \in R$ a rational valued constant, then
$\phi_i(\Delta \star r) = \Delta * r = \pi(r) = s_i(\pi(r))$ by Definition 4.3.14 and 5.1.1.

Induction step: Let $t = t_1 \circ t_2$ with $\circ \in \{+, -, *\}$, then

$$\begin{array}{lcl} \phi_i(\Delta \star (t_1 \circ t_2)) & = & \text{[by Definition 4.3.14]} \\ \phi_i((\Delta \star t_1) \circ (\Delta \star t_2)) & = & \text{[by induction hypotheses]} \\ \phi_i(\Delta \star t_1) \circ \phi_i(\Delta \star t_2) & = & \text{[by definition of } \pi \text{ and } s_i\text{]} \\ s_i(\pi(t_1)) \circ s_i(\pi(t_2)) = s_i(\pi(t_1 \circ t_2)) & & \end{array}$$

This concludes the proof. □

In what follows we establish the relationship between a discretised, granularity changed hybrid automaton and it's π-translated version. The operators introduced next are used in Theorem 5.2.17 that expresses the desired relationship.

Definition 5.2.3 (ˆ-operator)
Let $\mathcal{H}$ be a hybrid automaton, $\tilde{\mathcal{H}} = \Delta \star (\Gamma_\delta(\mathcal{H}))$, $\rho \in runs(\tilde{\mathcal{H}})$ and $\rho = \sigma_0 \overset{\text{tr}}{\mapsto} \sigma_1 \overset{\text{tr}}{\mapsto} \sigma_2 \overset{\text{tr}}{\mapsto} \ldots$ with $\sigma_i = (L_i, \phi_i)$. Then $\hat{\rho}$ is defined as follows: $\hat{\rho} = s_0, s_1, s_2, \ldots$ where s_i are states, i.e. valuations of variables and for all variables $x \in X$

$$s_i(x) := \begin{cases} L_i & \text{if } x = state \\ \phi_i(x) & \text{else} \end{cases}$$

COROLLARY 5.2.15 *Let $\mathcal{H}$ be a hybrid automaton, $\tilde{\mathcal{H}} = \Delta\star(\Gamma_\delta(\mathcal{H}))$, $\rho \in runs(\tilde{\mathcal{H}})$ then it holds for $\hat{\rho}$ that $\sigma_i \tilde{=} s_i$ for all i.*

Proof: The proof follows immediately from the definitions 5.2.2, 5.2.3, and the lemmata 5.2.12 and 5.2.13. □

DEFINITION 5.2.4 ($\breve{}$-OPERATOR)
Let $\mathcal{H}$ be a hybrid automaton, $\bar{\mathcal{H}} = \pi(\mathcal{H}, init)$ and let $t = s_0, s_1, \ldots$ be a behaviour of $\bar{\mathcal{H}}$. Then $\breve{t}$ is defined as follows: $\breve{t} = \sigma_0, \sigma_1, \sigma_2, \ldots$ with $\sigma_i := (L_i, \Phi_i)$ with $L_i := s_i(state)$ for the variable state. For all other variables x it holds that $\Phi_i(x) := s_i(x)$

LEMMA 5.2.16 *Let $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, \mathrm{inv}, guard, \mathrm{act})$ be a hybrid automaton. Let $\tilde{\mathcal{H}} = \Delta \star (\Gamma_\delta(\mathcal{H}))$ be the discretised granularity changed version of $\mathcal{H}$ and $\rho \in runs(\tilde{\mathcal{H}})$. Then the following holds:*

$$\{\rho'' \mid \mathrm{suf}(\rho'', \hat{\rho})\} = \{\hat{\rho'} \mid \mathrm{suf}(\rho', \rho)\}$$

Proof: $\sqsupseteq$:
We have a trace $\rho = \sigma_0, \sigma_1, \ldots$ of $\tilde{\mathcal{H}}$ with $\mathrm{suf}(\rho', \rho)$. Let $\rho' = \sigma_i, \sigma_{i+1}, \ldots$. Applying the $\hat{}$-operator to ρ' results in $\hat{\rho'} = s_i, s_{i+1}, \ldots$. In concatenating this sequence of states with the sequence $s_0, \ldots, s_{i-1} = \hat{\rho} \mid^{i-1}$ we obtain $\hat{\rho}$ which is a behaviour of $\hat{\mathcal{H}}$ according to Lemma 5.2.12. It follows that $\hat{\rho'}$ is a suffix of $\hat{\rho}$.

$\sqsubseteq$:
We have a trace $\hat{\rho} = s_0, s_1, \ldots$ of $\hat{\mathcal{H}}$ with $\mathrm{suf}(\rho'', \hat{\rho})$ according to Definition 5.2.3. Let $\rho'' = s_i, s_{i+1}, \ldots$. Applying the $\breve{}$-operator to ρ'' results in $\breve{\rho''} = \sigma_i, \sigma_{i+1}, \ldots$. In concatenating this sequence with the sequence $\sigma_0, \ldots, \sigma_{i-1} = \breve{\hat{\rho}} \mid^{i-1}$ we obtain ρ which is a behaviour of $\tilde{\mathcal{H}}$ according to Lemma 5.2.13. It follows that $\breve{\rho''}$ is a suffix of ρ and that $\hat{\breve{\rho''}} \in \{\hat{\rho'} \mid \mathrm{suf}(\rho', \rho)\}$. With $\hat{\breve{\rho''}} = \rho''$ we are done. □

According to the Lemmata 5.2.12, 5.2.13 and 5.2.2 we can assume in the following theorem that $\Delta = \frac{n}{\delta}$, $\delta = \frac{1}{m}$ and $c = \frac{1}{\delta}$ for $m, n \in \mathbb{N}$.

THEOREM 5.2.17 *Let $\mathcal{H} = (X, \mathcal{L}, \mathcal{E}, dif, \mathrm{inv}, guard, \mathrm{act})$ be a hybrid automaton, $init_{\mathcal{H}} = (L_0, \phi_0)$ be the initial state of $\mathcal{H}$ and let $\tilde{\mathcal{H}} = \Delta\star(\Gamma_{\frac{\delta}{n}}(\mathcal{H}))$, $init_{\tilde{\mathcal{H}}} = \Delta\star init_{\mathcal{H}}$ and $\hat{\mathcal{H}} = \pi(\mathcal{H}, init_{\mathcal{H}})$. Furthermore, let $(\sigma_0, \sigma_1, \ldots) = \rho \in runs(\tilde{\mathcal{H}})$ and let P be a PSL formula with no nested temporal operators. Then*

$$\tilde{\mathcal{H}}, \rho \models \Delta \star P \quad \text{if and only if} \quad [\![\pi(P)]\!]_\tau$$

where $\tau = s_o, s_1, \ldots$ is a behaviour of $\hat{\mathcal{H}}$ and $s_i \tilde{=} \sigma_i$ for all i.

Proof: The proof is by induction on the structure of P.

Base case 1: Let $P \mathrel{\widehat{=}} t_1 \circ t_2$ with $t_1, t_2 \in CT$ then

$$\begin{aligned}
\tilde{\mathcal{H}}, \rho \models \Delta \star (t_1 \circ t_2) \quad \text{iff} \quad & \phi_0(\Delta \star (t_1 \circ t_2)) \text{ with } \mathit{start}(\rho) = \sigma_0 \text{ and } \sigma_0 = (L_0, \phi_0) \\
\text{iff} \quad & s_0(\pi(t_1 \circ t_2)) \quad \text{[by Lemma 5.2.14]} \\
\text{iff} \quad & [\![\pi(t_1 \circ t_2)]\!]_{s_0} \quad \text{[by definition of } [\![.]\!]_\sigma\text{]} \\
\text{iff} \quad & [\![\pi(P)]\!]_\tau \text{ with } \tau = s_0, s_1, \ldots \text{ is a behaviour of } \tilde{\mathcal{H}} \\
& \text{according to Lemma 5.2.12 with } s_i \tilde{=} \sigma_i \text{ for all } i
\end{aligned}$$

Base case 2: Let $P \mathrel{\widehat{=}} \mathit{state} = L$ with $L \in \mathcal{L}$, then

$$\begin{aligned}
\tilde{\mathcal{H}}, \rho \models \Delta \star (\mathit{state} = L) \quad \text{iff} \quad & \tilde{\mathcal{H}}, \rho \models \mathit{state} = L \quad \text{[by Definition 4.3.14]} \\
\text{iff} \quad & \mathit{start}(\rho) = \sigma_0 = (L, \phi_0) \\
\text{iff} \quad & s_0(\mathit{state}) = L \quad \text{[by Lemmata 5.2.12 and 5.2.13]} \\
\text{iff} \quad & [\![\pi(\mathit{state} = L)]\!]_\tau \text{ is true with } \tau = s_0, s_1, \ldots \quad \text{[by} \\
& \text{definition of } [\![.]\!]_\sigma\text{]}
\end{aligned}$$

Induction steps:
Let $P \mathrel{\widehat{=}} \Box\mathcal{F}$, then

$$\begin{aligned}
\tilde{\mathcal{H}}, \rho \models \Delta \star (\Box\mathcal{F}) \quad \text{iff} \quad & \mathcal{H}, \rho \models \Box(\Delta \star F) \quad \text{[by Definition 4.3.14]} \\
\text{iff} \quad & \mathcal{H}, \bar{\rho} \models \Delta \star \text{ for all } \bar{\rho} \text{ with } \mathrm{suf}(\bar{\rho}, \rho) \quad \text{[by Definition 4.3.16]} \\
\text{iff} \quad & [\![\pi(F)]\!]_{\hat{\bar{\rho}}} \text{ for all } \mathrm{suf}(\bar{\rho}, \rho) \quad \text{[by induction hypothesis and} \\
& \text{with } \hat{\bar{\rho}} \text{ constructed according to Definition 5.2.3]} \\
\text{iff} \quad & [\![\pi(F)]\!]_{\rho'} \text{ is true for all } \rho' \text{ with } \mathrm{suf}(\rho', \rho) \quad \text{[by Lemma 5.2.16]} \\
\text{iff} \quad & [\![\pi(\Box F)]\!]_{\hat{\rho}} \text{ is true with } \hat{\rho} = s_0, s_1, \ldots \quad \text{[by Definition 5.1.2]}
\end{aligned}$$

Let $P \mathrel{\widehat{=}} \mathcal{F}_1 \wedge \mathcal{F}_2$, then it holds

$$\begin{aligned}
\tilde{\mathcal{H}}, \rho \models \Delta \star (\mathcal{F}_1 \wedge \mathcal{F}_2) \quad \text{iff} \quad & \tilde{\mathcal{H}}, \rho \models (\Delta \star \mathcal{F}_1) \text{ and } \tilde{\mathcal{H}}, \rho \models (\Delta \star \mathcal{F}_2) \\
\text{iff} \quad & [\![\pi(\mathcal{F}_1)]\!]_\tau \text{ is true and } [\![\pi(\mathcal{F}_2)]\!]_\tau \text{ is true} \quad \text{[by induction} \\
& \text{hypothesis]} \\
\text{iff} \quad & [\![\pi(\mathcal{F}_1) \wedge \pi(\mathcal{F}_2)]\!]_\tau \\
\text{iff} \quad & [\![\pi(\mathcal{F}_1 \wedge \mathcal{F}_2)]\!]_\tau \text{ holds with } \tau = s_0, s_1, s_2, \ldots \quad \text{[by} \\
& \text{Definition 5.1.2]}
\end{aligned}$$

The $\vee$-case is analogous to the $\wedge$-case. $\square$

Now we come to the main theorem which states the connection between a hybrid automaton and its π-translated version using discretisation and granularity change as a bridge between them.

THEOREM 5.2.18 (MAIN THEOREM) *Let $\mathcal{H}$ be a hybrid automaton with initial state $init_{\mathcal{H}}$ represented by σ, let P be an atemporal PSL formula, C a constraint formula, Q_1 and Q_2 PSL formulae with no nested temporal operators, c, $n \in \mathbb{N}$, $\delta \in \mathbb{Q}^+$ and let t be a behaviour of $\pi(\mathcal{H}, init_{\mathcal{H}})$ constructed according to Lemma 5.2.12 and Definition 5.2.2. Then the following holds:*

(i) $\mathcal{H}, \sigma \models \Box P$ if and only if $\forall c\colon \exists n\colon \pi(\mathcal{H}, init_{\mathcal{H}}), t \models \pi(\Box P)$

(ii) $\mathcal{H}, \sigma \models \Diamond P$ if and only if $\exists c\colon \forall n : \pi(\mathcal{H}, init_{\mathcal{H}}), t \models \pi(\Diamond P)$

(iii) $\mathcal{H}, \sigma \models Q_1 \wedge Q_2$ if and only if $\mathcal{H}, \sigma \models Q_1$ and $\mathcal{H}, \sigma \models Q_2$

Proof:
(i) Lemma 5.2.5 allows us to choose n to be 1. According to lemmata 5.2.12, 5.2.13 and 5.2.2 we can assume furthermore that $\Delta = \frac{1}{\delta}$, $\delta = \frac{1}{m}$ and $c = \frac{1}{\delta}$ for $m \in \mathbb{N}$. Thus, we can conclude as follows:

$\mathcal{H}, \sigma \models \Box P$ iff $\forall \delta \in \mathbb{Q}^+\colon \Gamma_\delta(\mathcal{H}), \sigma \models \Box P$ [by Theorem 5.2.6 and Lemma 5.2.5]

iff $\forall \delta \in \mathbb{Q}^+\colon \Delta \star (\Gamma_\delta(\mathcal{H})), \rho \models \Delta \star (\Box P)$ for all $\rho \in \text{runs}(\mathcal{H})$ where $\rho = \sigma, \sigma_1, \sigma_2, \ldots$ and $\text{start}(\rho) = \sigma$ [by Theorem 5.2.11]

iff for all $c \in \mathbb{N}$ it holds: $[\![\pi(\Box P)]\!]_t$ is true with $t = s_o, s_1, \ldots$ is a behaviour of $\pi(\mathcal{H}, init_{\mathcal{H}})$ and $\sigma_0 = \sigma$ and $s_i \tilde{=} \sigma_i$ for all i [by Theorem 5.2.17]

iff $\forall c \in \mathbb{N}\colon \pi(\mathcal{H}, init_{\mathcal{H}}), t \models \pi(\Box P)$

(ii) According to the Lemmata 5.2.12, 5.2.13 and 5.2.2 we can assume in the following proof that $\Delta = \frac{n}{\delta}$, $\delta = \frac{1}{m}$ and $c = \frac{1}{\delta}$ for $m, n \in \mathbb{N}$. Thus, we can conclude

as follows:

$$
\begin{aligned}
\mathcal{H},\sigma \models \Diamond P \quad \text{iff} \quad & \exists \delta \in \mathbb{Q}^+ \colon \forall n \in \mathbb{N} \colon \Gamma_{\frac{\delta}{n}}(\mathcal{H}), \sigma \models \Diamond P \quad \text{[by Theorem 5.2.6]} \\
\text{iff} \quad & \exists \delta \in \mathbb{Q}^+ \colon \forall n \in \mathbb{N} \colon \Delta \star (\Gamma_{\frac{\delta}{n}}(\mathcal{H})), \sigma \models \Delta \star \Diamond P \text{ for all } \rho \in \text{runs}(\mathcal{H}) \\
& \text{with } \rho = \sigma, \sigma_1, \sigma_2, \ldots \text{ and } \text{start}(\rho) = \sigma \quad \text{[by Theorem 5.2.11]} \\
\text{iff} \quad & \text{there exists a } c \in \mathbb{N} \text{ such that } \forall n \in \mathbb{N} \text{ it holds: } [\![\pi(\Diamond P)]\!]_t \text{ is true} \\
& \text{with } t = s_o, s_1, \ldots \text{ is a behaviour of } \pi(\mathcal{H}, init_{\mathcal{H}}) \text{ and } \sigma_0 = \sigma \\
& \text{and } s_i \tilde{=} \sigma_i \text{ for all } i \quad \text{[by Theorem 5.2.17]} \\
\text{iff} \quad & \exists c \in \mathbb{N} \colon \forall n \in \mathbb{N} \colon \pi(\mathcal{H}, init_{\mathcal{H}}), t \models \pi(\Diamond P)
\end{aligned}
$$

$$
\begin{aligned}
\text{(iii) } \mathcal{H},\sigma \models Q_1 \wedge Q_2 \quad \text{iff} \quad & \mathcal{H},\rho \models Q_1 \wedge Q_2 \text{ for all } \rho \in \text{runs}(\mathcal{H}) \text{ with } \text{start}(\rho) = \sigma \\
& \text{[by Definition 4.3.16]} \\
\text{iff} \quad & \mathcal{H},\rho \models Q_1 \text{ for all } \rho \in \text{runs}(\mathcal{H}) \text{ with } \text{start}(\rho) = \sigma \text{ and} \\
& \mathcal{H},\rho \models Q_2 \text{ for all } \rho \in \text{runs}(\mathcal{H}) \text{ with } \text{start}(\rho) = \sigma \quad \text{[by} \\
& \text{Definition 4.3.16]} \\
\text{iff} \quad & \mathcal{H},\sigma \models Q_1 \text{ and } \mathcal{H},\sigma \models Q_2
\end{aligned}
$$

□

Summarising the proof cases we have done a discretisation step according to the definition of $\Gamma_{\frac{\delta}{n}}$ and Theorem 5.2.6 that results in a discrete hybrid automaton $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ such that $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ satisfies the same PSL formulae as $\mathcal{H}$ does. Note that $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$ is an automaton that is parameterised with δ and n. In the next step we perform a granularity change according to Theorem 5.2.11 with $\Delta = \frac{n}{\delta}$. This results in an automaton $\mathcal{H}' = \Delta \star \Gamma_{\frac{\delta}{n}}(\mathcal{H})$ that satisfies $\Delta \star P$. Applying Theorem 5.2.17 we are done with the proof.

The purpose of $\mathcal{H}' = \Delta \star \Gamma_{\frac{\delta}{n}}(\mathcal{H})$ was to come as close as possible to $\pi(\mathcal{H})$ such that there is only one more simple transformation step in order to achieve $\pi(\mathcal{H})$. The property $\Delta \star P$ is identical to $\pi(P)$ with $\Delta = \frac{n}{\delta}$, $c = \frac{1}{\delta}$ and $\delta = \frac{1}{m}$ with $m \in \mathbb{N}$.

5.2.2.1 Summary of Discretisation and Granularity Change

The ultimate goal of the theoretical framework introduced above was to provide the means necessary for a translation from hybrid automata to VSE-II specifications. We ended up in a series of lemmata and theorems that can be combined to achieve this goal.

To this end consider the problem $\mathcal{H}, \sigma \models F$. The automaton $\mathcal{H}$ as well as the formula F might contain rational numbers. If so, we first normalise $\mathcal{H}$ as introduced

in Section 5.1.1. Then we perform a discretisation step which transforms $\mathcal{H}$ to $\Gamma_{\frac{\delta}{n}}(\mathcal{H})$. According to Corollary 5.2.7 we can assume without loss of generality that δ is a rational number of the form $1/m$, where m is a positive integer. We then change the granularity by $\Delta = n * m = \frac{n}{\delta}$. This results in an (infinite state) automaton that can immediately be translated into the VSE-II specification language.

The direct translation into VSE-II is described in Section 5, and together with the theorems in this section we showed the soundness and completeness proof for this translation.

The next section gives an idea of how to extend this method for the integration of Hybrid Automata into VSE-II to properties with nested temporal operators. Until now the presented methodology is not suited to handle nested temporal operators. Although many of the properties we want to prove about a system are of the kind $\Box P$ or $\Diamond P$ where P is an atemporal formula, we sometimes want to be able to handle nested temporal formulae as for example reactivity expressed as $\Box\Diamond P$.

5.3 Nested Temporal Operators

Until now we have introduced theoretical results and the translation schema to integrate Hybrid Automata into VSE-II to prove simple temporal formulae. That means the properties we are able to embed into the VSE-II environment from Hybrid Automata are limited to formulae with no nested temporal operators. The translation does not work for nested temporal operators because of the quantification of the variables n and c resulting from the π translation. A simple example will clarify this.

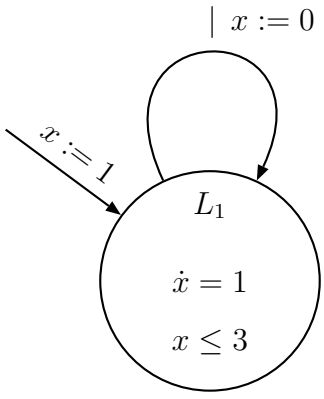

Figure 5.7: Simple Hybrid Automaton

EXAMPLE 5.3.1 *Let $\mathcal{H}$ be the hybrid automaton shown in Figure 5.7. and let*

- $P_1 : \Box x \leq 3$,
- $P_2 : \Diamond x = 0$ *and*
- $P_3 : \Box\Diamond x = 0$

be properties of the hybrid automaton. Looking at the π-translation of the hybrid automaton we get the following ϕ with the initial condition $x = n * c \wedge state = L_1$:

$$\begin{aligned}\phi \equiv\ & (state = L_1 \wedge state' = L_1 \wedge x' = 0 \wedge x' \leq 3nc) && (\phi_1)\\ & \vee\\ & (state = L_1 \wedge state' = L_1 \wedge \neg(x + 1 \leq 3nc) \wedge x' = 0 \wedge x' \leq 3nc) && (\phi_1^*)\\ & \vee\\ & (state = L_1 \wedge state' = L_1 \wedge x + 1 \leq 3nc \wedge x' = x + 1) && (\psi_1)\end{aligned}$$

This form is equivalent to the normal form:

$$x = n * c \wedge state = L_1 \wedge \Box[\phi_1, \phi_1^*, \psi_1]_x \wedge \mathcal{WF}_x(\phi_1^*) \wedge \mathcal{WF}_x(\psi_1)$$

The quantification of the variables n and c depends on the property to be proved. We have proven in the chapters before that in case of a safety property we can assume n to be 1 and so only c remains. For P_1 the translation process results in:

$$x = c \wedge state = L_1 \wedge \Box[\phi_1, \phi_1^*, \psi_1]_x \wedge \mathcal{WF}_x(\phi_1^*) \wedge \mathcal{WF}_x(\psi_1)$$

where all the occurrences of n in ϕ_1, ϕ_1^* and ψ_1 are substituted by 1.

In case of property P_2 we get the following formula:

$$\exists c \forall n(x = n * c \wedge state = L_1 \wedge \Box[\phi_1, \phi_1^*, \psi_1]_x \wedge \mathcal{WF}_x(\phi_1^*) \wedge \mathcal{WF}_x(\psi_1))$$

In case of property P_3 we can not proceed as in the two cases before. The reason is that the quantification over the variables c and n in the translation function π is only able to capture a single temporal operator with a hybrid automaton. There are no means in the machinery introduced so far to talk about several instances of hybrid automata with the corresponding quantifications over c and n.

An extension of this machinery is given in the next section. There we introduce a proposal how to deal with leads-to properties.

5.3.1 Leads-to Properties

Let us assume that we aim to prove a formula of the form $\Box(P \rightarrow \Diamond Q)$ where P and Q do not have further temporal operators. This means that we have to

check whether $(P \rightarrow \Diamond Q)$ holds for all reachable states. Now suppose we had two copies of the original hybrid automaton and we add transitions from each location of the first copy to its respective twin location without adding any guard or action. Then the original problem can be reformulated as: Changing from the first to the second automaton there are two possibilities in this situation. First, P does not hold and we are done since $P \rightarrow \Diamond Q$ holds in this case. Second, if P holds, then we have to check whether $\Diamond Q$ holds for the second automaton. Note that the "initial state" of the second automaton corresponds to an arbitrary reachable state of the first automaton in which P holds. Hence, the first copy is responsible for the first temporal operator and the second copy for the rest of the formula. In case of the property P_3, $\Box\Diamond P$, we proceed analogously.

The π-translation of an automaton or a formula introduces the variables c and n. Depending on the temporal operators of the properties to be proved they are universally or existentially quantified as indicated in Theorem 5.2.18. The π-translation of the second automaton has to introduce new variables c and n. This results in a nesting of first-order quantifiers originating from the nesting of temporal operators.

Applying this approach to the example shown in Figure 5.8 we get the automaton presented in Figure 5.8.

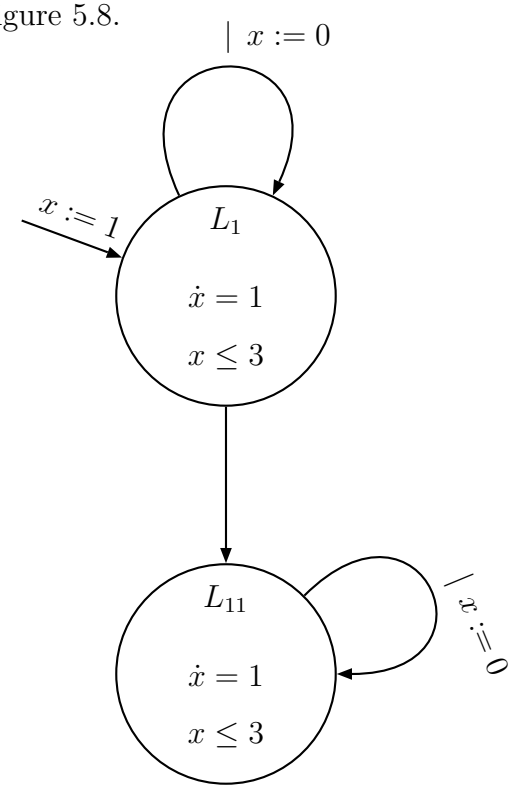

Figure 5.8: Copied hybrid automaton

In the presence of leads-to properties we have to generalise the given approach and adjust the translation function π. For the given example (see Figure 5.8) the translation has the following result:

$$
\begin{aligned}
&(state = L_1 \wedge state' = L_1 \wedge x' = 0 \wedge x' \leq 3n_1c_1) && (\phi_1)\\
&\vee\\
&(state = L_1 \wedge state' = L_1 \wedge \neg(x+1 \leq 3 * n_1c_1) \wedge x' = 0 \wedge\\
&\ x' \leq 3 * n_1c_1) && (\phi_1^*)\\
&\vee\\
&(state = L_1 \wedge state' = L_1 \wedge x' = x+1 \wedge x+1 \leq 3 * n_1c_1) && (\psi_1)\\
&\vee\\
&state = L_1 \wedge state' = L_{11} \wedge x' = x && (\chi_1)\\
&\vee\\
&(state = L_{11} \wedge state' = L_{11} \wedge x' = 0 \wedge x' \leq 3n_2c_2) && (\phi_{11})\\
&\vee\\
&(state = L_{11} \wedge state' = L_{11} \wedge \neg(x+1 \leq 3 * n_2c_2) \wedge x' = 0 \wedge\\
&\ x' \leq 3 * n_2c_2) && (\phi_{11}^*)\\
&\vee\\
&(state = L_{11} \wedge state' = L_{11} \wedge x' = x+1 \wedge x+1 \leq 3 * n_2c_2) && (\psi_{11})
\end{aligned}
$$

In case of property P_3 π computes the following formula:

$$
\begin{aligned}
&\forall c_1, \exists n_1\, \exists c_2\, \forall n_2 (x = n_1 * c_1 \wedge state = L_1 \wedge\\
&\Box[\phi_1, \phi_1^*, \psi_1, \chi_1, \phi_{11}, \phi_{11}^*, \psi_{11}]_x \wedge\\
&\mathcal{WF}_x(\phi_1^*) \wedge \mathcal{WF}_x(\psi_1) \wedge \mathcal{WF}_x(\phi_{11}^*) \wedge \mathcal{WF}_x(\psi_{11}))
\end{aligned}
$$

As well as the translation of the automaton also the translation of the property is changed. P_3[14] is translated to

$$\Box((state = L_1 \wedge state' = L_{11}) \rightarrow \Diamond x = 0)$$

Having a closer look at the π-translation we see that in changing P_3 to $P_3' = (\Box\Diamond x = 1)$ the result of π is:

$$\Box((state = L_1 \wedge state' = L_{11}) \rightarrow \Diamond x = n_2 * c_2)$$

The first part of the formula, $\Box(state = L_1 \wedge state' = L_{11})$, indicates the change from the first automaton to the second one. Semantically this means that we

[14] $state'$ stands for the value of the variable *state* in the next state.

are located in some arbitrary state which is the new initial state for the second automaton. In this new initial state the variables c_1 and n_1 introduced by the π-translation play an important role in the sense that the current values of these variables influence the new initial state. This is the reason why we only talk about n_2 and c_2 in the property P_3'.

So far this approach is limited to *response* or *leads-to* properties, arbitrary temporal formulae can be handled as sketched in the next section.

5.3.2 A more general Approach

A generalisation of the approaches given so far requires some extra machinery and cannot be done with the means we have at hand.

5.3.3 Invariants

There is a "normal" way to prove safety properties of the form $\Box\Phi$, with Φ atemporal. However, in practice this does not work satisfactorily. Instead, one tries to find a so called *invariant*, i.e. another atemporal formula $\Phi_\mathcal{I}$, that implies Φ and for which it is much easier to prove $\Box\Phi_\mathcal{I}$.

Formally, suppose we have to prove $\mathcal{H}, \sigma \models \Box\Phi$ and we know that

$$\begin{aligned} &\models && \Phi_\mathcal{I} \rightarrow \Phi \\ \mathcal{H}, \sigma \ &\models && \Box\Phi_\mathcal{I} \end{aligned}$$

Then, by *necessitation* we know that

$$\models \Box\left(\Phi_\mathcal{I} \rightarrow \Phi\right)$$

and therefore also

$$\mathcal{H}, \sigma \models \Box\Phi$$

Thus, whenever it becomes too complicated or even impossible to prove $\Box\Phi$ directly, the task is to find such a suitable $\Phi_\mathcal{I}$.

In general, finding such a suitable $\Phi_\mathcal{I}$ is a *creative act* and can hardly be automatised. Yet it often shows to be the only way to follow the original goal.

Certainly, it would be sufficient to find a less restrictive $\Phi_\mathcal{I}$. For instance, it would be enough to show

$$\mathcal{H}, \sigma \models \Box\left(\Phi_\mathcal{I} \rightarrow \Phi\right)$$

However, this is in general at least as complicated as the original problem and therefore would not be too helpful here.

5.3.4 Generalisation

Now consider the more general problem of proving $\mathcal{H}, \sigma \models \Box\Phi$, where Φ is *not necessarily atemporal*, i. e. Φ might contain further temporal operators.

In this case we proceed as above, i. e. we try to find an *atemporal* formula $\Phi_{\mathcal{I}}$ such that

$$\begin{aligned} & \models \Phi_{\mathcal{I}} \rightarrow \Phi \\ \mathcal{H}, \sigma & \models \Box\Phi_{\mathcal{I}} \end{aligned}$$

However, although this would certainly help to find such a $\Phi_{\mathcal{I}}$, we cannot seriously expect to find one. The problem lies with the Φ, which is not atemporal. Therefore, $\models \Phi_{\mathcal{I}} \rightarrow \Phi$ will only hold for the rather trivial cases.

Obviously, we have to take the Hybrid System into account when we consider Φ not atemporal. It would be nice to prove $\Phi_{\mathcal{I}} \rightarrow \Phi$ for all σ' that are reachable from σ in $\mathcal{H}$. Formally,

$$\mathcal{H}, \sigma \models \Box\left(\Phi_{\mathcal{I}} \rightarrow \Phi\right)$$

This together with $\mathcal{H}, \sigma \models \Box\Phi_{\mathcal{I}}$ would immediately lead to the desired result. However, it is not enough to find such a suitable candidate, we also have to prove that this formula holds. But how could we possibly do that? Certainly not with the translation technique, for, if Φ contains temporal operators we again have a formula with nested operators in hand.

One way out of the dilemma would be to find a $\Phi_{\mathcal{I}}$ such that

$$\mathcal{H}, \sigma' \models \Phi_{\mathcal{I}} \rightarrow \Phi$$

for all σ', not only for those that are reachable from σ.

Then, together with $\mathcal{H}, \sigma \models \Box\Phi_{\mathcal{I}}$ we also immediately obtain the desired result.

The general schema thus looks as follows:

$$\begin{aligned}
\mathcal{H}, \sigma &\models C && \text{iff } \sigma \models C, \text{ for constraint formulae } C \\
\mathcal{H}, \sigma &\models \text{state} = L && \text{iff } \sigma = (L, \phi) \\
\mathcal{H}, \sigma &\models \Phi_1 \wedge \Phi_2 && \text{iff } \mathcal{H}, \sigma \models \Phi_1 \;\; \& \;\; \mathcal{H}, \sigma \models \Phi_2 \\
& \text{and similarly for the other boolean connectives} \\
\mathcal{H}, \sigma &\models \Box\Phi && \text{iff } \mathcal{H}, \sigma \models \Box\Phi_{\mathcal{I}} \;\; \& \\
& && \quad \mathcal{H}, \sigma' \models \Phi_{\mathcal{I}} \rightarrow \Phi \text{ for all } \sigma' \\
& && \quad \text{for some suitable atemporal } \Phi_{\mathcal{I}} \\
\mathcal{H}, \sigma &\models \Diamond\Phi && \text{iff } \mathcal{H}, \sigma \models \Diamond\Phi_{\mathcal{I}} \;\; \& \\
& && \quad \mathcal{H}, \sigma' \models \Phi_{\mathcal{I}} \rightarrow \Phi \text{ for all } \sigma' \\
& && \quad \text{for some suitable atemporal } \Phi_{\mathcal{I}}
\end{aligned}$$

The latter two cases only apply in case Φ contains further temporal operators. Otherwise the discretisation can be done immediately.

Thus, the original problem (with nested operators) is reduced to one or more subproblems with fewer temporal operators. Obviously, deeper nestings of operators then require a recursive descent through the PSL-formula.

As an example, suppose we want to prove that $\mathcal{H}, \sigma \models \Box\Diamond(C_1 \wedge \Diamond C_2)$. A direct translation would not work. Therefore, we have to find an atemporal Φ_1 such that

$$\begin{aligned} &\mathcal{H}, \sigma \models \Box\Phi_1 \\ &\mathcal{H}, \sigma' \models \Phi_1 \rightarrow \Diamond(C_1 \wedge \Diamond C_2) \qquad \text{for all } \sigma' \end{aligned}$$

This second proof obligation still contains some nested operators. Thus, again, we have to find a suitable atemporal Φ_2 such that

$$\begin{aligned} &\mathcal{H}, \sigma \models \Box\Phi_1 \\ &\mathcal{H}, \sigma' \models \Phi_1 \rightarrow \Diamond\Phi_2 \qquad \text{for all } \sigma' \\ &\mathcal{H}, \sigma' \models \Phi_2 \rightarrow C_1 \wedge \Diamond C_2 \qquad \text{for all } \sigma' \end{aligned}$$

All the remaining proof obligations do not contain any nested temporal operators and therefore can be handled with the machinery described in this work, provided one is able to find such suitable atemporal formulae Φ_1 and Φ_2. In general, this is a creative act.

6

Observer Methodology

Thirty years ago, i.e. in the early seventies of last century, formal methods in software engineering had rather limited scope and the subject was well defined. However, classical code verification in Hoare style with a verification condition generator and subsequent proving of the verification conditions failed not only because of its inability to cope with complexity but also because this restricted approach did not meet important needs in software engineering. Meanwhile the situation has changed drastically and we are confronted with an enormous variety of formal approaches and tools. Although we have seen enormous progress in the development of automatic methods, still little attention has been paid to the integration of the many different and often highly specialised approaches and to the overall development process, in particular the early stages of software development.

Formal techniques for requirements analysis usually deal with particular aspects of the system to be designed. Examples are properties of information flow, correctness of (cryptographic) protocols, or real-time behaviour. It is not difficult to imagine a system that separates different applications by controlling the flow of information between them using authentication protocols as one of the security mechanisms and that has to satisfy certain additional real-time requirements.

Although it is well known that many errors occur in the early stages of the development process, later design stages like architectural design and implementation are also error prone and have to be treated formally in case of high assurance

levels. For example level EAL5 of the Common Criteria (CC) [1] requires a formal high-level design and a "correspondence proof" covering the so-called *Functional Specification*. In the case of real-time systems one has to define *how* the intended global behaviour is realized by separating the control component from its environment and assumptions (like delays, cycle time) have to be made explicit. This scenario is described in Section 6.2.

In other words, various views for requirements analysis have to be linked to a single abstract system specification that serves as a starting point for the refinement process (see Figure 6.1). Rather than having a `satisfies` relation between a specification and a collection of simple properties, requirements analysis will be based on its own descriptions (views) and postulated properties that refer to these descriptions. The description of a particular view will not necessarily use the same terminology as the system specification and often application specific formalisms and tools provide the means for an efficient analysis. For example, for establishing information flow properties a technique called non-interference, which is based on closure properties of sets of (system) traces, has to be used [71]. The analysis of protocols is based on a different kind of traces [74] that include steps of an attacker. A number of tools, like for example the one described in [73], has been used in this area. The real-time view can be implemented by Hybrid Automata [14] or by Timed Automata [16]. Tools like HyTech [48] provide efficient techniques to establish real-time properties.

In the following we present a general technique called *observer models* to link abstract descriptions of the real-time behaviour of a system to a system specification consisting of a control component, an environment, and a clock by means of an *observer mapping*. After an outline of the general technique we shall illustrate our approach using the specification of the Controller[1] for a gasburner as an example.

6.1 Observer Models for Real-Time Properties

Requirements of a system are usually specified and analysed by different formalisms that are specific for a particular view on that system. The choice of the formalisms can be influenced by several factors:

- preferences or expertise of the user,
- special features that need certain constructs in the specification language,
- particular system support, or
- the re-use of already existing specifications.

[1] Whenever we talk about system components in general we use small capitals, but special components of our methodology are written with a capital letter. Special components of a specification are denoted in typewriter format, `Controller` for example.

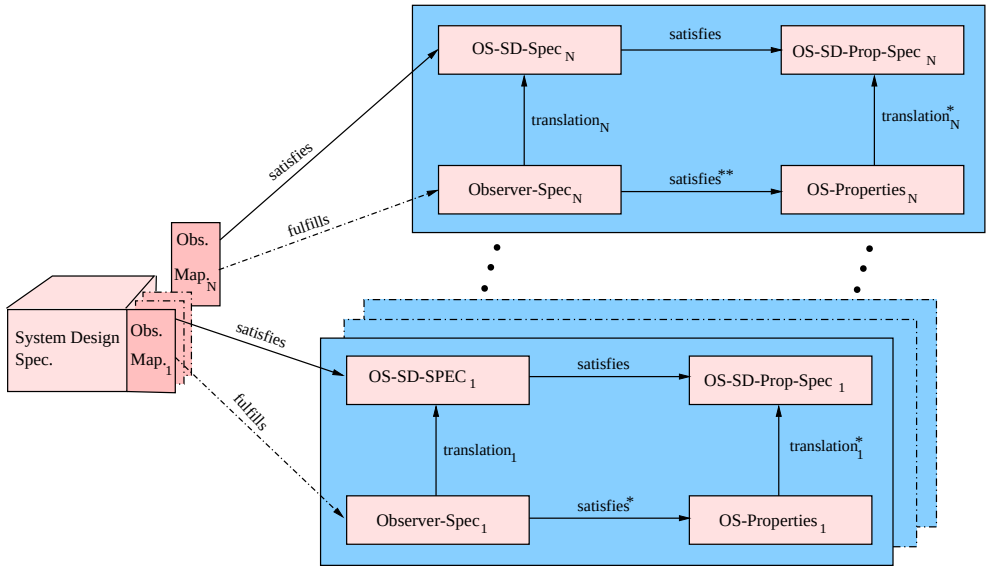

Figure 6.1: Observer Models

In Figure 6.1 each view is represented by an `Observer-Spec`$_i$ following its own description technique and formalism. One of these specifications might contain a global description of the runs of a protocol while another view concentrates on real-time properties. In the following we shall assume that the real-time view is given by Hybrid Automata [14].

A view will also include properties called `OS-Properties`$_i$ in Figure 6.1 that have to be established from the Observer specification. Real-Time requirements can be formulated and proven using tools like HyTech [48]. Note that we consider Hybrid Automata as a kind of comprehensive tool for the description of the entire system behaviour with respect to timing constraints. As can be seen from our example, the description is *global* in the sense that it does not distinguish between the control part of a system and its environment. States of the Hybrid Automaton therefore do not directly correspond to internal states of the controller. They rather describe certain situations that might occur in a run of the components constituting the complete system specification as there are a controller, an environment, and a clock in our general specification schema presented in Section 6.2.

To integrate various views into a common formal development, the `Observer-Spec`$_i$ and the `OS-Properties`$_i$ have to be translated into the language used to specify `System Design Spec`. The resulting specifications are `OS-SD-Spec` and `OS-SD-Prop-Spec`. The translation of Hybrid Automata into the specification language of VSE-II (see `translation`$_1$ in Figure 6.1) was presented in Section 5.

In Section 5 it is shown that the `satisfies` relation between `OS-SD-SPEC`$_i$ and `OS-SD-Prop-Spec`$_i$ holds, if and only if the `satisfies` relation holds between

`Observer-Spec`$_i$ and `OS-Properties`$_i$[2]. First of all this means that results obtained by using a tool like HyTech can be safely integrated into the overall development. However, since the language of `OS-SD-Spec` and `OS-SD-Prop-Spec` is more expressive (than that of Hybrid Automata) requirements specifications that are still "in the style of Hybrid Automata" but more general than these can be used in this context if one is inclined to use deductive techniques instead of model-checking. As an example one might wish to consider state transitions where one of the parameters (like speed) changes arbitrarily.

We are still left with the problem of establishing a link between the system specification and the particular real-time view we have defined. This is done by a mapping (called `Obs.Map`$_i$ in Figure 6.1) that (in the case of real-time requirements) interprets a given state of the interleaved computation of the controller, the environment, and the clock as a state of the (translation of the) hybrid automaton. It thereby turns the entire controller scenario into a model (in the sense of Hybrid Automata). For this we need to be sure that the translation faithfully preserves the semantics of Hybrid Automata. This result is established is Section 5.

6.2 A General Specification Scheme for Observer Models

The general scenario (see Figure 6.2), which we shall instantiate now using a realistic yet abstract gasburner specification, consists of three components: an Environment, a Controller and an Observer/Clock component. Which role do the individual components play? Generally, given a system design it is not always obvious which parts are to be assigned to the Environment and which parts belong to the Controller. In the application of formal methods we are often interested in the safety critical parts of the system to be developed. The other parts are considered to be irrelevant for the safety of the system. These parts consist for example of monitoring units[3]. It is important that the Controller, which possibly needs to be refined later, should contain all the safety critical parts.

The behaviour of the Environment is given by the specification of its interface, i.e. the Environment supplies the values for the various system interfaces (in time). To guarantee the right functioning of the system we have to make assumptions about the correct behaviour of the Environment[4], which can then be used in the proof of the system's properties. If the Environment component does not only exist as an interface definition but also as a component with accurately specified

[2]This is proved for the described real-time observation and for the specification languages VSE-SL and Hybrid Automata respectively.

[3]Of course, this is not always an uncritical unit. Just think of flight control devices that provide the pilot with the actual status of the plane.

[4]If one is interested in fault-tolerant systems, the possible faults can be described in the specification of the Environment, so that the control system must detect these and behave accordingly.

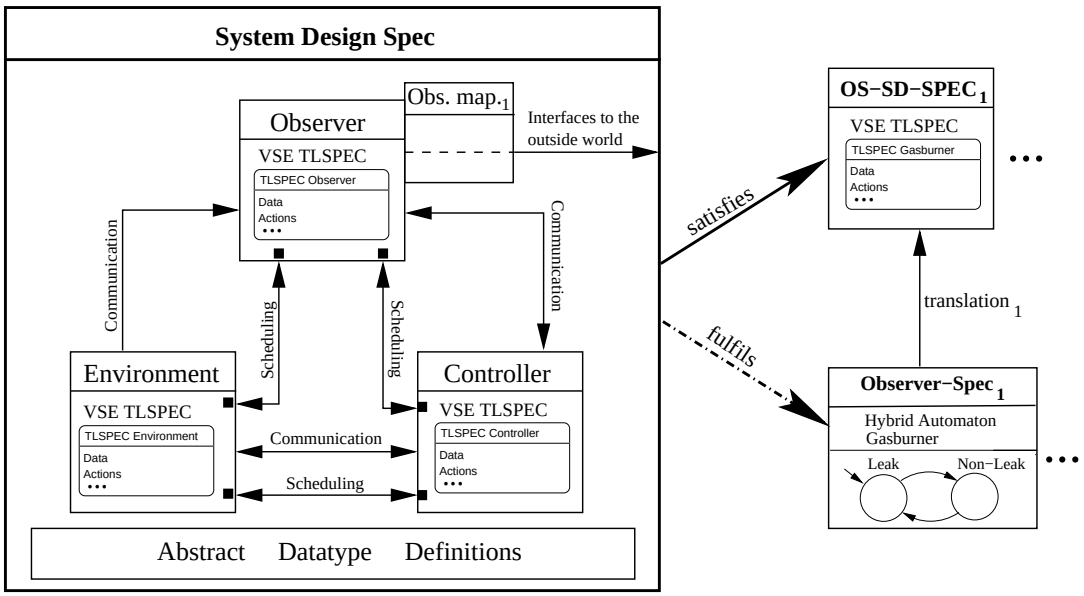

Figure 6.2: General Scenario (Observer Model Instantiated)

behaviour, then one can prove these assumptions about the Environment based on its behaviour. Of course, the type, range, and depth of the specification of both the Environment and Controller depend on the properties that should be fulfilled.

The Environment and the Controller are specified as usual temporal logic specifications. Both components can be structured into subcomponents.

The specification of the Observer/Clock component differs in some aspects from these usual specifications. One of the tasks of the Observer is that it is responsible for the time. But this fact does not influence the method described here. There could equally be an additional component which manages the time. The essential part of the Observer is that it observes the behaviour generated by the Controller and the Environment. These observations are filtered by the Observer and communicated to the outside world. This filtration of the behaviour of the whole system constitutes a special view on the system that will be a real-time view in our example described in Section 6.3. This is indicated by the right part of Figure 6.2 consisting of Observer-Spec.$_1$ (instantiated by the Hybrid Automaton Gasburner) and the translation of this gasburner into a VSE-II specification (see OS-SD-SPEC$_1$ in Figure 6.2). The languages used in the real specification are VSE-SL (VSE-II-Specification Language) and Hybrid Automata as indicated in Figure 6.2.

6.3 Gasburner as Real-Time Observation

The real-time scenario is obtained from the the general scenario in Figure 6.2 by instantiating it for the Observer, Environment and Controller components, i.e. the TLSPECS `Environment`, `Controller` and `Observer` respectively. Also we have to instantiate the `System Design Spec`, $\texttt{OS-SD-SPEC}_1$ and $\texttt{Observer-Spec}_1$ from Figure 6.1. A screenshot of the VSE-II development graph [52, 54] of the implementation of the gasburner scenario is given in Figure 6.3 where the `Environment`[5], the `Controller` and the `Observer` component are combined to the `gasburner` component representing the real gasburner.

Next, we show that the `gasburner` component satisfies the observation which is represented by the VSE-II specification resulting from the translation of the hybrid gasburner into VSE-II [77] as shown in Figure 6.3 by the temporal logic specification of `gasprop`[6]. We see that this observation given as a VSE-II specification represents a complex real-time property of the realistic gasburner.

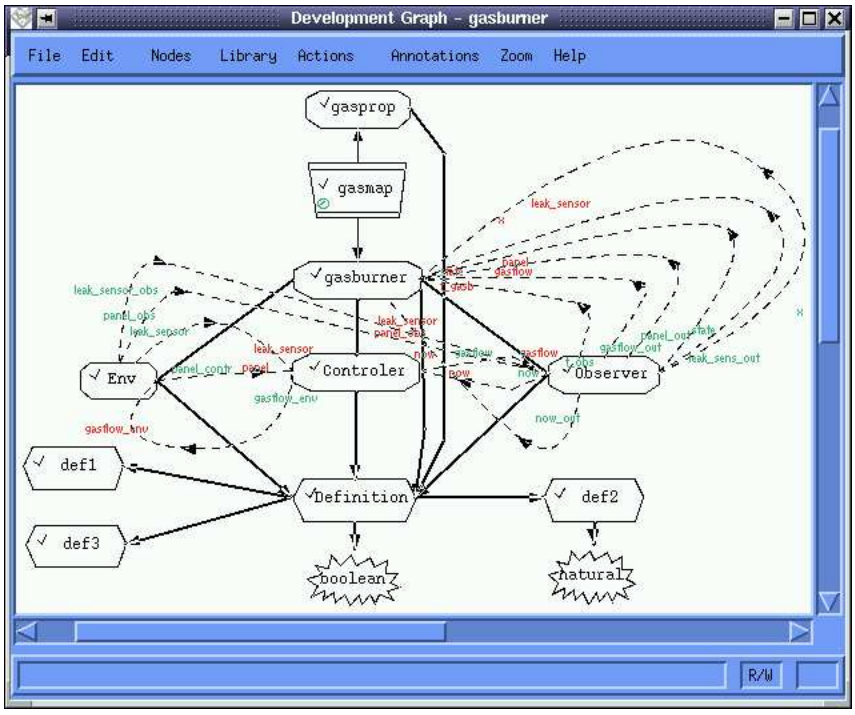

Figure 6.3: Development Graph of the Real Gasburner

[5] In Figure 6.3 we use `Env` to name the environment component.

[6] This corresponds to the $\texttt{OS-SD-SPEC}_1$ specification in the general scenario shown in Figure 6.2.

6.3.1 Underlying Datatypes

The definitions of the datatypes are located in the theories `def1`, `def2`, `def3`, and `Definition` in Figure 6.3. They define the possible values of the flexible variables of the component specifications. The definition of the datatypes is given in Figure 6.4. The whole system can either be "switched on" or "switched "off". These

```
THEORY def3
 TYPES OnOff_t =
       ENUMERATED BY on |
                     off
THEORYEND

THEORY def1
 TYPES gas_t =
       ENUMERATED BY blocked | run
THEORYEND

THEORY Definitions
 USING def1; def2; def3; boolean
 TYPES schedule_t =
       ENUMERATED BY obs | env | contr
Theoryend

THEORY def2
 USING natural
 TYPES state_t =
       ENUMERATED BY leaking |
                     non_leaking
 FUNCTIONS c : nat
 AXIOMS c > 0
THEORYEND
```

Figure 6.4: Theories Defining the Underlying Datatypes

states are described in the datatype `OnOff_t`. The datatype `gas_t` contains the states the gasflow can be in. It flows out of the nozzle of the gasburner (indicated by `run`) or it is blocked (indicated by `blocked`). The datatype `state_t` represents the virtual state of the real system, i.e. the system can be leaking or not leaking represented by `leaking` and `non_leaking`, respectively. If the system is in the leaking state, then unburned gas flows out of the nozzle of the gasburner. The system is non-leaking if the gas is burned or there is no gasflow at all. Finally we need a scheduling datatype (`schedule_t`) which is used to determine the possible

interactions between the different components.

A central role in the scenario plays the definition of the constant c[7]. As presented in the theory of the integration of hybrid automata into VSE-II in Section 5, c is used to realize an exact discretisation of a hybrid automaton. Note that it is defined as a natural number greater than zero.

6.3.2 Scheduling

The scheduling between the components of the gasburner, i.e. `Environment`, `Controller`, and `Observer` is realised with the help of the shared variable `who`. In the absence of such a scheduling variable, component steps are simply interleaved and can happen whenever they are enabled. Thus, the scheduling implements in some sense a filter for the enabling conditions. Combining the components without scheduling would result in a situation where we did not know which component computes next. In particular, we would not know when the observer makes its next observation. In order to define an observer component that is not too complex and in order to guarantee that the observer really observes something when there are significant changes we control the interleaving by a scheduler where computation of the components starts with the observer followed by the controller. After this first phase the computing order of the components is: environment, observer, controller. The scheduling is illustrated in Figure 6.5. Of course, there are other

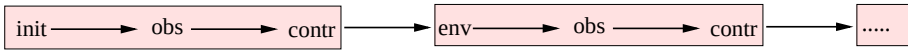

Figure 6.5: Scheduling

possibilities to schedule the components. One of these is to start with an initial sequence consisting of observer, controller and again the observer and after that the repeated computation order is environment, observer, controller and observer. This scheduling is shown in Figure 6.6. It turned out that this scheduling results in

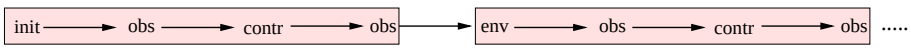

Figure 6.6: A Scheduling Alternative

a more complicated behaviour description of the Observer and in more complicated proofs. The implementation of the scheduling can be taken from the description of the various components shown in Figures 6.10, 6.11, 6.8 and 6.7.

The chosen scheduling assures that all the values of `now` become visible to the outside world, i.e. every point in time has an observation associated to it.

[7]The variable n introduced in Section 5 is omitted here since we are dealing only with safety properties (see also Theorem 5.2.6 and the corresponding Lemma 5.2.5).

6.3.3 Environment

The specification of the `Environment` component in VSE-II is shown in Figure 6.7. It consists of the definition of the possible initial states and the actions that the

```
TLSPEC Env
 PURPOSE  "Specification of the Environment"
 USING Definitions
 DATA OUT panel_contr, panel_obs : OnOff_t
      OUT leak_sensor_obs, leak_sensor : bool
      IN gasflow_env : gas_t
      SHARED INOUT who : schedule_t
 ACTIONS
 env_act ::= /*scheduling*/
             who = env AND who' = obs AND
             (/*switch the gasburner on or off*/
              (panel_contr' = on AND panel_obs' = on) OR
              (panel_contr' = off AND panel_obs' = off)) AND
              /*Environment behaves correct*/
             IF (panel_obs' = off OR gasflow_env = blocked)
             THEN (leak_sensor' = F AND leak_sensor_obs' = F)
             ELSE (leak_sensor' = T AND leak_sensor_obs' = T) OR
                  (leak_sensor' = F AND leak_sensor_obs' = F) FI
 SPEC INITIAL leak_sensor = T AND
              leak_sensor_obs = T AND
              panel_obs = on AND
              panel_contr = on AND
              who = obs
      TRANSITIONS [env_act] {panel_obs, panel_contr, who,
                             leak_sensor, leak_sensor}
TLSPECEND
```

Figure 6.7: Specification of the Environment

environment can perform[8]. Note that, because of the initialisation shown in Figure 6.7, the initially observed situation is leaking.

The action the environment can take is described in the `ACTIONS`-slot by the action `env_act`. The environment is specified such that the `panel` can change arbitrarily, but the `leak_sensor` representing the sensor to measure whether unburned gas flows out has to work correctly. This means that in a situation in which the `panel` is `off` or the gasflow is `blocked` the sensor is expected to deliver the

[8]The doubling of the variables has technical reasons. It simulates an identical output sent to several components.

right values. Specifying the environment this way results in the assumption that the environment is expected never to fail. Faulty sensors are excluded from the model[9].

6.3.4 Controller

The TLSPEC of the `Controller` component is given in Figure 6.8 and has the following external interfaces:

- `gasflow_env` and `timer` as output
- `leak_sensor`, `now` and `panel` as input and
- `cstate` as internal interface.

Its initial state description forces the gasburner observation to start in the `leaking` state. Moreover, the values of the controller's variables fit to the values of the environment's variables (and similarly for the initial value of the scheduling variable `who`). The actions the controller can take are described in the `ACTIONS` slot by `A1` through `A7`. Actions `A1`, `A2` and `A7` describe the actions in which the `panel` is `off`[10]. The result of switching the gasburner off depends on the system's previous state. This state is given by the values of the variables `leak_sensor` and `cstate`. Depending on this the control state `cstate` is set to `off`, the gasflow is stopped and the timer is started. The safety property we want to prove and which states that there is at most five percent of leakage time in observation intervals with a certain minimal length cannot be bypassed by simply switching the system off and on again. This is prevented by setting the timer in action `A2` to the actual value of `now`. If we had reset the variables after switching off the system, the safety property would be violated. When the system is switched on, i.e. `panel` is set to `on`, the actions `A3`, `A4`, `A5` and `A6` describe the possible actions of the Controller. For instance, `A5` checks whether the elapsed time since the last blocking of the gasflow is greater than or equal to $30 * \texttt{c}$ time units. In this case the gasflow gets rerun. The behaviour of the controller's actions is explained in the following enumeration.

`A1` reacts on the gasburner being switched off by blocking the gasflow when there is no leak.

`A2` handles the situation in which the gasburner is switched off but is still leaking. Because of the definition of the environment we see that this action is never enabled and we could remove it from the system description[11]. Such an action would be needed if the environment's sensors could deliver faulty values.

[9] Faulty sensors can be specified by giving them the possibility to change non-deterministically. Such problems do not concern us in this work, though.

[10] Only the Environment `env` can change the value of `panel`.

[11] Whether an action is never enabled has to be proven. Here we can show that the property $\Box(\texttt{panel} = \texttt{off} \Rightarrow \texttt{leak\_sensor} = \texttt{F})$ holds.

```
TLSPEC Controller
 USING Definitions
 DATA INTERNAL cstate : OnOff_t
      OUT timer : nat;
          gasflow_env, gasflow : gas_t
      IN leak_sensor : bool;
         now : nat;
         panel : OnOff_t
      SHARED INOUT who : schedule_t
 ACTIONS
 A1 ::= cstate = on AND panel = off AND leak_sensor = F AND
        cstate' = off AND gasflow' = blocked AND
        gasflow_env' = blocked AND UNCHANGED(timer)
 A2 ::= cstate = on AND panel = off AND leak_sensor = T AND
        cstate' = off AND gasflow' = blocked AND
        gasflow_env' = blocked AND timer' = now
 A3 ::= cstate = on AND panel = on AND leak_sensor = F AND
        UNCHANGED(cstate, gasflow, gasflow_env, timer)
 A4 ::= cstate = on AND panel = on AND leak_sensor = T AND
        cstate' = off AND gasflow' = blocked AND
        gasflow_env' = blocked AND timer' = now
 A5 ::= cstate = off AND panel = on AND
        now >= timer + (30 * c) AND
        cstate' = on AND gasflow' = run AND
        gasflow_env' = run AND UNCHANGED(timer)
 A6 ::= cstate = off AND panel = on AND
        now < timer + (30 * c) AND
        UNCHANGED(cstate, gasflow, gasflow_env, timer)
 A7 ::= cstate = off AND panel = off AND
        UNCHANGED(cstate, gasflow, gasflow_env, timer)
 A1to7 ::= who = contr AND who' = env AND
           A1 OR A2 OR ... OR A7
 /*Definition of the behaviour of the gasburner */
 SPEC INITIAL gasflow = run AND
              gasflow_env = run AND
              cstate = on AND
              timer = 0 AND
              who = obs
      TRANSITIONS [A1to7]
                  {gasflow, gasflow_env, timer, cstate}
TLSPECEND
```

Figure 6.8: Specification of the Controller

`A3` describes a situation in which the controller does not have to react, since there is no leaking indicated by the leak sensor. This action leaves the variables unchanged.

`A4` reacts to the leaking sensor of the environment. Since the system is in a leaking state, the gasflow is blocked and the timer is started. A consequence is that the gasflow can be opened again at the earliest $30 * c$ time units later (see action `A5`).

`A5` checks whether the elapsed time since the last blocking of the gasflow is greater than or equal to `30 * c` time units. If this is the case, the gasflow is run again.

`A6` handles the situation in which the time spent in a non-leaking situation, where the control state is set to `off`, is not yet greater than $30 * c$ time units. In this case all the variables remain unchanged, i.e. the gasflow stays blocked.

The actions are disjunctively connected to a single action `A1toA7`. Since in every moment of the execution only one of these actions is enabled, only this very action can be taken unless the controller stutters.

Up to now we have specified the controller and its environment. By adding a clock component that mimics the flow of time (the change of the variable `now`) we end up in a system for which we can prove some real-time properties.

As described in Section 6 our methodology is different in the sense that we are interested in a complete real-time perspective on a system rather that in particular aspects of it. Thus, it is not our main aim to prove single real-time properties of a system, we rather want to check its entire real-time behaviour. How this is realised is explained in the following section.

6.3.5 Observer

As can be seen from the specification of the Controller and the Environment, a simple refinement mapping [7] is not enough to prove the refinement. One reason is that in a refinement mapping only those variables can be mapped that are known in the actual state. In this sense a refinement mapping is a filter that maps the states of the implementing system to the states of the implemented system. The refinement model described here uses the usual refinement mapping extended by an observer component. The observer component together with the refinement mapping represents an external observer which filters the observed behaviour and maps these filtered behaviours to special observer behaviours represented in our case by the translated hybrid gasburner as shown in Section 6.3.6. As already mentioned in Section 6, we might look at the system from different angles, which could be a data flow perspective or, as in the example presented here, a real-time perspective. The responsibility of the observer is to map the states of the

real gasburner to the (virtual) states of the translated hybrid gasburner. In the simplest case the mapping consists only of a variable mapping which just renames or recomputes the values of the variables according to certain given functions. The mapping of the variable `state` of the translated hybrid gasburner is somewhat more complicated since there is no immediate correlation between variables of the implementing system and the variable `state`. Moreover, the mapping does not only depend on the actual state of the implementing system but also on some other information given by the behaviour of the observer. This situation is illustrated in Figure 6.9. Some of the steps of the hybrid gasburner in Figure 6.9 relate to

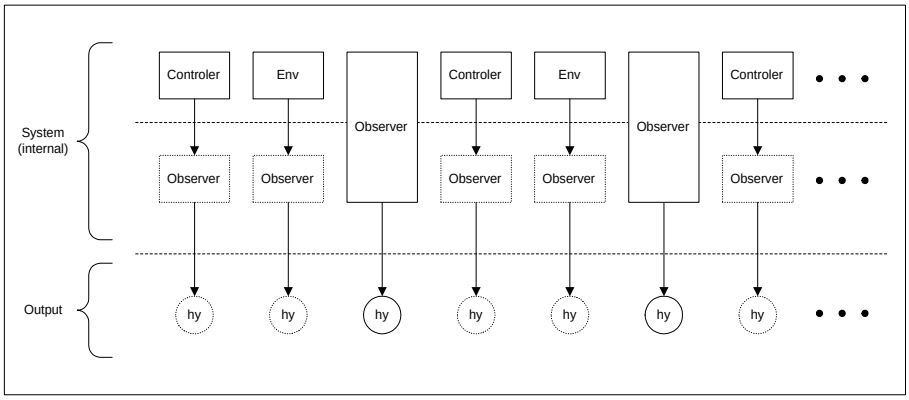

Figure 6.9: Refinement Relation

stuttering steps and some relate to real steps. Thus, the observer calculates steps of the hybrid gasburner from the steps of the environment and the controller. In addition, the observer makes the clock available to all the system components.

We now come to the actual specification of the Observer (given in Figures 6.10 and 6.11).

The observer's responsibility is to observe the controller and the environment and to map their behaviours to the behaviour of the hybrid gasburner. It thus determines our view or perspective on the system from the outside.

This specification should not rebuild the specification of the hybrid gasburner shown in Section 6.3.6. This can be accomplished by certain syntactical restrictions on the observer's actions. In order to enable or disable the actions the observer can take, it is only allowed to use the input variables sent from the controller or the environment. Also, the variables used in the refinement mapping `now_out`, `x`, `t_obs` and `state` may only be used in equations (in-equations) where there is at least one primed occurrence of that variable (with the exception of the initial state). In other words, these variables are not allowed to influence the behaviour of the observer in that they enable or disable possible actions of the observer. Only the variables of the controller and the environment are used in the enabling conditions of the observer's actions. Specifying the observer that way prevents the user from

```
TLSPEC Observer
 USING Definitions
 DATA IN panel_obs:OnOff_t;
         leak_sensor:bool;
         gasflow:gas_t;
         timer:nat

      SHARED INOUT who:schedule_t

      OUT panel_out:OnOff_t;
          leak_sens_out:bool;
          gasflow_out:gas_t;
          now, now_out:nat;
          x,t_obs:nat;
          state:state_t

      INTERNAL leak_sens_int:bool;
               gas_int:gas_t
 ACTIONS
 /*Condition for the initial phase*/
 init_cond :: = leak_sens = T AND leak_sens_int = T AND
                gasflow = run AND gas_int = run

 /*Condition for the leaking to non_leaking phase*/
 leak_to_non_leak ::= gasflow =blocked AND gas_int = run AND
                      leak_sens = F AND leak_sens_int = T

 /*Condition for the non_leaking to leaking phase*/
 non_leak_to_leak ::= gasflow =run AND leak_sens = T AND
                      leak_sens_int = F

  /*Special case for initial phase*/
 obs1 ::= state' = leaking AND
          unchanged(x, ...,gas_int, now, now_out)

 /*leaking to non_leaking action*/
 obs2 ::= state' = non_leaking AND x' = 0 AND t_obs' = t_obs AND
          now' = now AND now_out' = now_out AND gas_int' = gasflow
          AND leak_sens_int' = leak_sens
```

Figure 6.10: Specification of the Observer (part 1)

```
 /*non_leaking to leaking action*/
 obs3 ::= state' = leaking AND x' = 0 AND
          now' = now AND now_out' = now_out AND t_obs' = t_obs
          AND leak_sens_int' = T AND gas_int' = run
 /*Action describing the remaining in non_leaking state.
   Enabled if: not init_cond AND not leak_to_non_leak
   AND not non_leak_to_leak*/

 obs4 ::= state' = non_leaking AND
          x' = x + 1 AND
          now' = now + 1 AND
          now_out' = now_out + 1 AND
          t_obs' = t_obs AND
          leak_sens_int' = leak_sens AND
          gas_int' = gasflow

 /*Observer action with scheduling and setting of output
   variables*/
 obs_act ::= who = obs AND who' = contr AND
             gasflow_out' = gasflow AND
             panel_out' = panel_obs AND
             leak_sens_out' = leak_sensor AND
             IF init_cond
             THEN obs1
             ELSE IF leak_to_non_leak
                  THEN obs2
                  ELSE IF non_leak_to_leak
                       THEN obs3 ELSE obs4
                       FI
                  FI
             FI

 SPEC INITIAL who = obs AND panel_out = on AND
              leak_sens_out = T AND gasflow_out = run AND
              gas_int = run AND leak_sens_int = T
              AND now = 0 AND now_out = 0 AND x = 0
              AND t_obs = 0 AND state = leaking
      TRANSITIONS [obs_act] {now, gasflow_out, panel_out,
                             leak_sens_out, who, leak_sens_out}
TLSPECEND
```

Figure 6.11: Specification of the Observer (part 2)

rebuilding the hybrid gasburner in the observer component what would result in an inadequate observation of the controller and the environment.

All variables of the observer are initialised by the `INITIAL` predicate in the `SPEC` slot. The initialisation of the variables used in the refinement mapping (`x`, `state`, `now` and `t_obs`) has to fit to the initial values of the variables of the hybrid gasburner so that the real gasburner starts in a leaking situation (`gasflow_out` = `run` and `leak_sens_out` = `T`).

The observer has to handle the following situations that might occur in the hybrid gasburner:

- The system is in its initial state and gas flows out of the nozzle without being burned.
- The system changes from leaking to non-leaking.
- The system remains non-leaking.
- The system changes from non-leaking to leaking.

To recognise these situations in the observer component it is not enough to have a look at the values of the input variables in the actual state. Just by looking at these variables the observer could not recognise whether there was a situation change from `leaking` to `non_leaking` or not. In order to detect all these situation changes the observer stores parts of the previous observation in internal variables (`gas_int` and `leak_sens_int`). With the help of these internal variables the situation is unambiguously recognisable. They are used in the enabling conditions for the actions `obs1` through `obs4` and they are defined in the actions

- `init_cond`,
- `leak_to_non_leak` and
- `non_leak_to_leak`

as described in Figure 6.10.

Let us have a more detailed look at one of the possible situations. Assume that the condition `leak_to_non_leak` (see Figure 6.10) is satisfied. This means that the actual values of the variables `gasflow` and `leak_sens` are `blocked` and `F` (false), respectively. Moreover, assume that the previous observations of the observer were that there was a running gas flow and a leak, i.e. `gas_int` is `run` and `leak_sens_int` is `T`. This situation indicates that the system happened to be in a leaking situation and changed to a non-leaking situation. The observer reacts on this change by setting the `state` variable to `non_leak` and by resetting the variable `x` to `0`. This corresponds exactly to the behaviour of the hybrid system in such a situation. That way the behaviour of the hybrid gasburner is filtered out of the behaviour of the controller and the environment.

6.3.6 The Gasburner as VSE-II Specification

As usual we illustrate hybrid automata as annotated graphs. The hybrid gasburner is pictured in Figure 6.12: The meaning of the variables of the hybrid gasburner in

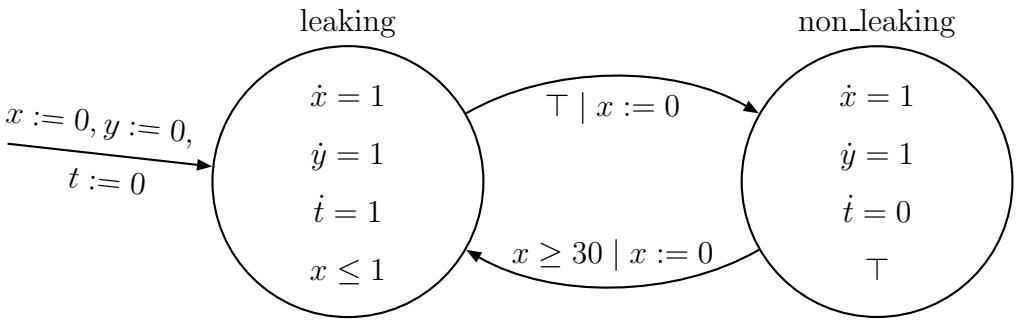

Figure 6.12: Gasburner Hybrid Automaton

Figure 6.12 is as follows. y accumulates overall time. x represents a control clock that guarantees that the system remains for at most 1 time unit within location `leaking` and for at least 30 time units within location `non_leaking`. t counts leakage time, i.e. the amount of time the system resides within `leaking`.

The translation of the hybrid gasburner (see [77] and Section 5) then results in the VSE-II gasburner specification:

```
TLSPEC gasburner
 PURPOSE  "Specification of the gasburner. "
 USING definition
 DATA OUT x, y, t : nat
      OUT state : state_t
 ACTIONS
 phi_1 ::= state = leaking AND state' = non_leaking AND
           x' = 0 AND UNCHANGED(y, t)

 phi_2 ::= state = non_leaking AND state' = leaking AND
           x >= 30 * c AND x' <= c AND x' = 0 AND
           UNCHANGED(y, t)

 phi_1_star ::= state = leaking AND state' = non_leaking AND
                NOT x + 1 <= c AND x' = 0 AND
                UNCHANGED(y, t)

 psi_1 ::= state = leaking AND state' = leaking AND
           x + 1 <= c AND y' = y + 1 AND
           x' = x + 1 AND t' = t + 1
```

```
 psi_2 ::= state = non_leaking AND state' = non_leaking AND
           x' = x + 1 AND y' = y + 1 AND UNCHANGED(t)

 SPEC INITIAL x = 0 AND y = 0 AND
              t = 0 AND state = leaking
      TRANSITIONS [phi_1, phi_2, phi_1_star, psi_1, psi_2]
                  {x, y, t, state}
      FAIRNESS WF(phi_1_star) {x, y, t, state},
               WF(psi_1) {x, y, t, state},
               WF(psi_2) {x, y, t, state}

 SATISFIES Safety
TLSPECEND
```

This VSE-II gasburner specification constitutes the `gasprop` TLSPEC given in Figure 6.3 and represents a complex real-time property we want to prove from the `gasburner`.

6.3.7 The Behaviour of the Real Gasburner

This section consists mainly of the table in Figure 6.13 which represents a possible behaviour of the real gasburner (see the upper part of the table) including the mapping done by the observer component and the refinement mapping[12]. The table shows how the observer component works and, therefore, how it maps the behaviour of the real gasburner to the hybrid gasburner (see the lower part of Table 6.13). The proof given in Section 6.3.9 finally states that every behaviour of the real gasburner is mapped to a behaviour of the hybrid gasburner in a way that the refinement relation can be established. For readability we use the abbreviations *n_l* for `non_leaking` and *bl* for `blocked` in Table 6.13. If we just consider the lower part we can see that this is a possible behaviour of the hybrid gasburner.

6.3.8 Handling of the Constant c

The constant c, as introduced in Sections 5 and 4.2, plays a special role in the example presented here. It is used to get an exact discretisation of the continuous behaviour of the original hybrid gasburner. In the specification of the controller (see Figure 6.8) c is used in the actions `A5` and `A6`.

The role of the natural constant c is the same as in the translation of hybrid automata in Section 5. It frees the system from choosing a time unit. Let us assume, for example, the value of c is 10. Then the clock has to make 300 steps,

[12]The refinement mapping is shown in the development graph of the real gasburner in Figure 6.3. It consists of a mapping of the variables to be implemented to terms constructed from the variables of the implementing system.

	init	obs	contr	env	obs	contr	env	obs	contr	env	obs	...
cstate	on	on	off	off	off	off	off	off	of	off	off	...
timer	0	0	0	0	0	0	0	0	0	0	0	...
gasflow	run	run	bl	bl	bl	bl	bl	bl	bl	bl	bl	...
panel	on	on	on	on	on	on	on	on	on	on	on	...
leak_sens	T	T	T	F	F	F	F	F	F	F	F	...
now	0	0	0	0	0	0	0	1	1	1	2	...
gas_int	run	run	run	run	bl	bl	bl	bl	bl	bl	bl	...
leak_sens_int	T	T	T	T	F	F	F	F	F	F	F	...

y	0	0	0	0	0	0	0	1	1	1	2	...
x	0	0	0	0	0	0	0	1	1	1	2	...
t	0	0	0	0	0	0	0	0	0	0	0	...
state	l	l	l	l	n_l	n_l	n_l	n_l	n_l	n_l	n_l	...

Figure 6.13: Example Behaviour with Refinement Mapping

before the barrier is reached. If, however, c is 1000 the clock has to tick 30.000 times to reach the barrier. If we consider seconds as the basic time unit, then steps take $\frac{1}{10}$ of a second or a millisecond, respectively.

However, the real proof of the (safety) properties has to be done independently from specific values of the constant c. Therefore, it is valid for all possible integer values and thus for all granularities (even the infinitesimal).

6.3.9 Refinement Proof

We sketch the refinement proof indicated in Figure 6.3 by a VSE-II `satisfies`-link. The proof is done locally to the observer component. During the proof one immediately realises that assumptions about the behaviour of the environment of the observer are needed. These assumptions deal with two different proof situations. First, we have to know whether we are in a leaking or a non-leaking state. We insert this knowledge by the following invariants into the proof:

$$\Box(intern(observer) \wedge init\_cond \rightarrow state = leak)$$
$$\Box(intern(observer) \wedge leak\_non\_leak \rightarrow state = leak)$$
$$\Box(intern(observer) \wedge non\_leak\_leak \rightarrow state = non\_leak)$$
$$\Box(intern(observer) \wedge non\_leak\_non\_leak \rightarrow state = non\_leak)$$

The definitions of *init_cond*, *leak_non_leak*, *non_leak_leak* and *non_leak_non_leak* are given below.

$$
\begin{aligned}
init\_cond &\;\widehat{=}\; leak\_sens = T \wedge leak\_sens\_int = T \wedge gasflow = run \\
&\quad \wedge gas\_int = run \\
leak\_non\_leak &\;\widehat{=}\; gasflow = bl \wedge gas\_int = run \wedge leak\_sens = F \\
&\quad \wedge leak\_sens\_int = T \\
non\_leak\_leak &\;\widehat{=}\; leak\_sens = T \wedge leak\_sens\_int = F \wedge gasflow = run \\
non\_leak\_non\_leak &\;\widehat{=}\; \neg(init\_cond \vee leak\_non\_leak \vee non\_leak\_leak)
\end{aligned}
$$

The formula $intern(observer)$ represents the internal behaviour of the observer component, i.e. the behaviour of the observer without the hiding quantification.

In the second proof we need knowledge about the behaviour of the controller in the `non_leak_leak` situation. Again this knowledge is inserted with the following assumptions:

$$
\begin{aligned}
&\Box((leak\_sens = T \wedge leak\_sens\_int = F \wedge gasflow = run) \\
&\quad \rightarrow now \geq timer + 30 * c) \\
&\Box((leak\_sens = T \wedge leak\_sens\_int = F \wedge gasflow = run) \rightarrow x = now - timer
\end{aligned}
$$

It is evident that the proofs of these assumptions need knowledge about the controller component as well as about the scheduling. Finally, the proof is performed locally to the observer component and is exported as a lemma to the global proof obligation.

6.3.10 Summary

We have presented a methodology (observer models) for formal requirements engineering. Its applicability is illustrated with the help of a realistic gasburner.

One of the open issues is how to refine a specification without doing the whole proof work again. This problem seems to be very similar to that of a refinement in the security area, for example in protocol analysis.

6.4 Storm Surge Barrier

Another application scenario we have presented in Section 4.2.8 is the Storm Surge Barrier. We have shown there how this real-time system can be specified and how special real-time properties can be specified and verified within VSE-II. In this section we present how the observer methodology can be applied to the Storm Surge Barrier as well. We sketch how the real-time properties of the Storm Surge Barrier, or more precisely the real-time properties of its control system, can be specified using hybrid automata as a real-time observation description. One of the

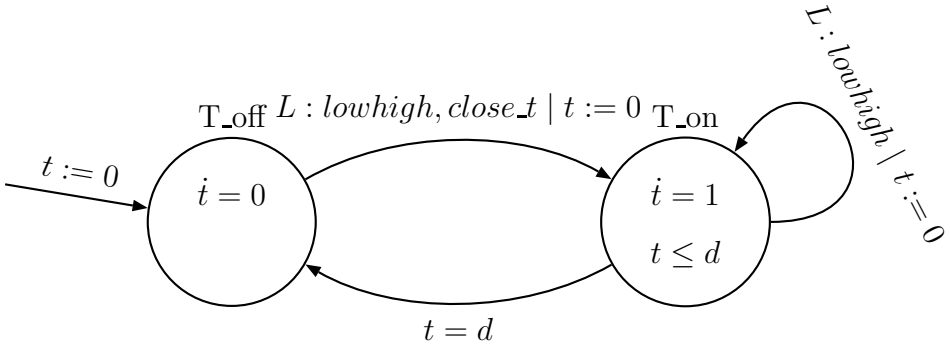

Figure 6.14: Hybrid Automaton for the Timer

real-time constraints in the description of the SVKO in Section 4.2.8 (see also the specification in Appendix A) was that a timer awakes at a certain signal and it is cumulatively re-enabled, i.e. if the signal is sent repeatedly within the time period the timer is running, then the whole time period the timer is running is extended by this new amount of time. This behaviour is specified in the VSE-II specification given in Appendix A in the `TLSPEC SVKO_system` component.

This property can be specified using a hybrid automata which is a composition of the hybrid automata given in Figures 6.14 and 6.15.

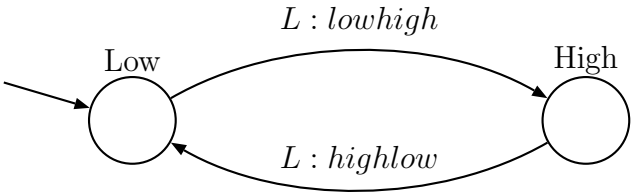

Figure 6.15: Hybrid Automaton for Environment

The automaton in Figure 6.14 describes the behaviour of the timer itself. If the automaton is in state `T_off` this means that the timer is off and it waits for a trigger from the environment, i.e., an event of the environment that in our case expresses that the sensors measure a critical waterlevel. The trigger is delivered by the automaton shown in Figure 6.15. This automaton represents a very abstract version

of the behaviour of the environment. The state `low` means that the waterlevel is not critical. In case where the environment delivers a `lowhigh` trigger this means that the waterlevel is critical and the automaton changes location from `Low` to `High`. This behaviour enforces the automaton from Figure 6.14 to change location from location `T_off` to `T_on`. Within this action the variable `t` representing the timer is reset to `0`.The timer is switched off, if `t` reaches `d` where the automaton switches again to `T_off`.

The before mentioned cumulative behaviour is specified by the transition that has location `T_on` as both source and target. This action is only enabled if the environment sends a `lowhigh` signal that says that there is another change from `Low` to `High` caused by the environment. The timer `t` is reset and the system control stays again for at least `d` time units in location `T_on`.

Finally, we give an automaton that describes the behaviour for the closing of the gates. The automaton shown in Figure 6.16 has two locations, one for the open state and one for the close state.

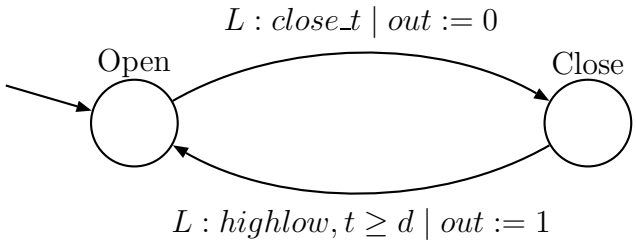

Figure 6.16: Hybrid Automaton for Open Close Signal

The location change from `Open` to `Close` is triggered by the timer automaton via the label close_t. The signal is again set to open if the environment changes from high to low and the timer `t` has at least advanced until `d`.

The behaviour of the environment is by no means restricted just as it was the case in the VSE-II specification. Thus, we can guarantee that the behaviour of the specified system reacts correctly to arbitrary changes of the environment. That way we do also catch the changes of the waterlevels as they might really occur and treat them correctly.

The real-time behaviour of the control system is built by the composition of the automata from Figures 6.15, 6.14 and 6.16. Recall that, in order to apply the observer methodology we had to specify an observer mapping and show that the specification of the Storm Surge Barrier from Section 4.2.8 satisfies (implements) the π-translation of the composed automaton.

7

Other Methods

The expert reader is no doubt aware of the contemporary literature about other formal methods that deal with the specification and/or verification of real-time systems. In fact, there are just too many approaches to list and describe them completely here. Our choice, however, seems to be representative for most of them and in the following we shall present the Duration Calculus, Timed CSP, and Timed Automata.

As usual, each of these description methods has its advantages and shortcomings. For example, if the system under consideration can best be described in terms of how long certain situations last rather than when they occur, the Duration Calculus might be the best choice. If, on the other hand, the system can best be described in terms of successive events, Timed CSP might simplify matters.

Interestingly, all these methods do not contradict the observer methodology proposed in this thesis. On the contrary, the general ideas that led to the embedding of Hybrid Automata into the VSE-II framework equally apply to the Duration Calculus, Timed CSP, and, in particular and most easily, Timed Automata.

In this chapter we shall briefly describe all three approaches, and emphasise both the differences and the common grounds with the approach proposed in this thesis.

7.1 Duration Calculus

The duration calculus extends interval temporal logic with assertions about the duration of states, there is no absolute time [33]. We present the duration calculus

by its syntax and semantics and revisit the gasburner example in order to show how real-time systems can be specified and how properties can be verified in this calculus.

7.1.1 Syntax of the Duration Calculus

The syntax of the duration calculus is introduced in three steps. We first give the syntax of *state expressions* followed by the syntax of *terms*. After that we present the syntax of *formulae*.

State expressions are defined by

$$s ::= 0 \mid 1 \mid v \mid \neg s \mid s_1 \vee s_2$$

where 0 and 1 are boolean constants representing *true* and *false* respectively. v stands for a state variable with $v \in V_s$ and V is an infinite set of variables which consists of a set V_s of state variables and a set V_g of global variables.

A *term* t is defined as

$$t ::= c \mid x \mid \int s$$

where c represents a constant, $x \in V_g$ (x is time independent) and s is a state expression.

A *formula* F is defined by

$$F ::= true \mid t_1 \circ t_2 \mid \neg F \mid F_1 \vee F_2 \mid F_1 ^\frown F_2 \mid \forall x.F$$

where $x \in V_g$, $\circ \in \{=, <, >, \leq, \geq\} = R$ is a set of binary relations and "$^\frown$" relates two formulas in subsequent intervals.

7.1.2 Semantics of the Duration Calculus

The semantics of the Duration Calculus is given with respect to three possible time domains in the literature [33, 32]: $\mathbb{N}$, $\mathbb{Q}^+$ and $\mathbb{R}^+$, for discrete, rational and continuous time respectively. In this context an $\mathbb{N}$-interval is defined as a (bounded, closed) interval whose endpoints b and e are natural numbers. A real interval is defined by $\{r \in \mathbb{R}^+ \mid \ b \leq r \leq e\}$. In the following we denote by *time* one of the time domains $\mathbb{N}$, $\mathbb{Q}^+$ or $\mathbb{R}^+$.

The semantics of state expressions is defined by the interpretation I^s which relates state expressions to (total) functions with domain $\mathbb{R}^+$ and codomain $\{0, 1\}$:

$$
\begin{aligned}
I^s(0) &= 0_{\mathbb{R}^+} \\
I^s(1) &= 1_{\mathbb{R}^+} \\
I^s(v) &= I^s_v : \mathbb{R}^+ \to \{0_{\mathbb{R}^+}, 1_{\mathbb{R}^+}\} \text{ with } v \in V_s \\
I^s(\neg s_1) &= \bar{\neg} I^s_{s_1} \text{ with } I^s_{s_1} : \mathbb{R}^+ \to \{0_{\mathbb{R}^+}, 1_{\mathbb{R}^+}\} \text{ and } s_1 \text{ is a state} \\
I^s(s_1 \vee s_2) &= I^s_{s_1} \bar{\vee} I^s_{s_2} \text{ where } s_1 \text{ and } s_2 \text{ are states}
\end{aligned}
$$

The operators $\bar{\neg}$ and $\bar{\vee}$ are defined by:

$$
\begin{aligned}
&\bar{\neg} : \{0,1\} \to \{0,1\} \\
&\bar{\neg}(0) = 1 \\
&\bar{\neg}(1) = 0 \\
&\bar{\vee} : \{0,1\}, \{0,1\} \to \{0,1\} \\
&a \bar{\vee} b = \left\{ \begin{array}{l} 1 \text{ if } a = 1 \text{ or } b = 1 \\ 0 \text{ if } a = 0 \text{ and } b = 0 \end{array} \right\}
\end{aligned}
$$

The difference between $\vee$ and $\bar{\vee}$ at the semantical level was made explicit, but for clarity we omit this distinction in the following for readability. From the context it will be clear which one is meant.

The semantics of a *term* t in an interpretation I is a function I^t defined by $I^t : term, Val, \mathbb{R}, \mathbb{R} \to \mathbb{R}$, where *Val* is a set of valuations and a valuation $\nu \in Val$ associates a value $\nu(x) \in time$ to every variable $x \in V_g$ in an observation interval $[b, e]$. I^t is defined with respect to a valuation $\nu \in Val$ as follows:

$$
\begin{aligned}
I^t(c, \nu, b, e) &= \bar{c} \text{ with } \bar{c} \in time \\
I^t(\textstyle\int s_1, \nu, b, e) &= \textstyle\int_b^e I^s(s_1)(y)dy \text{ with } b,\, e \in time \text{ and } s_1 \text{ a state} \\
I^t(x, \nu, b, e) &= \nu(x), \text{ where } x \in V_g.
\end{aligned}
$$

The semantics of a constant c is defined as a value $\bar{c} \in time$. The semantics of global variables is defined using a valuation ν. The most interesting part in the semantic definition is that of the term $\int s_1$. It is defined using the semantical definition of states and the (Riemann/Lebesgue) integral . Integrals on the semantical side always have their given limits. We write $\int_b^e f(t)dt$ for a Riemann/Lebesgue integral.

The last step is the definition of the semantics of formulae. The semantics of a formula F in an interpretation I is defined by an interpretation function I^f with valuation $\nu \in Val$ as:

$$
\begin{array}{lcl}
I^f(true, \nu, b, e) & = & T \\
I^f(t_1 \circ t_2, \nu, b, e) & = & T \textbf{ iff } I^t(t_1, \nu, b, e) \odot I^t(t_2, \nu, b, e) \text{ holds where} \\
 & & t_1, t_2 \text{ are terms, } \circ \in R \text{ and } \odot \text{ is the corresponding} \\
 & & \text{relation to } \circ \text{ on } time \\
I^t(x, \nu, b, e) & = & \nu(x), \text{ where } x \in V_g \\
I^f(\neg F, \nu, b, e) & = & T \textbf{ iff } I^f(F, \nu, b, e) = F \\
I^f(F_1 \vee F_2, \nu, b, e) & = & T \textbf{ iff } I^f(F_1, \nu, b, e) = T \text{ or } I^f(F_2, \nu, b, e) = T \\
I^f(F_1 \frown F_2, \nu, b, e) & = & T \textbf{ iff for some } m \in time \text{ with } m \in [b, e]: \\
 & & I^f(F_1, \nu, b, m) = T \text{ and } I^f(F_2, \nu, m, e) = T \\
I^f(\forall x F[x], \nu, b, e) & = & T \textbf{ iff } I^f(F[x], \hat{\nu}, b, e) = T \text{ for all } \hat{\nu} \ x\text{-equivalent} \\
 & & \text{to } \nu.
\end{array}
$$

DEFINITION 7.1.1
Two valuations ν and $\hat{\nu}$ are x-equivalent if and only if for all $y \neq x$ $\nu(y) = \hat{\nu}(y)$.

Now we can define truth, validity and satisfiability.

DEFINITION 7.1.2
A formula F is true in the interpretation I ($I \models F$) if and only if $I^f(F, \nu, b, e) = T$ for every valuation $\nu \in$ Val.

DEFINITION 7.1.3
A formula F is valid ($\models F$) if and only if $I \models F$ holds for every interpretation I.

DEFINITION 7.1.4
A formula F is satisfiable if and only if there exists an interpretation I and a valuation ν such that $I^f(F, \nu, b, e) = T$. If $I^f(F, \nu, b, e) = T$ then we say that F holds for the interpretation I^f in the interval $[b, e]$.

After these rather standard definitions we present some abbreviations that are frequently used:

$$\lceil P \rceil \ \widehat{=} \ \int P = \int 1 \wedge \neg \int 1 = \int 0 \tag{7.1}$$

$$\lceil\rceil \ \widehat{=} \ \neg \lceil 1 \rceil \tag{7.2}$$

$$\Diamond F \ \widehat{=} \ true \frown F \frown true \tag{7.3}$$

$$\Box F \ \widehat{=} \ \neg \Diamond \neg F \tag{7.4}$$

The formula 7.1 means that P is lifted from a state expression to a formula. Informally it means that P holds almost everywhere on a non-point interval. The

expression "almost everywhere" indicates that there could be a finite number of exceptions in the interval where P does not hold.

If the formula 7.2 holds on an interval, then we know that this interval is a point interval. With the abbreviation given in 7.1 the formula $\lceil\rceil$ reduces to $\int 1 = \int 0$.

The formula 7.3 says that F holds for some subinterval whereas the formula 7.4 expresses that F holds in every subinterval. Note that if we write $\Box\lceil P\rceil$, then this does not mean that P holds on every point on a non-point interval. It means that $\lceil P\rceil$ holds on every subinterval and that P holds almost everywhere in every non-point subinterval.

Before specifying and verifying the gasburner example presented in Section 7.1.3, we list some axioms and give an induction rule (introduced in [33]) that are used in the example.

Axiom 7.1.1

$$\int 0 = 0$$

Axiom 7.1.2 *For an arbitrary state P :*

$$\int P \geq 0$$

Axiom 7.1.3 *For arbitrary states P and Q:*

$$\int P + \int Q = \int P \vee Q + \int P \wedge Q$$

Axiom 7.1.4 *Let P be a state and r, s non-negative reals.*

$$\left(\int P = r + s\right) \Leftrightarrow \left(\int P = r\right)^\frown\left(\int P = s\right)$$

The following induction rule extends a hypothesis over adjacent subintervals.

Rule 7.1.5 (Induction Rule) *Let X denote a formula occurring in the formula $R(X)$ and let P be a state. If $R(\lceil\rceil)$ holds and $R(X \vee X^\frown\lceil P\rceil \vee X^\frown\lceil\neg P\rceil)$ is provable from $R(X)$, then $R(true)$ holds.*

The induction rule relies on the finite variability property of states and on the finitude of the intervals: any interval can be split into a finite alternation of states P and $\neg P$.

The difficulties in applying the induction rule essentially come with defining the formula $R(X)$. An example demonstrating how the Duration Calculus can be used for the specification and verification of a system is given in the next chapter.

7.1.3 The Gasburner Example in the Duration Calculus

The requirements and the design decisions we have defined in Section 3.1 (item 1 to item 4), where item 1 and item 2 describe the safety property expressed by the following formula:

$$\int 1 \geq 60sec \Rightarrow 20 \int Leak \leq \int 1 \qquad \text{(Safe)}$$

Expressing that a leak is stoppable within one second, is formulated by the formula:

$$\Box(\lceil Leak \rceil \Rightarrow \int 1 \leq 1sec \qquad \text{(Des-1)}$$

Finally, to guarantee that the system will remain non_leaking for at least 30 seconds is represented as

$$\Box[(\Diamond\lceil Leak\rceil)^\frown(\Diamond\lceil \neg Leak\rceil)^\frown(\Diamond\lceil Leak\rceil) \Rightarrow \int 1 \geq 30sec] \qquad \text{(Des-2)}$$

Before we present the proof sketch summarised from [33], some remarks on the specification style in the duration calculus seem opportune:

- The specification is not constructive in the sense that the specification describes what the properties of the system are and not how the system fulfils these properties.
- As the example shows, in-constructive specifications of systems can be very short but concise.
- The coding of the requirement and of the design decisions is relatively straightforward.

For proving the property Safe in [33] two lemmata are used.

LEMMA 7.1.6 *Des-1* $\wedge$ *Des-2* $\Rightarrow \Box(((\lceil Leak\rceil \wedge \int 1 \leq 0.5sec)^\frown\lceil \neg Leak\rceil^\frown\lceil Leak\rceil) \Rightarrow 20 \int Leak \leq \int 1)$

LEMMA 7.1.7 $\lceil Leak\rceil \Rightarrow \lceil Leak\rceil^\frown(\lceil Leak\rceil \wedge \int 1 \leq 0.5sec)$

We do not go into more detail concerning the proofs of the lemmata. They are given in [33]. The main difficulty with these lemmata is not their proof but their invention. As mentioned earlier in this work, finding invariants is a non-trivial task in a verification/proof process.

The main proof of Des-1 $\wedge$ Des-2$\wedge \int 1 \geq 60sec \Rightarrow 20 \int Leak \leq \int 1$ is sketched in [33]. It uses the fact that for a state P $(true^\frown\lceil P\rceil) \vee (true^\frown\lceil \neg P\rceil) \vee \lceil\rceil$ holds. This lemma divides the proof into three cases. The $\lceil\rceil$ case is trivial because the

premise $\int 1 \geq 60sec$ is not satisfied. The two other cases are proven with the induction rule defining $R(X)$ by the predicate:

$$
\begin{aligned}
&(\text{Des-1} \wedge \text{Des-2} \wedge \textstyle\int 1 \geq 60sec) \Rightarrow \\
&((X \frown \lceil Leak \rceil \Rightarrow 20 \textstyle\int Leak \leq \int 1) \wedge \\
&\ (X \frown \lceil \neg Leak \rceil \Rightarrow 20 \textstyle\int Leak \leq \int 1) \wedge \\
&\ (X \frown \lceil \neg Leak \rceil \Rightarrow (\lceil \neg Leak \rceil \vee \\
&\qquad\qquad (20 \textstyle\int leak \leq \int 1) \frown (\lceil leak \rceil \wedge \int 1 \leq 0.5sec) \frown \lceil \neg Leak \rceil)))
\end{aligned}
$$

The proof is far from being straightforward since there are several critical points to decide on:

- Finding and proving the right lemmata (invariants).
- Defining the right $R(X)$ for the application of the induction rule.

More elaborate versions of the proof of the property mentioned above using the RAISE justification tool [96] for the verification show the complexity of the proof when it is done in detail. In the version presented in [96] several lemmata are used to simplify the proving process and the main proof is several pages long.

The basic time elements in the Duration Calculus are intervals over $\mathbb{R}$. The relations between these basic time elements are equality, betweenness, precedence or adjacent, for example.

There are many other issues concerning the Duration Calculus that are not treated in this section. One such research topic is to analyse decidability results for the Duration Calculus. The reader who is interested in this field is referred to [32].

7.2 Timed CSP

7.2.1 Introduction to Timed CSP

In this chapter we give a short introduction to Timed CSP (Timed Communicating Sequential Processes). It is mainly a summary of [92] and [93]. The language of Timed CSP is a simple extension of Hoare's Communicating Sequential Processes. The language of CSP was designed for describing systems of interacting components, and is supported by an underlying theory for reasoning about them. It is an abstract language to describe communication patterns of concurrent system components that interact through message passing.

The conceptual framework taken by CSP is to consider system components as processes. They are independent self-contained entities with particular interfaces through which they interact with their environment.

Before we describe the language and a semantics of timed CSP, we first introduce the notation used in untimed CSP including definitions of events and processes.

7.2.1.1 Untimed CSP

Systems are modelled in terms of the events they can perform. The set of all possible events is denoted by Σ. Events are atomic communications between an object and its environment. The word process is used to denote the behaviour patterns of a system component. Processes are described in terms of possible events that they may be engaged in.

- The process $STOP$ is the process that can engage in no events at all (deadlock).
- The output $c!v \rightarrow P$ first performs $c.v$, the output of v on channel c. Hereafter it behaves as P.
- The input $c?x : T \rightarrow P(x)$ can accept any input of type T on channel c. Its first event will be any event of the form $c.t$ with $t \in T$. After that it behaves as $P(t)$.
- The process $P \Box Q$ (P choice Q) can behave as P or as Q. Its possible communications are those of P and those of Q.
- The process $P \sqcap Q$ is able to behave non-deterministically either as P or as Q.
- Processes may also be composed in parallel. If D is a set of events, then the process $P \parallel [D] \parallel Q$ behaves as P and Q acting concurrently with the request that they have to synchronise on any event in the synchronisation set D;

events which are not in D may be performed by either process independently of the other. $P \parallel Q$ is equivalent to $P \parallel [\{\}] \parallel Q$.

- The expression $\mu X.F(X)$ is used to denote the unique fixed point of the semantic domain mapping represented by F[1].

Processes may be recursively defined by means of equational definitions. Process names must appear on the left hand side of such definitions, and CSP expressions which may include those names appear on the right hand side.

EXAMPLE 7.2.1 $LIGHT = on \rightarrow off \rightarrow LIGHT$

This defines a process $LIGHT$ whose only possible behaviour is to perform *on* and *off* alternately. Mutually recursive processes may also be defined where a (possible infinite) collection of process names X_k appear on the left hand side of definitions, and CSP expressions $F_k(\vec{X})$ possibly involving any of those names appear on the right. For a set of recursive definitions to be a mutual recursion, each name appearing in any of the process bodies must be bound in one of the recursive definitions.

EXAMPLE 7.2.2

$$\begin{aligned} COUNT_0 &= up \rightarrow COUNT_1 \\ COUNT_{n+1} &= (up \rightarrow COUNT_{n+2} \Box (down \rightarrow COUNT_n) \end{aligned}$$

This defines a collection of processes; $COUNT_0$ can do a number of *up* and *down* events but can never do more *down*'s than *up*'s.

7.2.1.2 Semantics of Untimed CSP

The semantics of a process P is defined to be the set of sequences of events ($traces(P)$) that it may possibly perform. Examples of traces include

- $\langle\rangle$, the empty trace, which is possible for any process and
- $\langle on, off, on \rangle$ which is a possible trace of $LIGHT$.

There are many definitions of operators on traces. An incomplete selection is the following:

DEFINITION 7.2.1 (PROJECTION)
If D is a set of events, then the trace $tr \lceil D$ is defined to be the maximal subsequence of tr all of whose events are drawn from D.

[1] Only recursions that admit such fixed points are allowed.

Definition 7.2.2 (Message Extraction)
Message extraction $tr \downarrow C$, where tr is a trace and C a set of channel names, provides the maximal sequence of messages passed on channel C.

Definition 7.2.3 (Event Extraction)
If tr is a sequence, then $\sigma(tr)$ is the set of events appearing in the sequence tr. The operator σ naturally extends to processes: $\sigma(P)$ is the set of events that appear in some trace of P.

With the notion of traces of a process P we are able to define a refinement relation between two processes.

Definition 7.2.4 (Refinement)
Let P and Q be processes, then if $traces(P) \subseteq traces(Q)$ then we say that Q is a refinement of P, written $P \sqsubseteq Q$.

With the help of the semantics we define what it means for a specification to satisfy a property.

Definition 7.2.5 (*sat*)
Specifications or properties are given as predicates on traces, and a process P satisfies a specification S if all of its traces satisfy S:

$$P \, sat \, S \Leftrightarrow \forall tr \in traces(P).S$$

Using the definition of *sat* we can define proof rules (a proof system) associated with the traces model for CSP. We present only one proof rule here since it is often possible to give more specialised proof rules for special application areas. This amounts to developing a specialised theory and proof system.

For each operator there is a proof rule. For example, the rule for the prefix operator is:

$$\frac{P \, sat \, S(tr)}{a \rightarrow P \, sat \, (tr = \langle\rangle \vee (tr = \langle a \rangle \hat{} tr' \wedge S(tr')))}$$

This proof rule says that if the hypotheses holds that the process P satisfies the specification S on trace tr of P, then we can deduce that the process $a \rightarrow P$ satisfies the specification $tr = \langle\rangle$, which is a property of every process, or it satisfies the property $tr = \langle a \rangle \hat{} tr' \wedge S(tr')$ which says that the trace tr starts with the event a and $S(tr')$ holds where $\langle a \rangle \hat{} tr' = tr$.

At this point the proof rules to analyse processes can be built as done in [91]. In this paper a specific theory to analyse authentication protocols is built on top of the general CSP semantic framework presented before.

Since we are interested in the specification and verification of real-time systems we now present the language and the semantics of timed CSP.

7.2.1.3 The Language of Timed CSP

The language of timed CSP [92] is a simple extension of untimed CSP. It consists of several process constructors including primitives for parallel composition, nondeterministic choice and hiding.

Syntax In timed CSP each of the untimed CSP operators is interpreted in a timed context, and two timing operators are added. The syntax of TCSP terms is given by the following BNF rule:

$$
\begin{array}{rlllll}
P ::= & STOP & SKIP & WAIT & a \xrightarrow{t} P & P;\ P \\
 & P \Box P & P \sqcup P & P \stackrel{t}{\rhd} P & P \,{}_A\|_A\, P & P \,|||\, P \\
 & F(P) & P \setminus A & \mu X.F(X) & &
\end{array}
$$

In the above rule, event a is taken from the set of all synchronisations Σ. The event set A ranges over the set of subsets of Σ, and t is a non-negative real number. There is no lower bound on the interval between consecutive events. This allows one to model asynchronous processes in a satisfactory fashion without artificial constructs on the traces at which independent events may be observed.

In the following we will explain the syntactical constructs of TCSP in more detail[2].

The terms $STOP$, $SKIP$ and $\surd$ have the same meaning in TCSP as in untimed CSP.

- The process $WAIT_t$ is a delayed $SKIP$. It represents a process which does nothing except terminate successfully after time t.
- The process $a \xrightarrow{t} P$ will behave as P precisely t time units after synchronisation event a is observed.
- The sequential composition operator provides a means of transferring control on termination. In the construct $P;\ Q$, control is passed from P to Q if and when P performs the termination event $\surd$.
- The event $\surd$ is not visible to the environment, and occurs as soon as it becomes available.
- The external choice $P \,\Box\, Q$ may be resolved by the environment.
- The environment has no influence on an internal choice $P \sqcap Q$. The outcome of such a choice is non-deterministic.

[2]There will be some explanations repeated and some will be new. All will be mentioned for the sake of completeness.

- The timeout operator $P \trianglerighteq Q$ transfers control from P to Q if no communications occur before time t. If an attempt at communication involving P is made at time t precisely, then the outcome will be nondeterministic. If either of the components should terminate, then the entire timeout construct terminates immediately.

- In the construct $P \,{}_A\|_B\, Q$ term P may perform events in A, term Q may perform only those events in B, and the two terms must cooperate on events drawn from the intersection of A and B.

- The asynchronous parallel combinator, $|||$, allows both components to evolve concurrently without interacting.

- The hiding operator provides a mechanism for abstraction in TCSP. The term $P \backslash A$ behaves as P except that events in a are concealed from the environment. Concealed events no longer require the cooperation of the environment, and so occur as soon as P is ready to perform them[3].

- The relabelled term $f(P)$ has a similar control structure to term P, with observable events renamed according to function f.

In untimed CSP, the term $a \rightarrow P$ models a system which is initially prepared to engage in event a, and then eventually behaves as P. To model real-time systems we must be able to model constraints upon the time between the observation of an a and the onset of P.

The relation between prefix and delay is:

$$a \xrightarrow{t_1+t_2} P \equiv a \xrightarrow{t_1} WAIT_{t_2};\ P$$

Example 7.2.3 *The formalisation of a process P that makes event b available exactly four seconds after event a is observed, only to halt two seconds after b is performed, is:*

$$P = a \xrightarrow{4} b \xrightarrow{2} STOP$$

Events are considered to be instantaneous. If the duration of an action is of interest, then that action may be modelled by specifying the beginning and the end of the action as separate events.

[3]There is a slight different explanation of the hiding operator in the literature. For example, in [49] $P \setminus C$ is a process which behaves like P, except that each occurrence of any event in C is concealed which means that the alphabet of $P \setminus C$ is the alphabet of P without the alphabet of C.

7.2.1.4 Semantics of Timed CSP

The language of CSP considers processes as interacting system components. Although processes may have differing internal behaviours, they are solely analysed in terms of their external behaviour, i.e. only activities at the interface can have any effect on a context in which processes might be placed. In particular, if two processes cannot be distinguished within any CSP environment, then they should be considered equivalent.

In the timed world, contexts can be more sensitive to the behaviour of their components than in the untimed world. It is not only important which events a component is able to perform or to refuse, but also the time point at which this happens. Thus, components can also be distinguished by their respective timing behaviour.

For example, the process $Q_1 = a \rightarrow STOP$ and $Q_2 = WAIT_2;\ a \rightarrow STOP$ are both able to perform the event a, and both processes will have the same traces. However, the times at which the a is on offer are different, so the processes should be distinguished on the basis of their timed behaviour. If placed in parallel with the time-sensitive process $(a \rightarrow STOP) \overset{1}{\triangleright} STOP$ the first resulting system is able to perform the event a but the second is not. A timed semantics must differentiate Q_1 from Q_2.

Another example shows two processes that cannot be distinguished by any environment:

$$(WAIT_2;\ a \rightarrow STOP) \sqcap (WAIT_2;\ b \rightarrow STOP)$$

and

$$WAIT_2;\ ((a \rightarrow STOP) \sqcap (b \rightarrow STOP))$$

Although both processes have different executions, the first process resolves the internal choice at time 0 whereas the second process does not resolve the internal choice until time 2, there is no environment that can distinguish between the two processes.

A semantics for timed processes that makes the required distinctions is described for example in [92, 93]. In [93] a compositional model is described that considers processes as sets of *timed failures* where a timed failure is a record of an execution, consisting of a *timed trace* containing informations about the performed events, and a *timed refusal* containing information about when events could be refused. Timed failures information captures those aspects of a process's behaviour that can be observed by interacting with it.

We omit the exact definition of the semantics of Timed-CSP[4], instead we give an example that shows how real-time systems can be specified using Timed-CSP.

[4]The interested reader is refered to [92, 93].

7.2.1.5 Gasburner in Timed-CSP

The specification of the gasburner in this section is taken from [43]. In this work a method is presented how to use Z [94] and Timed-CSP for the specification of software for safety-critical applications. The formal specification of a system is structured in a *dynamic* and a *functional* part. In the dynamic part the reactive behaviour of a component is specified. Real-time requirements and the ordering of events are handled within this part using Timed-CSP. In the functional part the internal behaviour is specified. It is mainly concerned with the structure of the possible system states and system operations that are defined as relations between inputs, outputs and the system state before and after the execution of an operation. Z is used to handle the functional part of the specification.

In the description of the example we mainly refer to the dynamic part, the Timed-CSP part, of the specification of the gasburner. The architecture of the gasburner described in [43] is as follows.

- The gas actuator controls the emission of gas and receives the commands to start or stop the emission of gas from the controller.
- The ignition actuator starts or stops the ignition of escaping gas from the nozzle of the gasburner.

Both actuators can be triggered by the controller at arbitrary time instants. The decisions of the controller are based on observations made by the two sensors, the thermometer and thermostat sensor. The thermometer sensor measures the temperature and reports it to the controller. The thermostat sensor signals the controller whether there is a request by the user to activate or deactivate the gasburner.

In the following description we will concentrate on the dynamic behaviour of the gasburner and we will only mention the functional behaviour without a formal definition as it is given in [43] using Z. Furthermore, only the controller part of the gasburner is described. The environment is represented by the actuators and the sensors in the model as mentioned above.

The dynamic behaviour of the gasburner controller is given by the real-time CSP process $GasBurnerControl$.

$$\begin{aligned} GasBurnerControl \;\widehat{=}\; & GasBurnerInit \rightarrow \\ & (GasBurnerControl_{Ready} \parallel \\ & Timer1 \parallel \\ & Timer2) \end{aligned}$$

The process $GasBurnerControl$ starts with the initialisation represented by the event $GasBurnerInit$. This event expresses that the gasburner is in an idle mode

where there is no gas flow and no flame. The event $GasBurnerInit$ is followed by the three parallel processes $GasBurnerControl_{Ready}$, $Timer1$ and $Timer2$.

In order to define the $GasBurnerControl_{Ready}$ process the fixpoint operator μ is used where $\mu X.F(X)$ is the least fixed point of the function $F(X)$. The μ-operator allows recursive processes to be defined without the need to name them.

$$
\begin{array}{l}
GasBurnerControl_{Ready} \mathrel{\widehat{=}} \mu X. \\
\quad HighPriority \\
\Box \\
\quad heat\_on\_request \rightarrow Waite; \ \text{if } BurnerIsDeactivated \\
\qquad \text{then } set\_timer1 \rightarrow HeatOnRequestExecution \rightarrow ActuatorCtr;\ X \\
\qquad \text{else } ShutDownExecution \rightarrow ActuatorCtr;\ Stop\ fi \\
\Box \\
\quad flame\_on \rightarrow (HighPriority \rhd \text{ if } IgitionIsActivated \\
\qquad \text{then } reset\_timers \rightarrow IgnitionOKExecution \rightarrow ActuatorCtr;\ X \\
\qquad \text{else } ShutDownExecution \rightarrow ActuatorCtr;\ Stop \text{ fi}) \\
\Box \\
\quad flame\_off \rightarrow (HighPriority \rhd \text{ if } FlamePresent \\
\qquad \text{then } FlameFailureExecution \rightarrow ActuatorCtr;\ X \\
\qquad \text{else } ShutDownExecution \rightarrow ActuatorCtr;\ Stop \text{ fi}) \\
\Box \\
\quad timer1\_elapsed \rightarrow (HighPriority \rhd \\
\qquad set\_timer2 \rightarrow IgnitionExecution \rightarrow ActuatorCtr;\ X) \\
\Box \\
\qquad timer2\_elapsed \rightarrow (HighPriority \rhd \\
\qquad\quad IgnitionFailureExecution \rightarrow ActuatorCtr;\ X)
\end{array}
$$

In the description of $GasBurnerControl_{Ready}$ there are several processes and events that are explained informally as follows:

- *heat_on_request* (*heat_off_request*) marks a situation in which there is a request from outside to start (stop) the controller.
- $flame\_off$ describes a state in which there is no flame.

- $timer1\_elapsed$ ($timer2\_elapsed$) marks a situation in which $timer1$ ($timer2$) is elapsed.
- $BurnerIsDeactivated$ marks a situation in which the controller of the gasburner is not active.
- $IgnitionIsActivated$ represents a situation in which the ignition is activated and can be started.
- $FlamePresent$ describes a state in which the flame of the gasburner is burning.
- set_timer1 sets $timer1$ to a specific value.
- $reset\_timers$ resets the timer at a given point in time.
- $HeatOnRequestExecution$ marks the beginning of an request to start the gasburner.
- $IgnitionOKExecution$ reflects a situation in which ignition is executed successfully.
- $IgnitionExecution$ marks the beginning of an ignition phase.
- $IgnitionFailureExecution$ describes a situation in which there is an ignition failure.
- $FlameFailureExecution$ reflects a behaviour in which there should be no flame since there has been a $flame\_off$ event.
- *ShutDownExecution*: This event represents the shut down of the controller.
- *ActuatorCtr*: A process that communicates with the physical real gasburner. It sends the commands of the controller to the actuators.
- $HighPriority$: The process that specifies the behaviour of the controller in case that the event $heat\_off\_request$ occurs. It is defined as follows:

$$
\begin{aligned}
&HighPriority \mathrel{\widehat{=}} \\
&\quad heat\_off\_request \rightarrow Wait_{\epsilon}; \\
&\qquad \text{if } \neg\, BurnerIsDeactivated \\
&\qquad \text{then } reset\_timers \rightarrow HeatOffRequestExecution \rightarrow \\
&\qquad\quad ActuatorCtr;\ GasBurnerControl_{Ready} \\
&\qquad \text{else } ShutDownExecution \rightarrow ActuatorCtr;\ Stop \text{ fi}
\end{aligned}
$$

The process definition of the controller is recursive. This represents the cyclic behaviour of the controller that continuously reacts to incoming events from the sensors. The two timer components consist of a *set_timer* event, followed by a wait process and a *timer_elapsed* event. These processes can be interrupted by *reset_timers* at any time.

In this version of the specification of the gasburner there are two classes of events that can occur simultaneously: high priority and low priority events. If two such events happen simultaneously[5], then only the high priority event is treated. This is expressed with the help of the timeout operator $\triangleright^{\leq}$. The event *heat_on_request* is neither a high nor a low priority event.

A property of this definition of a gasburner that is manually proven in [43] is that in each interval of 30 seconds there may be gas leaking from the nozzle of the gasburner at most $2 + 2\epsilon$ seconds, where ϵ is the response time for the technical components.

[5]In this model this means with a time difference of at most ϵ.

7.3 Automata

Since we use Hybrid Automata (see section 4.3) as one of the basic formalisms in this work we give a short overview of a common definition of timed automata and argue that hybrid automata can be viewed as an extension of timed automata by relaxing some of the syntactic constraints. Therefore, all the theoretical results developed in Sections 5 and 6 can also be applied to timed automata.

We start by giving a very common definition of timed automata and explain the behaviour of such an automaton by a small example. After that, we present the version of timed automata used in the UPPAAL tool together with a short description of the tool itself.

7.3.1 Timed Automata

Timed automata generalise finite automata by additional constraints so that real-time systems can be handled. They were proposed as an abstract model for real-time systems in [36, 17].

A timed automaton operates with a finite control, i.e. a finite set of locations and a finite set of real-valued clocks. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were started (or reset). Each transition of the automaton may reset some of the clocks. Each location of the automaton puts certain constraints on the values of the atomic propositions as well as on the values of the clocks. The control of the automaton can reside in a particular location only if the values of the propositions and clocks satisfy the corresponding constraints.

A formal definition of a timed automaton is as follows.

Definition 7.3.1 (timed automata)
A timed automaton A is defined by an eight-tuple $A = (P, L, L_0, C, \mu, \nu, E, F)$ where the elements of the tuple have the following meaning:

- *P is a finite set of propositions.*
- *L is a finite set of locations.*
- *L_0 is a set of initial locations with $L_0 \subseteq L$.*
- *C is a finite set of clocks.*
- *A mapping μ that associates with each location in L a boolean formula over the set P of propositions. If $l \in L$, then the formula $\mu(l)$ is called a* propositional constraint.
- *A labelling function ν that associates with each location in L a* timing constraint *over the variables in C. Each timing constraint is a boolean combination of atomic timing constraints, i.e. comparisons between terms involving*

clock variables and primitive operations as, for instance, addition by constants.

- *A set $E \subseteq L \times L \times \mathcal{P}(C)$ of transitions where each transition (edge) (l, l', λ) consists of a source location l, a target location l' and a set $\lambda \subseteq C$ of clocks to be reset. $\mathcal{P}(\mathcal{C})$ denotes the power set of C.*
- *A family $F \subseteq \mathcal{P}(C)$ the acceptance set of locations.*

A run of a timed automata is defined as a timed state sequence. At any time instant during a run, the configuration of the automaton is completely determined by the location in which the control resides and by the values of all propositions and all clocks. The values of the clocks are given by a *clock interpretation.*

DEFINITION 7.3.2 (CLOCK INTERPRETATION)
The clock interpretation is a mapping $\gamma : C \to \mathbb{R}$ from the set of clocks to the real numbers. For any clock $x \in C$ the value of x under the interpretation γ is the non-negative real number $\gamma(x)$.

DEFINITION 7.3.3 (STATE)
A state of the timed automaton A is a triple (l, σ, γ), where $l \in L$ is a location and

- *$\sigma \subseteq P$ is an observation that satisfies the propositional constraint $\mu(l)$.*
- *γ is a clock interpretation that satisfies the timing constraint $\nu(l)$.*

Before we present the formal definition of a run, first some intuition: Suppose that at time $t \in \mathbb{R}$ a timed automaton is in state (l, σ, λ). Assume further that the location l of the automaton and the observation σ remain unchanged during the time interval I[6] with $l(I) = t$. We know that all clocks proceed with the same rate as time elapses. The value of a clock x at time $t' \in I$ is $\gamma(x) + t' - t$. During this interval the clock values satisfy the timing constraint that is associated with l:

$$(\gamma(x) + t' - T) \models \nu(l)$$

If the automaton changes its location at time $r(I) = t''$ along the edge (l, l', λ), then this change can happen in two ways.

1. I is right closed: The state at time t'' is $(l, \sigma, \gamma + t'' - t)$.
2. I is right open: In this case the state at time t'' is (l', σ', γ'), where σ' is an observation satisfying $\mu(l')$ and the clock interpretation γ' is defined by
   (a) $\gamma'(x) = 0$ for all clocks to be reset, i.e. for all clocks x with $x \in \lambda$, and
   (b) $\gamma'(x) = \gamma(x) + t'' - t$ for all other clocks.

[6] $l(I)$ ($r(I)$) denotes the left (right) end-point of the interval I.

Definition 7.3.4 (run)
A run of the automaton $A = (P, L, L_0, C, \mu, \nu, E, F)$ *is a finite or infinite sequence*

$$r\colon \mapsto_{\gamma_0} r_1 \mapsto^{\lambda_1}_{\gamma_1} r_2 \mapsto^{\lambda_2}_{\gamma_2} r_3 \mapsto^{\lambda_3}_{\gamma_3} \cdots$$

with triples $r_i = (l_i, \sigma_i, I_i)$ *where* l_i *are locations* $l_i \in L$, σ_i *are observations* $\sigma_i \subseteq P$, I_i *are intervals,* λ_i *are clock sets* $\lambda_i \subseteq C$ *and* γ_i *are clock interpretations* $\gamma_i : C \to \mathbb{R}$ *such that*

- $l_0 \in L_0$*;*
- $(r_i, r_{i+1}, \lambda_i) \in E$ *for all* $i \geq 0$*;*
- σ_i *satisfies* $\mu(l_i)$ *for all* $i \geq 0$*;*
- $I = I_0 I_1 I_2 \ldots$ *is an interval sequence;*
- *for all* $x \in C$ *and* $i \geq 0$, $\gamma_{i+1}(x) = 0$ *if* $x \in \lambda_{i+1}$, *and* $\gamma_{i+1}(x) = \gamma_i(x) + r(I_i) - l(I_i)$ *otherwise;*
- $\gamma_i + t - l(I_i)$ *satisfies* $\nu(l_i)$ *for all* $i \geq 0$ *and* $t \in I_i$*;*
- *either* $I = I_0 I_1 I_2 \ldots$ *is finite and* $\{l_n\} \in F$*, or* r *is infinite and* $\{l \mid l = l_i \text{ for infinitely many } i \geq 0\} \in F$

Every run r of the timed automata A uniquely determines a timed state sequence τ_r.

As a first example of timed automata we give an automaton that satisfies the stimulus response property as described in Section 2.2.2.2.

The automaton in Figure 7.1 has the locations l_0, ..., l_4. The initial location is l_0 indicated by an arrow without a given source. The automaton starts in location l_0 where p and q do not hold. If the stimulus p occurs the automaton changes to location l_1 and sets the clock x to 0. The automaton is not allowed to stay in location l_1 because of the condition $x = 0$ and changes immediately to location l_2. The clock x is used to measure the time that elapses since the time of the stimulus. In location l_2 the automaton decides within one time unit to go back to location l_0 or to go to location l_3. This decision is taken nondeterministically. Going back to location l_0 means not to respond, whereas going to location l_3 means to respond the stimulus and that within 2 time units after the stimulus occurred. The automaton changes to location l_0 and waits for another stimulus.

Unfortunately, the gasburner automaton given in Figure 4.19 in Section 4.3 cannot be expressed by the version of timed automata as presented here, because stop watches or guards on transitions describing state changes cannot be expressed.

In the next section we present another version of timed automata used in the UPPAAL tool.

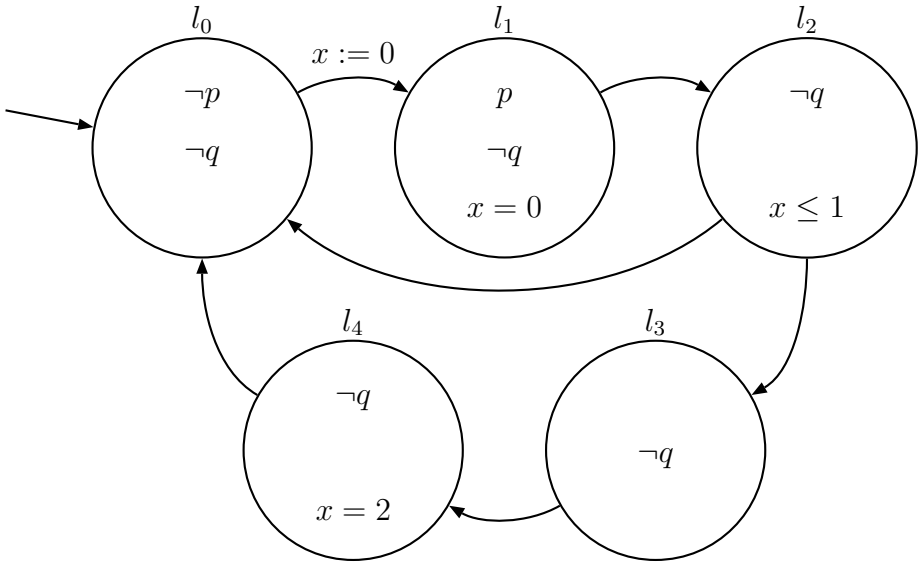

Figure 7.1: Stimulus Response Automaton

7.3.2 UPPAAL Timed Automata

In this section we sketch the version of timed automata used in the UPPAAL tool [29, 28]. In UPPAAL finite-state automata extended with clock and data variables are used to describe processes and networks of such automata. UPPAAL timed automata are very similar to the timed automata described before except for some slight differences concerning the composition and synchronisation techniques.

7.3.2.1 Syntax of UPPAAL Timed Automata

Alur and Dill developed the theory of timed automata [17] as an extension of classical finite-state automata with clock variables. In a finite-state automaton, a transition has the form $l \xrightarrow{\alpha} l'$ meaning that the process modelled by the automaton will perform an α-transition in state l and reach state l'. Alur and Dill [17] extended the un-timed transition to the timed version $l \xrightarrow{\alpha,g,\Phi} l'$, where g is a simple linear constraint over the clock variables and Φ is a set of clocks. Intuitively, this means that a process in location l may perform the α-transition instantaneously when g is true and reach location l' after having reset the clocks in Φ. g is called the guard of the transition.

Assume a finite set of clock variables C denoted by x, y or z and a finite set of data variables V denoted by i, j or k. We use $G(C, V)$ to denote the set of formulae ranged over by g with the following syntax: $g ::= a \mid g \wedge g$ where a is either a timing constraint over C or a data constraint over V. A timing constraint is in the form $x \circ n$ or $x - x' \circ n$ where n is a natural number and $\circ \in \{\leq, \geq, =, <, >\}$. A data constraint is of the same form, i.e. $i \circ j$ or $i - i' \circ j$ where i,j may be integer.

$G(C, V)$ is called a *guard*.

Reset-Operations Clock and data variables are manipulated by a reset operation. Since the timed automata used in UPPAAL are distinguished in clock variables and data variables, there are two reset operations: A reset of a clock variable has the form $x := n$ where n is a natural number and a reset operation on an integer variable (data variable) should be in the form: $i := c * i + c_0$ where c and c_0 are integer constants. Note that c and c_0 can be zero or negative. We use R to denote the set of all possible reset operations.

Channel and Synchronisation We assume that processes synchronise with each other via channels. We further assume that each communication synchronises two processes (handshake). Let A be a set of channel names. We use $Act = \{\alpha? \mid \alpha \in A\} \cup \{\alpha! \mid \alpha \in A\}$ to denote the set of actions that processes can perform to synchronise with each other.

Committed Locations Communication between processes is one-to-one, i.e. only two processes can be synchronised. In this way broadcasting can not be implemented since the atomicity of a broadcast will be lost if it is implemented with several one-to-one synchronisations. To clarify the situation assume that a sender S wants to broadcast a message m to two receivers $R1$ and $R2$. As mentioned before this requires synchronisation between three processes and cannot be expressed directly in the timed automata used in UPPAAL, where synchronisation, as in CCS [75], is between two processes based on complementarity of actions. However, a broadcast can be implemented using the notion of a committed location. Committed locations are locations that must be left immediately. In this way S can first synchronise with R_1 on m_1 and then with R_2 on m_2 and the intermediate location S_2 of S that starts in location S_1 and ends up in location S_3 is marked as committed location.

Formal Syntax Definition In the following we give a formal syntax definition of timed automata with constraints and committed locations used in UPPAAL.

Definition 7.3.5 (Syntax)
An automaton A over actions Act with clock variables C and data variables V is a tuple $\langle L, l_0, E, C, I\rangle$ where

1. *L is a finite set of locations,*
2. *$L_C \subseteq L$ is the set of committed locations,*
3. *l_0 is the initial location,*

4. *$E \subseteq L \times G(C,V) \times Act \times \mathcal{P}(R) \times L$ is the set of transitions between locations*[7]*, and*

5. *I is a function that assigns to each location l a clock constraint $I(l)$. This constraint is called* invariant *of l and has the form $x \circ n$ where $\circ \in \{<, \leq\}$.*

A transition from the location l to location l' with guard g, action a to be performed and a set r of variable reset operations is represented by $l \xrightarrow{g,a,r} l'$

7.3.2.2 Informal Semantics

Informally, a process modelled by an automaton starts at location l_0 where all the clocks and data variables are initialised to 0. The values of the clocks increase synchronously with the time at location l_0. If the current values of the clocks satisfy the enabling condition g, the process can change the location by following an edge $l \xrightarrow{g,a,r} l'$. Executing this transition the variables are updated by r.

To be more precise about the semantics of UPPAAL timed automata we would have to introduce such notions as *runs* (see section 4.3) or *trajectories* [80]. These definitions are very similar to the ones given in Section 4.3. We omit them and proceed with an example. The reader interested in a formal definition of the semantics is referred to [80, 29].

7.3.2.3 Example Timed Automaton

As in the case of the version of timed automata presented in Section 7.3.1 we are not able to express the gasburner presented in Section 4.3 because in the timed automata used in UPPAAL there are no stop watches. One could think now that timed automata are only able to express very simple systems but that is not true. It is surprising how complex systems can be expressed and proved using the timed automata version of UPPAAL. Of course, there has to be always a compromise between expressiveness of the specification language and the proof support available for the language. Since it is not a main concern to investigate the expressiveness of timed automata in this work, we only refer to a result using timed automata and UPPAAL. In terms of complexity the Philips Audio-Control Protocol with bus-collision is a comprehensive case-study to which UPPAAL has been successfully applied. More information about the modelling or the protocol itself can be found in [57].

7.3.2.4 Conclusion

The timed automata we have presented in this section are all covered by the approach presented in Section 5, i.e. our examinations concerning linear hybrid automata are also applicable to timed automata. We use linear hybrid automata

[7] $\mathcal{P}(R)$ represents the powerset of R.

since they are more general and in some cases more natural (adequate) to express real-time system behaviours.

Nevertheless, there are translation techniques from linear hybrid automata to timed automata [80]. But these techniques are limited to Non-Zero Constant Slope Hybrid Systems ($\text{CSHS}_{\neq 0}$).

8

Conclusion and Future Work

In our society, much of the quality of life is tied to the quality and reliability of the information and computing systems we use. Highly complex computer systems are now to be found in almost every component of our infrastructure: government, education, health care, entertainment, e-commerce, manufacturing, telecommunications, transport, aerospace, defence, hazardous environments, and energy. Such systems depend intrinsically on the software that controls them, and so our quality of life and in some cases, even life itself, depends on the reliability of software and the computer. Modern computing systems typically run on distributed, heterogeneous networks and are subject to complex constraints for their functionality and performance. Although progress has been made in sound development methods, no single software development technique is adequate to address *all* issues of complex system development.

All engineering disciplines use mathematics and mathematical models to describe and analyse the behaviour of their devices and systems. Using formal methods in software (system) development is the consequent next step now in computer science as well. This way, formal software development becomes an *engineering discipline*.

The first step is done: The available methods are mature and have shown their applicability in many industrial applications. The next step is to provide methodologies to ease the use of formal methods and to integrate different approaches

in order to exploit their respective advantages. The resulting synergetic effect allows for the separation of different aspects of a system and for the use of *adequate* description means.

In this work we presented a methodology (observer models), based on the very general approach taken by the VSE-II system, to relate different views of the same system based on one specification that covers all these views. The methodology separates different views on that system and allows for different specification techniques or tools to realize such a view. Since all the views relate to one common specification that is given in VSE-II and since all the views are embedded into VSE-II the consistency among these views is assured. Think for example of a telecommunication system or an e-commerce system. There are protocols controlling the communication with real-time constraints on answers in that protocol, and databases held in the background that are updated according to the events that have taken place. Our methodology supports specialised description techniques for databases as well as for protocols. The methodology was applied to a real-time scenario. We have embedded Hybrid Automata into VSE-II in a way that both tools, VSE-II and for example HyTech, have advantages from this marriage. VSE-II with its interactive deduction system is now also connected directly to model checking approaches. Automatically computed results can be shipped from the model checking shore to the VSE-II shore. These automatically computed results can be used in the interactive proofs as lemmata. Vice versa, the developed technique allows for an integration of VSE-II lemmata into the model checking approach. Imagine a situation where the model checker does not terminate and imagine further that we have an idea of an invariant that helps the model checker to find the proof. Then we can switch to the VSE-II side and prove this invariant with the interactive deduction unit. When we are back in the model checker we are allowed to use this invariant as an invariant for the model checker because of the developed theory.

To achieve this for the specification of real-time systems using Hybrid Automata we have developed a discretisation technique for Hybrid Automata that allows for an exact discretisation. This means that we do not lose any property by translating such an automaton to his discretised version in VSE-II. This translation technique is not only usable for the translation of Hybrid Automata to VSE-II. The discretisation technique is general enough so that it can be applied to a wide range of other hybrid description techniques.

We have shown how real-time systems can be specified and verified in (pure) VSE-II using a general description technique for real-time systems. Several examples such as the familiar gasburner are used to illustrate this methodology.

Our main aim is to give the system engineer a methodology at hand that s/he can start to work without being confronted with the whole world of formal methods. This goal is achieved by the *Observer Model* which can be used for a smooth entrance into the world of formal methods starting with relatively easy applications and description techniques and going on to more complicated ones using more elaborate techniques, if necessary.

Future work will concentrate on branching time semantics. Our Property Specification Language (PSL) can express the usual linear temporal properties, but in Hybrid Systems it is common to have a branching time semantics, such that we can describe counter examples which is not possible in linear time semantics. Another extension of VSE-II would be to have automata (Hybrid Automata) as components in a VSE-II development graph at the same level as temporal logic specifications (TLSPEC's). Problems in this context are for example: How can we define the composition of a TLSPEC and a Hybrid Automaton? How can we prove properties of Hybrid Automata in the presence of assumptions about the environment?

A

Specification of the SVKO

In this chapter we have listed the specification of the Emergency Closing System (SVKO) presented in Chapter 4.2.8. The specification is available in the VSE-II tool and is identical (up to some minor improvements for readability) to this presentation. The graphical representation, i.e. the development graph of the SVKO, is given in Chapter 4.2.8. Working on the specification means usually manipulating the development graph and the specifications in it using a syntax-oriented editor. Users used to the tool can also specify a system in an arbitrary editor and then import these specifications into the development graph.

Before we present the specification first some remarks. The names in the specification are rather long because first we print the specification with the full qualified names of the variables. In the VSE tool there is the possibility to produce online a short view on the variables where this part is removed. But in order to be unambiguous and to avoid problems in reading the specification we decided to have the full qualified version. The second reason is that we take the names from the original description. We tried to stay as close to the original names and functionalities of these variables as possible to make it easier for the engineers to understand the specification[1].

[1] Everyone who has already specified a non-trivial system knows that a good naming of variables is not trivial and important for better understanding.

```
1. BasicDatas
=============

THEORY BasicDatas
 PURPOSE
 "basic operations for the hardwired protection system"
 USING natural;
       boolean
 TYPES zustand =
       GENERATED BY s1 |
                    s2 |
                    s3
 FUNCTIONS
           /* global setting */
           /* time durations _|- and -|_ resp. : */
           down, up : nat;
           /* allowed signal limits */
           max_val, min_val : nat;
           minus : nat, nat -> nat;
           /* initial setting of the environment variables */
           IWL_OWL_const : nat;
           IWL_TRIGGER_HIGH1_const, IWL_TRIGGER_HIGH2_const,
           IWL_TRIGGER_HIGH3_const : nat;
           IWL_TRIGGER_LOW1_const, IWL_TRIGGER_LOW2_const,
           IWL_TRIGGER_LOW3_const : nat;
           OWL_TRIGGER1_const, OWL_TRIGGER2_const,
           OWL_TRIGGER3_const : nat;
           OPEN_DIFFERENCE1_const, OPEN_DIFFERENCE2_const,
           OPEN_DIFFERENCE3_const : nat
 PREDICATES less_comp : nat, nat;
            voter2from3 : bool, bool, bool
 VARS n1, n2 : nat;
      b1, b2, b3 : bool
 AXIOMS FOR max_val : max_val > min_val + 1
        FOR min_val : min_val < max_val - 1 AND
                      min_val > 0
        FOR less_comp : ALL n1, n2 :
                        less_comp(n1,n2) <->
                        /* B > A */
                        n1 < n2 AND
                        n2 > min_val AND
                        n2 < max_val AND
                        n1 > min_val AND
```

```
                    n1 < max_val
        FOR voter2from3 : ALL b1, b2, b3 :
                          voter2from3(b1,b2,b3) <->
                          /* 2/3 */
                          (b1 = t AND
                           b2 = t) OR
                          (b2 = t AND
                           b3 = t) OR
                          (b1 = t AND
                           b3 = t)
        FOR minus : ALL n1, n2 :
                    IF n1 >= max_val OR
                       n1 <= min_val OR
                       n2 >= max_val OR
                       n2 <= min_val OR
                       n2 > n1
                    THEN minus(n1,n2) = 0
                    /* signal value 0 < min_val and is
                       not valid */
                    ELSE minus(n1,n2) = n1 - n2
                    FI
        FOR IWL_OWL_const :
                           /* not_CLOSE must be TRUE in INIT state,
                              therefore: */
                           IWL_OWL_const < OWL_TRIGGER1_const AND
                           IWL_OWL_const < OWL_TRIGGER2_const AND
                           IWL_OWL_const < OWL_TRIGGER3_const
THEORYEND

2. SVKO_combine
===============

TLSPEC SVKO_combine
 PURPOSE
 " combine of the SVKO components"
 USING Basicdatas
 DATA OUT
          /* output of the system time.*/
          time_out : nat;
          /* general output datas */
          not_CLOSE_out, OPEN_out, OPEN_ALLOWED_out : bool;
          /* environment datas */
          IWL_GEM1_out, IWL_GEM2_out, IWL_GEM3_out : nat;
```

```
          IWL_TRIGGER_HIGH1_out, IWL_TRIGGER_HIGH2_out,
          IWL_TRIGGER_HIGH3_out : nat;
          IWL_TRIGGER_LOW1_out, IWL_TRIGGER_LOW2_out,
          IWL_TRIGGER_LOW3_out : nat;
          OWL_GEM1_out, OWL_GEM2_out, OWL_GEM3_out : nat;
          OWL_TRIGGER1_out, OWL_TRIGGER2_out,
          OWL_TRIGGER3_out : nat;
          OPEN_DIFFERENCE1_out, OPEN_DIFFERENCE2_out,
          OPEN_DIFFERENCE3_out : nat;
          SELECT_HIGH_TRIGGER_LEVEL_out, FORBID_OPEN_out : bool
 COMBINE environment_data SHARED
          [EnEnv <- Update.EnEnv ,EnEnv <- SVKO_system.EnEnv];
         /* environment.out -> SVKO_system.in */
         SVKO_system
         [SVKO_system.IWL_GEM1_sys_in <-
            environment_data.IWL_GEM1_sys_out,
          SVKO_system.IWL_GEM2_sys_in <-
            environment_data.IWL_GEM2_sys_out,
          SVKO_system.IWL_GEM3_sys_in <-
            environment_data.IWL_GEM3_sys_out,
          SVKO_system.OWL_GEM1_sys_in <-
            environment_data.OWL_GEM1_sys_out,
          SVKO_system.OWL_GEM2_sys_in <-
            environment_data.OWL_GEM2_sys_out,
          SVKO_system.OWL_GEM3_sys_in <-
            environment_data.OWL_GEM3_sys_out,
          SVKO_system.IWL_TRIGGER_HIGH1_sys_in <-
            environment_data.IWL_TRIGGER_HIGH1_sys_out,
          SVKO_system.IWL_TRIGGER_HIGH2_sys_in <-
            environment_data.IWL_TRIGGER_HIGH2_sys_out,
          SVKO_system.IWL_TRIGGER_HIGH3_sys_in <-
            environment_data.IWL_TRIGGER_HIGH3_sys_out,
          SVKO_system.IWL_TRIGGER_LOW1_sys_in <-
            environment_data.IWL_TRIGGER_LOW1_sys_out,
          SVKO_system.IWL_TRIGGER_LOW2_sys_in <-
            environment_data.IWL_TRIGGER_LOW2_sys_out,
          SVKO_system.IWL_TRIGGER_LOW3_sys_in <-
            environment_data.IWL_TRIGGER_LOW3_sys_out,
          SVKO_system.OWL_TRIGGER1_sys_in <-
            environment_data.OWL_TRIGGER1_sys_out,
          SVKO_system.OWL_TRIGGER2_sys_in <-
            environment_data.OWL_TRIGGER2_sys_out,
          SVKO_system.OWL_TRIGGER3_sys_in <-
```

```
    environment_data.OWL_TRIGGER3_sys_out,
  SVKO_system.OPEN_DIFFERENCE1_sys_in <-
    environment_data.OPEN_DIFFERENCE1_sys_out,
  SVKO_system.OPEN_DIFFERENCE2_sys_in <-
    environment_data.OPEN_DIFFERENCE2_sys_out,
  SVKO_system.OPEN_DIFFERENCE3_sys_in <-
    environment_data.OPEN_DIFFERENCE3_sys_out,
  SVKO_system.SELECT_HIGH_TRIGGER_LEVEL_sys_in <-
    environment_data.SELECT_HIGH_TRIGGER_LEVEL_sys_out,
  SVKO_system.FORBID_OPEN_sys_in <-
    environment_data.FORBID_OPEN_sys_out,
  SVKO_system.time <- Update.time] SHARED
    [EnSys <- Update.EnSys,
     EnUpd <- Update.EnUpd,
     EnEnv <- Update.EnEnv ,
     EnEnv <- environment_data.EnEnv];

 /*environment.out -> Update.in,
   SVKO_system.out -> Update.in,
   Update.out -> SVKO_combine.in */
  Update [Update.time_comb -> SVKO_combine.time_out,
  Update.not_CLOSE_upd_in <- SVKO_system.not_CLOSE_sys_out,
  Update.OPEN_upd_in <- SVKO_system.OPEN_sys_out,
  Update.OPEN_ALLOWED_upd_in <-
    SVKO_system.OPEN_ALLOWED_sys_out,
  Update.IWL_GEM1_upd_in <-
    environment_data.IWL_GEM1_upd_out,
  Update.IWL_GEM2_upd_in <-
    environment_data.IWL_GEM2_upd_out,
  Update.IWL_GEM3_upd_in <-
    environment_data.IWL_GEM3_upd_out,
  Update.OWL_GEM1_upd_in <-
    environment_data.OWL_GEM1_upd_out,
  Update.OWL_GEM2_upd_in <-
    environment_data.OWL_GEM2_upd_out,
  Update.OWL_GEM3_upd_in <-
    environment_data.OWL_GEM3_upd_out,
  Update.IWL_TRIGGER_HIGH1_upd_in <-
    environment_data.IWL_TRIGGER_HIGH1_upd_out,
  Update.IWL_TRIGGER_HIGH2_upd_in <-
    environment_data.IWL_TRIGGER_HIGH2_upd_out,
  Update.IWL_TRIGGER_HIGH3_upd_in <-
    environment_data.IWL_TRIGGER_HIGH3_upd_out,
```

```
Update.IWL_TRIGGER_LOW1_upd_in <-
  environment_data.IWL_TRIGGER_LOW1_upd_out,
Update.IWL_TRIGGER_LOW2_upd_in <-
  environment_data.IWL_TRIGGER_LOW2_upd_out,
Update.IWL_TRIGGER_LOW3_upd_in <-
  environment_data.IWL_TRIGGER_LOW3_upd_out,
Update.OWL_TRIGGER1_upd_in <-
  environment_data.OWL_TRIGGER1_upd_out,
Update.OWL_TRIGGER2_upd_in <-
  environment_data.OWL_TRIGGER2_upd_out,
Update.OWL_TRIGGER3_upd_in <-
  environment_data.OWL_TRIGGER3_upd_out,
Update.OPEN_DIFFERENCE1_upd_in <-
  environment_data.OPEN_DIFFERENCE1_upd_out,
Update.OPEN_DIFFERENCE2_upd_in <-
  environment_data.OPEN_DIFFERENCE2_upd_out,
Update.OPEN_DIFFERENCE3_upd_in <-
  environment_data.OPEN_DIFFERENCE3_upd_out,
Update.SELECT_HIGH_TRIGGER_LEVEL_upd_in <-
  environment_data.SELECT_HIGH_TRIGGER_LEVEL_upd_out,
Update.FORBID_OPEN_upd_in <-
  environment_data.FORBID_OPEN_upd_out,
Update.not_CLOSE_upd_out ->
  SVKO_combine.not_CLOSE_out,
Update.OPEN_upd_out -> SVKO_combine.OPEN_out,
Update.OPEN_ALLOWED_upd_out ->
  SVKO_combine.OPEN_ALLOWED_out,
Update.IWL_GEM1_upd_out -> SVKO_combine.IWL_GEM1_out,
Update.IWL_GEM2_upd_out -> SVKO_combine.IWL_GEM2_out,
Update.IWL_GEM3_upd_out -> SVKO_combine.IWL_GEM3_out,
Update.OWL_GEM1_upd_out -> SVKO_combine.OWL_GEM1_out,
Update.OWL_GEM2_upd_out -> SVKO_combine.OWL_GEM2_out,
Update.OWL_GEM3_upd_out -> SVKO_combine.OWL_GEM3_out,
Update.IWL_TRIGGER_HIGH1_upd_out ->
  SVKO_combine.IWL_TRIGGER_HIGH1_out,
Update.IWL_TRIGGER_HIGH2_upd_out ->
  SVKO_combine.IWL_TRIGGER_HIGH2_out,
Update.IWL_TRIGGER_HIGH3_upd_out ->
  SVKO_combine.IWL_TRIGGER_HIGH3_out,
Update.IWL_TRIGGER_LOW1_upd_out ->
  SVKO_combine.IWL_TRIGGER_LOW1_out,
Update.IWL_TRIGGER_LOW2_upd_out ->
  SVKO_combine.IWL_TRIGGER_LOW2_out,
```

```
          Update.IWL_TRIGGER_LOW3_upd_out ->
            SVKO_combine.IWL_TRIGGER_LOW3_out,
          Update.OWL_TRIGGER1_upd_out ->
            SVKO_combine.OWL_TRIGGER1_out,
          Update.OWL_TRIGGER2_upd_out ->
            SVKO_combine.OWL_TRIGGER2_out,
          Update.OWL_TRIGGER3_upd_out ->
            SVKO_combine.OWL_TRIGGER3_out,
          Update.OPEN_DIFFERENCE1_upd_out ->
            SVKO_combine.OPEN_DIFFERENCE1_out,
          Update.OPEN_DIFFERENCE2_upd_out ->
            SVKO_combine.OPEN_DIFFERENCE2_out,
          Update.OPEN_DIFFERENCE3_upd_out ->
            SVKO_combine.OPEN_DIFFERENCE3_out,
          Update.SELECT_HIGH_TRIGGER_LEVEL_upd_out ->
            SVKO_combine.SELECT_HIGH_TRIGGER_LEVEL_out,
          Update.FORBID_OPEN_upd_out ->
            SVKO_combine.FORBID_OPEN_out] SHARED
                [EnSys <- SVKO_system.EnSys ,
                 EnUpd <- SVKO_system.EnUpd ,
                 EnEnv <- SVKO_system.EnEnv ,
                 EnEnv <- environment_data.EnEnv]
 SATISFIES SVKO_safety
TLSPECEND

3. SVKO_safety
==============

TLSPEC SVKO_safety
 PURPOSE
 " Specification of the safety model of the SVKO system."
 INCLUDE SVKO_inc = SVKO_combine
 VARS t0 : nat
 SPEC [] ((SVKO_inc.time_out = t0 AND
           WL_less_TL(SVKO_inc.IWL_GEM1_out,
           SVKO_inc.IWL_GEM2_out, SVKO_inc.IWL_GEM3_out,
           SVKO_inc.IWL_TRIGGER_HIGH1_out,
           SVKO_inc.IWL_TRIGGER_HIGH2_out,
           SVKO_inc.IWL_TRIGGER_HIGH3_out,
           SVKO_inc.IWL_TRIGGER_LOW1_out,
           SVKO_inc.IWL_TRIGGER_LOW2_out,
           SVKO_inc.IWL_TRIGGER_LOW3_out,
           SVKO_inc.OWL_GEM1_out,
```

```
            SVKO_inc.OWL_GEM2_out,
            SVKO_inc.OWL_GEM3_out,
            SVKO_inc.OWL_TRIGGER1_out,
            SVKO_inc.OWL_TRIGGER2_out,
            SVKO_inc.OWL_TRIGGER3_out,
            SVKO_inc.SELECT_HIGH_TRIGGER_LEVEL_out) = T) ->
           [] ((SVKO_inc.time_out = t0 + 1 AND
                WL_less_TL(SVKO_inc.IWL_GEM1_out,
                SVKO_inc.IWL_GEM2_out,
                SVKO_inc.IWL_GEM3_out,
                SVKO_inc.IWL_TRIGGER_HIGH1_out,
                SVKO_inc.IWL_TRIGGER_HIGH2_out,
                SVKO_inc.IWL_TRIGGER_HIGH3_out,
                SVKO_inc.IWL_TRIGGER_LOW1_out,
                SVKO_inc.IWL_TRIGGER_LOW2_out,
                SVKO_inc.IWL_TRIGGER_LOW3_out,
                SVKO_inc.OWL_GEM1_out,
                SVKO_inc.OWL_GEM2_out,
                SVKO_inc.OWL_GEM3_out,
                SVKO_inc.OWL_TRIGGER1_out,
                SVKO_inc.OWL_TRIGGER2_out,
                SVKO_inc.OWL_TRIGGER3_out,
                SVKO_inc.SELECT_HIGH_TRIGGER_LEVEL_out) = F) ->
                ((t0 + 1 < SVKO_inc.time_out AND
                  SVKO_inc.time_out < t0 + 1 + down) ->
                SVKO_inc.not_CLOSE_out = F)));
      [] (NOT (SVKO_inc.OPEN_out = t AND
                NOT (SVKO_inc.not_CLOSE_out = t)))
TLSPECEND

4. SVKO_system
==============

TLSPEC SVKO_system
 PURPOSE
 " SVKO system (determination of the not CLOSE
   and OPEN signals)"
 USING StatFunctions
 DATA OUT
          /*to update*/
          not_CLOSE_sys_out, OPEN_sys_out : bool;
          OPEN_ALLOWED_sys_out : bool
      INTERNAL timer_not_CLOSE, timer_OPEN : nat;
```

```
              not_CLOSE_TRIGGER_out, OPEN_TRIGGER_out,
              OPEN_PULS_out : bool;
              state : zustand
     IN
        /* from environment */
        IWL_GEM1_sys_in, IWL_GEM2_sys_in,
          IWL_GEM3_sys_in : nat;
        IWL_TRIGGER_HIGH1_sys_in, IWL_TRIGGER_HIGH2_sys_in,
          IWL_TRIGGER_HIGH3_sys_in : nat;
        IWL_TRIGGER_LOW1_sys_in, IWL_TRIGGER_LOW2_sys_in,
          IWL_TRIGGER_LOW3_sys_in : nat;
        OWL_GEM1_sys_in, OWL_GEM2_sys_in, OWL_GEM3_sys_in : nat;
        OWL_TRIGGER1_sys_in, OWL_TRIGGER2_sys_in,
          OWL_TRIGGER3_sys_in : nat;
        OPEN_DIFFERENCE1_sys_in, OPEN_DIFFERENCE2_sys_in,
          OPEN_DIFFERENCE3_sys_in : nat;
        SELECT_HIGH_TRIGGER_LEVEL_sys_in,
          FORBID_OPEN_sys_in : bool;
        time : nat
     SHARED INOUT
                  /* scheduling variables */
                  EnEnv, EnUpd, EnSys : bool
VARS IWL1_TRIGGER, IWL2_TRIGGER, IWL3_TRIGGER : bool;
     OWL1_TRIGGER, OWL2_TRIGGER, OWL3_TRIGGER : bool;
     OWL1_IWL1, OWL2_IWL2, OWL3_IWL3 : bool;
     IWL1_OWL1_OPEN_DIFFERENCE1, IWL2_OWL2_OPEN_DIFFERENCE2,
       IWL3_OWL3_OPEN_DIFFERENCE3 : bool;
     not_CLOSE_TRIGGER, not_CLOSE : bool;
     CONSIDER_OPEN, OPEN_TRIGGER, OPEN_PULS,
       OPEN_ALLOWED, OPEN : bool
ACTIONS
SVKO_Action ::= EX IWL1_TRIGGER, IWL2_TRIGGER, IWL3_TRIGGER,
                  OWL1_TRIGGER, OWL2_TRIGGER, OWL3_TRIGGER,
                  OWL1_IWL1, OWL2_IWL2, OWL3_IWL3,
                  IWL1_OWL1_OPEN_DIFFERENCE1,
                  IWL2_OWL2_OPEN_DIFFERENCE2,
                  IWL3_OWL3_OPEN_DIFFERENCE3,
                  not_CLOSE_TRIGGER, not_CLOSE,
                  CONSIDER_OPEN, OPEN_TRIGGER, OPEN_PULS,
                  OPEN_ALLOWED, OPEN :
                /* static computations */
                IWL1_TRIGGER =
                  IWL_TRIGGER_comparator(IWL_GEM1_sys_in,
```

```
          IWL_TRIGGER_HIGH1_sys_in,
          IWL_TRIGGER_LOW1_sys_in,
          SELECT_HIGH_TRIGGER_LEVEL_sys_in)
AND
IWL2_TRIGGER =
  IWL_TRIGGER_comparator(IWL_GEM2_sys_in,
          IWL_TRIGGER_HIGH2_sys_in,
          IWL_TRIGGER_LOW2_sys_in,
          SELECT_HIGH_TRIGGER_LEVEL_sys_in)
AND
IWL3_TRIGGER =
  IWL_TRIGGER_comparator(IWL_GEM3_sys_in,
          IWL_TRIGGER_HIGH3_sys_in,
          IWL_TRIGGER_LOW3_sys_in,
          SELECT_HIGH_TRIGGER_LEVEL_sys_in)
AND
OWL1_TRIGGER =
  OWL_TRIGGER_comparator(OWL_GEM1_sys_in,
                         OWL_TRIGGER1_sys_in)
AND
OWL2_TRIGGER =
  OWL_TRIGGER_comparator(OWL_GEM2_sys_in,
                         OWL_TRIGGER2_sys_in)
AND
OWL3_TRIGGER =
  OWL_TRIGGER_comparator(OWL_GEM3_sys_in,
                         OWL_TRIGGER3_sys_in)
AND
OWL1_IWL1 =
  OWL_IWL_comparator(OWL_GEM1_sys_in,
                     IWL_GEM1_sys_in)
AND
OWL2_IWL2 =
  OWL_IWL_comparator(OWL_GEM2_sys_in,
                     IWL_GEM2_sys_in)
AND
OWL3_IWL3 =
  OWL_IWL_comparator(OWL_GEM3_sys_in,
                     IWL_GEM3_sys_in)
AND
IWL1_OWL1_OPEN_DIFFERENCE1 =
  OPEN_DIFFERENCE_comparator(OWL_GEM1_sys_in,
          IWL_GEM1_sys_in,
```

```
          OPEN_DIFFERENCE1_sys_in)
AND
IWL2_OWL2_OPEN_DIFFERENCE2 =
  OPEN_DIFFERENCE_comparator(OWL_GEM2_sys_in,
          IWL_GEM2_sys_in,
          OPEN_DIFFERENCE2_sys_in)
AND
IWL3_OWL3_OPEN_DIFFERENCE3 =
  OPEN_DIFFERENCE_comparator(OWL_GEM3_sys_in,
          IWL_GEM3_sys_in,
          OPEN_DIFFERENCE3_sys_in)
AND
not_CLOSE_TRIGGER =
  input_puls_func(IWL1_TRIGGER,
                  IWL2_TRIGGER,
                  IWL3_TRIGGER,
                  OWL1_TRIGGER,
                  OWL2_TRIGGER,
                  OWL3_TRIGGER,
                  OWL1_IWL1,
                  OWL2_IWL2,
                  OWL3_IWL3)
AND
not_CLOSE =
  not_CLOSE_func(time,
                 timer_not_CLOSE',
                 not_CLOSE_TRIGGER)
AND
CONSIDER_OPEN =
  CONSIDER_OPEN_func(IWL1_OWL1_OPEN_DIFFERENCE1,
                     IWL2_OWL2_OPEN_DIFFERENCE2,
                     IWL3_OWL3_OPEN_DIFFERENCE3)
AND
OPEN_TRIGGER = OPEN_TRIGGER_func(state', state)
AND
OPEN_ALLOWED = OPEN_ALLOWED_func(state')
AND
OPEN_PULS = OPEN_PULS_func(time, timer_OPEN)
AND
OPEN = OPEN_func(not_CLOSE_sys_out',
                 OPEN_PULS_out')
AND
/* dynamic computations */
```

```
not_CLOSE_sys_out' = not_CLOSE
AND
not_CLOSE_TRIGGER_out' = not_CLOSE_TRIGGER
AND
OPEN_PULS_out' = OPEN_PULS
AND
OPEN_sys_out' = OPEN
AND
OPEN_TRIGGER_out' = OPEN_TRIGGER
AND
OPEN_ALLOWED_sys_out' = OPEN_ALLOWED
AND
IF (not_CLOSE_TRIGGER_out = T AND
    not_CLOSE_TRIGGER = F)
THEN
     /* -|_ */
     timer_not_CLOSE' = time + down
ELSE timer_not_CLOSE' = timer_not_CLOSE
FI AND
/* state machine */
(IF state = s1 AND
    not_CLOSE = F AND
    FORBID_OPEN_sys_in = F
 THEN state' = s2 AND
      timer_OPEN' = timer_OPEN
 ELSE IF state = s2 AND
         FORBID_OPEN_sys_in = T
      THEN state' = s1 AND
           timer_OPEN' = timer_OPEN
      ELSE IF state = s2 AND
              CONSIDER_OPEN = T AND
              not_CLOSE = T AND
              FORBID_OPEN_sys_in = F
           THEN state' = s3 AND
                timer_OPEN' = time + up
           ELSE IF state = s3 AND
                   OPEN_PULS_out = T
                THEN state' = s1 AND
                     timer_OPEN' = timer_OPEN
                ELSE state' = state AND
                     timer_OPEN' = timer_OPEN
                FI
           FI
```

```
                    FI
                 FI) AND
                /* scheduling actions */
                EnSys = T AND
                EnSys' = F AND
                EnUpd' = T AND
                EnEnv' = F
 SPEC INITIAL
              /* initiation of the pulsgever timers */
              timer_not_CLOSE = 0;
              timer_OPEN = 0;
              /*initiation of the visible output
                and internal variables */
              state = s1;
              not_CLOSE_sys_out = T;
              OPEN_sys_out = F;
              OPEN_ALLOWED_sys_out = F;
              not_CLOSE_TRIGGER_out = T;
              OPEN_TRIGGER_out = F;
              OPEN_PULS_out = F;
              EnSys = T
      TRANSITIONS [SVKO_Action] {timer_not_CLOSE,
                                 timer_OPEN,
                                 not_CLOSE_sys_out,
                                 OPEN_sys_out,
                                 OPEN_ALLOWED_sys_out,
                                 not_CLOSE_TRIGGER_out,
                                 OPEN_TRIGGER_out,
                                 OPEN_PULS_out, state}
 HIDE timer_not_CLOSE, timer_OPEN,
      not_CLOSE_TRIGGER_out,
      OPEN_TRIGGER_out,
      OPEN_PULS_out, state
TLSPECEND

5. StatFunctions
================

THEORY StatFunctions
 PURPOSE
 "description of the intermediate values functions "
 USING BasicDatas
 FUNCTIONS IWL_TRIGGER_comparator : nat, nat,
```

```
                                  nat, bool -> bool;
          OWL_TRIGGER_comparator : nat, nat -> bool;
          OWL_IWL_comparator : nat, nat -> bool;
          OPEN_DIFFERENCE_comparator : nat, nat, nat -> bool;
          input_puls_func : bool, bool, bool,
                            bool, bool, bool,
                            bool, bool, bool -> bool;
          not_CLOSE_func : nat, nat, bool -> bool;
          CONSIDER_OPEN_func : bool, bool, bool -> bool;
          OPEN_PULS_func : nat, nat -> bool;
          OPEN_func : bool, bool -> bool;
          OPEN_TRIGGER_func : zustand, zustand -> bool;
          OPEN_ALLOWED_func : zustand -> bool;
          WL_less_TL : nat, nat, nat, nat,
                       nat, nat, nat, nat,
                       nat, nat, nat, nat,
                       nat, nat, nat, bool -> bool
VARS n1, n2, n3 : nat;
     b1, b2, b3, b4, b5, b6, b7, b8, b9 : bool;
     z1, z2 : zustand;
     ig1, ig2, ig3, ith1, ith2, ith3,
     itl1, itl2, itl3, og1, og2, og3, ot1, ot2, ot3 : nat
AXIOMS FOR IWL_TRIGGER_comparator :
             IWL_TRIGGER_comparator(n1,n2,n3,b1) = T <->
               /* IWLx < TRIGGER */
               less_comp(n1,n2) AND
               (less_comp(n1,n3) OR
               b1 = T)
       FOR OWL_TRIGGER_comparator :
             OWL_TRIGGER_comparator(n1,n2) = T <->
               /* OWLx < TRIGGER */
               less_comp(n1,n2)
       FOR OWL_IWL_comparator :
             OWL_IWL_comparator(n1,n2) = T <->
               /* OWLx < IWLx */
               less_comp(n1,n2)
       FOR OPEN_DIFFERENCE_comparator :
             OPEN_DIFFERENCE_comparator(n1,n2,n3) = T <->
               /* IWLx - OWLx > OPEN_DIFFERENCE */
               less_comp(n1,minus(n2,n3))
       FOR input_puls_func :
             input_puls_func(b1,b2,b3,b4,b5,b6,b7,b8,b9) = T
             <->
```

```
        /* >= 1 (Figure 5.) */
        voter2from3(b1,b2,b3) OR
        voter2from3(b4,b5,b6) OR
        voter2from3(b7,b8,b9)
FOR not_CLOSE_func :
      DEFFUNC not_CLOSE_func(n1, n2, b1) =
        /* not_CLOSE (Figure 5.) */
        IF n1 <= n2
        THEN b1 && F
        ELSE b1 && T
        FI
FOR CONSIDER_OPEN_func :
      CONSIDER_OPEN_func(b1,b2,b3) = T <->
        /* CONSIDER_OPEN */
        voter2from3(b1,b2,b3)
FOR OPEN_TRIGGER_func :
      OPEN_TRIGGER_func(z1,z2) = F <->
        /* OPEN_TRIGGER (Figure 6.) */
        z1 = s1 OR

FOR OPEN_ALLOWED_func :
      DEFFUNC OPEN_ALLOWED_func(z1) =
        /* OPEN_ALLOWED (Figure 6.) */
        IF z1 = s1
        THEN F
        ELSE T
        FI
FOR OPEN_PULS_func :
      OPEN_PULS_func(n1,n2) = T <->
        /* OPEN_PULS */
        n1 <= n2
FOR OPEN_func :
      DEFFUNC OPEN_func(b1, b2) =
        /* OPEN signal (Figure 6.) */
        b1 && b2
FOR WL_less_TL :
      WL_less_TL(ig1,ig2,ig3,ith1,
                 ith2,ith3,itl1,itl2,
                 itl3,og1,og2,og3,
                 ot1,ot2,ot3,b1) = T
      <->
      /* water level < trigger level description
          for the safety property */
```

```
              input_puls_func(
                IWL_TRIGGER_comparator(ig1,ith1,itl1,b1),
                IWL_TRIGGER_comparator(ig2,ith2,itl2,b1),
                IWL_TRIGGER_comparator(ig3,ith3,itl3,b1),
                OWL_TRIGGER_comparator(og1,ot1),
                OWL_TRIGGER_comparator(og2,ot2),
                OWL_TRIGGER_comparator(og3,ot3),
                OWL_IWL_comparator(og1,ig1),
                OWL_IWL_comparator(og2,ig2),
                OWL_IWL_comparator(og3,ig3)) = T
THEORYEND

6. Update
=========

TLSPEC Update
 PURPOSE
 "time update of the system"
 USING BasicDatas
 DATA OUT
          /* to user */
          not_CLOSE_upd_out, OPEN_upd_out : bool;
          OPEN_ALLOWED_upd_out : bool;
          time, time_comb : nat;
          IWL_GEM1_upd_out,
          IWL_GEM2_upd_out,
          IWL_GEM3_upd_out : nat;
          IWL_TRIGGER_HIGH1_upd_out,
          IWL_TRIGGER_HIGH2_upd_out,
          IWL_TRIGGER_HIGH3_upd_out : nat;
          IWL_TRIGGER_LOW1_upd_out,
          IWL_TRIGGER_LOW2_upd_out,
          IWL_TRIGGER_LOW3_upd_out : nat;
          OWL_GEM1_upd_out,
          OWL_GEM2_upd_out,
          OWL_GEM3_upd_out : nat;
          OWL_TRIGGER1_upd_out,
          OWL_TRIGGER2_upd_out,
          OWL_TRIGGER3_upd_out : nat;
          OPEN_DIFFERENCE1_upd_out,
          OPEN_DIFFERENCE2_upd_out,
          OPEN_DIFFERENCE3_upd_out : nat;
          SELECT_HIGH_TRIGGER_LEVEL_upd_out,
```

```
          FORBID_OPEN_upd_out : bool
      IN
         /* from system */
         not_CLOSE_upd_in, OPEN_upd_in : bool;
         OPEN_ALLOWED_upd_in : bool;
         /* from environment */
         IWL_GEM1_upd_in,
         IWL_GEM2_upd_in,
         IWL_GEM3_upd_in : nat;
         IWL_TRIGGER_HIGH1_upd_in,
         IWL_TRIGGER_HIGH2_upd_in,
         IWL_TRIGGER_HIGH3_upd_in : nat;
         IWL_TRIGGER_LOW1_upd_in,
         IWL_TRIGGER_LOW2_upd_in,
         IWL_TRIGGER_LOW3_upd_in : nat;
         OWL_GEM1_upd_in,
         OWL_GEM2_upd_in,
         OWL_GEM3_upd_in : nat;
         OWL_TRIGGER1_upd_in,
         OWL_TRIGGER2_upd_in,
         OWL_TRIGGER3_upd_in : nat;
         OPEN_DIFFERENCE1_upd_in,
         OPEN_DIFFERENCE2_upd_in,
         OPEN_DIFFERENCE3_upd_in : nat;
         SELECT_HIGH_TRIGGER_LEVEL_upd_in,
         FORBID_OPEN_upd_in : bool
      SHARED INOUT
                    /* scheduling variables */
                    EnEnv, EnUpd, EnSys : bool
ACTIONS
Update_Action ::= IWL_GEM1_upd_out' = IWL_GEM1_upd_in AND
                  IWL_GEM2_upd_out' = IWL_GEM2_upd_in AND
                  IWL_GEM3_upd_out' = IWL_GEM3_upd_in AND
                  OWL_GEM1_upd_out' = OWL_GEM1_upd_in AND
                  OWL_GEM2_upd_out' = OWL_GEM2_upd_in AND
                  OWL_GEM3_upd_out' = OWL_GEM3_upd_in AND
                  IWL_TRIGGER_HIGH1_upd_out' =
                    IWL_TRIGGER_HIGH1_upd_in AND
                  IWL_TRIGGER_HIGH2_upd_out' =
                    IWL_TRIGGER_HIGH2_upd_in AND
                  IWL_TRIGGER_HIGH3_upd_out' =
                    IWL_TRIGGER_HIGH3_upd_in AND
                  IWL_TRIGGER_LOW1_upd_out' =
```

```
                    IWL_TRIGGER_LOW1_upd_in AND
                  IWL_TRIGGER_LOW2_upd_out' =
                    IWL_TRIGGER_LOW2_upd_in AND
                  IWL_TRIGGER_LOW3_upd_out' =
                    IWL_TRIGGER_LOW3_upd_in AND
                  OWL_TRIGGER1_upd_out' =
                    OWL_TRIGGER1_upd_in AND
                  OWL_TRIGGER2_upd_out' =
                    OWL_TRIGGER2_upd_in AND
                  OWL_TRIGGER3_upd_out' =
                    OWL_TRIGGER3_upd_in AND
                  OPEN_DIFFERENCE1_upd_out' =
                    OPEN_DIFFERENCE1_upd_in AND
                  OPEN_DIFFERENCE2_upd_out' =
                    OPEN_DIFFERENCE2_upd_in AND
                  OPEN_DIFFERENCE3_upd_out' =
                    OPEN_DIFFERENCE3_upd_in AND
                  SELECT_HIGH_TRIGGER_LEVEL_upd_out' =
                    SELECT_HIGH_TRIGGER_LEVEL_upd_in AND
                  FORBID_OPEN_upd_out' =
                   FORBID_OPEN_upd_in AND
                  time' = time + 1 AND
                  time_comb' = time' AND
                  not_CLOSE_upd_out' = not_CLOSE_upd_in AND
                  OPEN_upd_out' = OPEN_upd_in AND
                  OPEN_ALLOWED_upd_out' = OPEN_ALLOWED_upd_in
                  AND
                  /* scheduling actions */
                  EnUpd = T AND
                  EnUpd' = F AND
                  EnSys' = T AND
                  EnEnv' = T
SPEC INITIAL
             /* initiation of the visible output */
             not_CLOSE_upd_out = T AND
             OPEN_upd_out = F AND
             OPEN_ALLOWED_upd_out = F AND
             /* setting of the scheduling variable */
             EnUpd = F AND
             /* the time initiation */
             time = 0 AND
             time_comb = 0 AND
             /* transfer from the environment init state */
```

```
          IWL_GEM1_upd_out = IWL_OWL_const AND
          IWL_GEM2_upd_out = IWL_OWL_const AND
          IWL_GEM3_upd_out = IWL_OWL_const AND
          OWL_GEM1_upd_out = IWL_OWL_const AND
          OWL_GEM2_upd_out = IWL_OWL_const AND
          OWL_GEM3_upd_out = IWL_OWL_const AND
          IWL_TRIGGER_HIGH1_upd_out =
            IWL_TRIGGER_HIGH1_const AND
          IWL_TRIGGER_HIGH2_upd_out =
            IWL_TRIGGER_HIGH2_const AND
          IWL_TRIGGER_HIGH3_upd_out =
            IWL_TRIGGER_HIGH3_const AND
          IWL_TRIGGER_LOW1_upd_out =
            IWL_TRIGGER_LOW1_const AND
          IWL_TRIGGER_LOW2_upd_out =
            IWL_TRIGGER_LOW2_const AND
          IWL_TRIGGER_LOW3_upd_out =
            IWL_TRIGGER_LOW3_const AND
          OWL_TRIGGER1_upd_out =
            OWL_TRIGGER1_const AND
          OWL_TRIGGER2_upd_out =
            OWL_TRIGGER2_const AND
          OWL_TRIGGER3_upd_out =
            OWL_TRIGGER3_const AND
          OPEN_DIFFERENCE1_upd_out =
            OPEN_DIFFERENCE1_const AND
          OPEN_DIFFERENCE2_upd_out =
            OPEN_DIFFERENCE2_const AND
          OPEN_DIFFERENCE3_upd_out =
            OPEN_DIFFERENCE3_const AND
          SELECT_HIGH_TRIGGER_LEVEL_upd_out = F AND
          FORBID_OPEN_upd_out = F
TRANSITIONS [Update_Action]
              {not_CLOSE_upd_out, OPEN_upd_out,
               OPEN_ALLOWED_upd_out, time, time_comb,
               IWL_GEM1_upd_out, IWL_GEM2_upd_out,
               IWL_GEM3_upd_out,
               IWL_TRIGGER_HIGH1_upd_out,
               IWL_TRIGGER_HIGH2_upd_out,
               IWL_TRIGGER_HIGH3_upd_out,
               IWL_TRIGGER_LOW1_upd_out,
               IWL_TRIGGER_LOW2_upd_out,
               IWL_TRIGGER_LOW3_upd_out,
```

```
                    OWL_GEM1_upd_out,
                    OWL_GEM2_upd_out,
                    OWL_GEM3_upd_out,
                    OWL_TRIGGER1_upd_out,
                    OWL_TRIGGER2_upd_out,
                    OWL_TRIGGER3_upd_out,
                    OPEN_DIFFERENCE1_upd_out,
                    OPEN_DIFFERENCE2_upd_out,
                    OPEN_DIFFERENCE3_upd_out,
                    SELECT_HIGH_TRIGGER_LEVEL_upd_out,
                    FORBID_OPEN_upd_out}
TLSPECEND

7. environment_data
===================

TLSPEC environment_data
 PURPOSE
 "SVKO environment"
 USING BasicDatas
 DATA OUT
          /* system datas */
          IWL_GEM1_sys_out,
          IWL_GEM2_sys_out,
          IWL_GEM3_sys_out : nat;
          OWL_GEM1_sys_out,
          OWL_GEM2_sys_out,
          OWL_GEM3_sys_out : nat;
          IWL_TRIGGER_HIGH1_sys_out,
          IWL_TRIGGER_HIGH2_sys_out,
          IWL_TRIGGER_HIGH3_sys_out : nat;
          IWL_TRIGGER_LOW1_sys_out,
          IWL_TRIGGER_LOW2_sys_out,
          IWL_TRIGGER_LOW3_sys_out : nat;
          OWL_TRIGGER1_sys_out,
          OWL_TRIGGER2_sys_out,
          OWL_TRIGGER3_sys_out : nat;
          OPEN_DIFFERENCE1_sys_out,
          OPEN_DIFFERENCE2_sys_out,
          OPEN_DIFFERENCE3_sys_out : nat;
          SELECT_HIGH_TRIGGER_LEVEL_sys_out,
          FORBID_OPEN_sys_out : bool;
          /* update datas */
```

```
          IWL_GEM1_upd_out,
          IWL_GEM2_upd_out,
          IWL_GEM3_upd_out : nat;
          OWL_GEM1_upd_out,
          OWL_GEM2_upd_out,
          OWL_GEM3_upd_out : nat;
          IWL_TRIGGER_HIGH1_upd_out,
          IWL_TRIGGER_HIGH2_upd_out,
          IWL_TRIGGER_HIGH3_upd_out : nat;
          IWL_TRIGGER_LOW1_upd_out,
          IWL_TRIGGER_LOW2_upd_out,
          IWL_TRIGGER_LOW3_upd_out : nat;
          OWL_TRIGGER1_upd_out,
          OWL_TRIGGER2_upd_out,
          OWL_TRIGGER3_upd_out : nat;
          OPEN_DIFFERENCE1_upd_out,
          OPEN_DIFFERENCE2_upd_out,
          OPEN_DIFFERENCE3_upd_out : nat;
          SELECT_HIGH_TRIGGER_LEVEL_upd_out,
          FORBID_OPEN_upd_out : bool
     SHARED INOUT
                  /* scheduling variables */
                  EnEnv : bool
ACTIONS
WaterLevelChange ::= EnEnv = T AND
                     EnEnv' = F
SPEC INITIAL
             /* setting of the scheduling variable */
             EnEnv = F AND
             /* initiation of the system input */
             IWL_GEM1_sys_out = IWL_OWL_const AND
             IWL_GEM2_sys_out = IWL_OWL_const AND
             IWL_GEM3_sys_out = IWL_OWL_const AND
             OWL_GEM1_sys_out = IWL_OWL_const AND
             OWL_GEM2_sys_out = IWL_OWL_const AND
             OWL_GEM3_sys_out = IWL_OWL_const AND
             IWL_TRIGGER_HIGH1_sys_out =
                IWL_TRIGGER_HIGH1_const AND
             IWL_TRIGGER_HIGH2_sys_out =
                IWL_TRIGGER_HIGH2_const AND
             IWL_TRIGGER_HIGH3_sys_out =
                IWL_TRIGGER_HIGH3_const AND
             IWL_TRIGGER_LOW1_sys_out =
```

```
   IWL_TRIGGER_LOW1_const AND
IWL_TRIGGER_LOW2_sys_out =
   IWL_TRIGGER_LOW2_const AND
IWL_TRIGGER_LOW3_sys_out =
   IWL_TRIGGER_LOW3_const AND
OWL_TRIGGER1_sys_out =
   OWL_TRIGGER1_const AND
OWL_TRIGGER2_sys_out =
   OWL_TRIGGER2_const AND
OWL_TRIGGER3_sys_out =
   OWL_TRIGGER3_const AND
OPEN_DIFFERENCE1_sys_out =
   OPEN_DIFFERENCE1_const AND
OPEN_DIFFERENCE2_sys_out =
   OPEN_DIFFERENCE2_const AND
OPEN_DIFFERENCE3_sys_out =
   OPEN_DIFFERENCE3_const AND
SELECT_HIGH_TRIGGER_LEVEL_sys_out = F AND
FORBID_OPEN_sys_out = F AND
/* initiation of the update input */
IWL_GEM1_upd_out = IWL_OWL_const AND
IWL_GEM2_upd_out = IWL_OWL_const AND
IWL_GEM3_upd_out = IWL_OWL_const AND
OWL_GEM1_upd_out = IWL_OWL_const AND
OWL_GEM2_upd_out = IWL_OWL_const AND
OWL_GEM3_upd_out = IWL_OWL_const AND
IWL_TRIGGER_HIGH1_upd_out =
   IWL_TRIGGER_HIGH1_const AND
IWL_TRIGGER_HIGH2_upd_out =
   IWL_TRIGGER_HIGH2_const AND
IWL_TRIGGER_HIGH3_upd_out =
   IWL_TRIGGER_HIGH3_const AND
IWL_TRIGGER_LOW1_upd_out =
   IWL_TRIGGER_LOW1_const AND
IWL_TRIGGER_LOW2_upd_out =
   IWL_TRIGGER_LOW2_const AND
IWL_TRIGGER_LOW3_upd_out =
   IWL_TRIGGER_LOW3_const AND
OWL_TRIGGER1_upd_out =
   OWL_TRIGGER1_const AND
OWL_TRIGGER2_upd_out =
   OWL_TRIGGER2_const AND
OWL_TRIGGER3_upd_out =
```

```
            OWL_TRIGGER3_const AND
         OPEN_DIFFERENCE1_upd_out =
            OPEN_DIFFERENCE1_const AND
         OPEN_DIFFERENCE2_upd_out =
            OPEN_DIFFERENCE2_const AND
         OPEN_DIFFERENCE3_upd_out =
            OPEN_DIFFERENCE3_const AND
         SELECT_HIGH_TRIGGER_LEVEL_upd_out = F AND
         FORBID_OPEN_upd_out = F
TRANSITIONS [WaterLevelChange]
             {IWL_GEM1_sys_out, IWL_GEM2_sys_out,
              IWL_GEM3_sys_out, OWL_GEM1_sys_out,
              OWL_GEM2_sys_out, OWL_GEM3_sys_out,
              IWL_TRIGGER_HIGH1_sys_out,
              IWL_TRIGGER_HIGH2_sys_out,
              IWL_TRIGGER_HIGH3_sys_out,
              IWL_TRIGGER_LOW1_sys_out,
              IWL_TRIGGER_LOW2_sys_out,
              IWL_TRIGGER_LOW3_sys_out,
              OWL_TRIGGER1_sys_out,
              OWL_TRIGGER2_sys_out,
              OWL_TRIGGER3_sys_out,
              OPEN_DIFFERENCE1_sys_out,
              OPEN_DIFFERENCE2_sys_out,
              OPEN_DIFFERENCE3_sys_out,
              SELECT_HIGH_TRIGGER_LEVEL_sys_out,
              FORBID_OPEN_sys_out, IWL_GEM1_upd_out,
              IWL_GEM2_upd_out, IWL_GEM3_upd_out,
              OWL_GEM1_upd_out, OWL_GEM2_upd_out,
              OWL_GEM3_upd_out,
              IWL_TRIGGER_HIGH1_upd_out,
              IWL_TRIGGER_HIGH2_upd_out,
              IWL_TRIGGER_HIGH3_upd_out,
              IWL_TRIGGER_LOW1_upd_out,
              IWL_TRIGGER_LOW2_upd_out,
              IWL_TRIGGER_LOW3_upd_out,
              OWL_TRIGGER1_upd_out,
              OWL_TRIGGER2_upd_out,
              OWL_TRIGGER3_upd_out,
              OPEN_DIFFERENCE1_upd_out,
              OPEN_DIFFERENCE2_upd_out,
              OPEN_DIFFERENCE3_upd_out,
              SELECT_HIGH_TRIGGER_LEVEL_upd_out,
```

```
                    FORBID_OPEN_upd_out}
      ;
      [] (IWL_GEM1_sys_out = IWL_GEM1_upd_out AND
          IWL_GEM2_sys_out = IWL_GEM2_upd_out AND
          IWL_GEM3_sys_out = IWL_GEM3_upd_out AND
          OWL_GEM1_sys_out = OWL_GEM1_upd_out AND
          OWL_GEM2_sys_out = OWL_GEM2_upd_out AND
          OWL_GEM3_sys_out = OWL_GEM3_upd_out AND
          IWL_TRIGGER_HIGH1_sys_out =
             IWL_TRIGGER_HIGH1_upd_out AND
          IWL_TRIGGER_HIGH2_sys_out =
             IWL_TRIGGER_HIGH2_upd_out AND
          IWL_TRIGGER_HIGH3_sys_out =
             IWL_TRIGGER_HIGH3_upd_out AND
          IWL_TRIGGER_LOW1_sys_out =
             IWL_TRIGGER_LOW1_upd_out AND
          IWL_TRIGGER_LOW2_sys_out =
             IWL_TRIGGER_LOW2_upd_out AND
          IWL_TRIGGER_LOW3_sys_out =
             IWL_TRIGGER_LOW3_upd_out AND
          OWL_TRIGGER1_sys_out =
             OWL_TRIGGER1_upd_out AND
          OWL_TRIGGER2_sys_out =
             OWL_TRIGGER2_upd_out AND
          OWL_TRIGGER3_sys_out =
             OWL_TRIGGER3_upd_out AND
          OPEN_DIFFERENCE1_sys_out =
             OPEN_DIFFERENCE1_upd_out AND
          OPEN_DIFFERENCE2_sys_out =
             OPEN_DIFFERENCE2_upd_out AND
          OPEN_DIFFERENCE3_sys_out =
             OPEN_DIFFERENCE3_upd_out AND
          SELECT_HIGH_TRIGGER_LEVEL_sys_out =
             SELECT_HIGH_TRIGGER_LEVEL_upd_out AND
          FORBID_OPEN_sys_out = FORBID_OPEN_upd_out)
TLSPECEND

              ============================
              Crossreference of Relations
              ============================
```

```
BasicDatas [Theory]
        References:
                Update [Tlspec]
                StatFunctions [Theory]
                SVKO_combine [Tlspec]
                environment_data [Tlspec]


SVKO_combine [Tlspec]
        References:
                SVKO_safety [Tlspec]


SVKO_safety [Tlspec]
        References:
                SVKO_combine [Tlspec]


SVKO_system [Tlspec]
        References:
                SVKO_combine [Tlspec]


StatFunctions [Theory]
        References:
                SVKO_system [Tlspec]


Update [Tlspec]
        References:
                SVKO_combine [Tlspec]


environment_data [Tlspec]
        References:
                SVKO_combine [Tlspec]
```

B

Real-Time in TLA and VSE-II

B.1 Lossy Queue

As a specification and verification example taken from [4] we describe the lossy queue shown in Figure B.1 in TLA. The queue is a closed system, i.e. it does not communicate with the environment. The environment is thus part of the system itself and does not communicate.

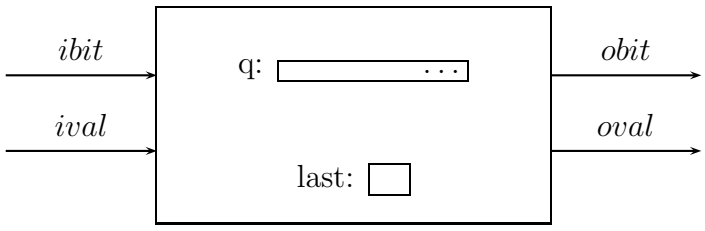

Figure B.1: Lossy Queue

The lossy queue is specified using the following flexible variables with their corresponding meanings:

- *ival*: input value
- *ibit*: input flag
- *oval*: output value
- *obit*: output flag
- *q*: the sequence of messages received but not yet output.
- *last*: a flag that prohibits the queue from inserting a message more than once.

The lossy queue receives a message m as a pair consisting of the input value *ival* and the input flag *ibit*. The receiver, the queue, notifies a new message by keeping track of the flag *ibit*. If this bit is changed, then there is a new input. The output consists of a pair $oval, obit$.

Since there is no implemented acknowledgement protocol concerning the insertion of elements into the queue, it must not catch all the values ever sent.

Being more precise, inputs are always lost in pairs, i.e. if in the actual input, a pair $(ival, ibit)$, is lost, then the next input will also be lost because of the use of the variable *ibit*, see Figure B.2. We see that the specification of the lossy queue allows for behaviours where input values are lost. A property that the queue can guarantee is that the sequence consisting of output values is a "part of" the sequence of the input sequence. The informal definition of the relation "part of" is that a sequence of values σ is part of a sequence τ if and only if all the elements from σ also occur in the sequence τ and the order of the elements is maintained.

The specification of the described lossy queue is given in Figure B.2.

The actions perform the following steps:

- $Init_Q$: Initially the values of the variables *ibit* and *obit* are arbitrary booleans. The values of *ival* and *oval* are taken from some set Msg representing the set of all possible messages. q is set to the empty sequence and *last* is set to be equal to *ibit*.
- Inp: Inp describes the state changes representing the sending of an input value. The Inp action is always enabled. It follows that input values can be sent at every time without any preconditions. When an input is sent then the flexible variable *ibit* is inverted. The new value is taken from the set Msg of allowed messages and all the other variables remain unchanged.
- EnQ: The message is received and inserted into the queue q. The enabling condition for Enq is $last \neq ibit$. After the insertion of the message into the queue the *last* flag is set to *ibit*. This way it is not possible to insert a message more than once into the queue when it was sent only once. It is clear that this condition does not prevent the reception of the same message more than once. All the other variables remain unchanged.

$$
\begin{array}{lcl}
Init_Q & \widehat{=} & ibit, obit \in \{true, false\} \\
 & & \wedge\ ival, oval \in Msg \\
 & & \wedge\ last = ibit \\
 & & \wedge\ q = \ll\gg \\
Inp & \widehat{=} & ibit' = \neg ibit \\
 & & \wedge\ ival' \in Msg \\
 & & \wedge\ (obit, oval, q, last)' = (obit, oval, q, last) \\
EnQ & \widehat{=} & last \neq ibit \\
 & & \wedge\ q' = q \circ \ll ival \gg \\
 & & \wedge\ last' = ibit \\
 & & \wedge\ (ibit, obit, ival, oval)' = (ibit, obit, ival, oval) \\
DeQ & \widehat{=} & q \neq \ll\gg \\
 & & \wedge\ oval' - Head(q) \\
 & & \wedge\ q' = Tail(q) \\
 & & \wedge\ obit' = \neg obit \\
 & & \wedge\ (ibit, ival, last)' = (ibit, ival, last) \\
N_Q & \widehat{=} & Inp \vee EnQ \vee DeQ \\
v & \widehat{=} & ibit, obit, ival, oval, q, last \\
\Pi_Q & \widehat{=} & Init_q \wedge \Box[N_Q]_{\langle v \rangle} \\
\Phi_Q & \widehat{=} & \exists q, last : \Pi_Q
\end{array}
$$

Figure B.2: TLA Specification of the Lossy Queue

- DeQ: In some sense DeQ is the inverse action to EnQ. It removes the head element of the queue q and sets the output variable $oval$ to that value. An action like DeQ makes only sense if the queue q is not the empty queue.
- N_Q is defined to be the disjunction of all the possible actions described above.
- Φ_Q represents the definition of the lossy queue where the variables q and $last$ are hidden, i.e. internal variables.

The specification as given in Figure B.2 describes the behaviour of the lossy queue with respect to the variables $ibit, obit, oval, ival$ and abstracts from the values of the variables q and $last$.

A behaviour fulfils Φ_Q if and only if there are sequences of values for q and $last$ such that Pi_Q is fulfilled, where Pi_Q represents the specification of Φ_Q without hiding (see Definition 4.1.17).

In TLA specifications variables can have arbitrary values from a given domain. TLA itself is not typed. Type-correctness can be expressed using a type invariant T_Q:

$$\begin{aligned} T_Q \;\widehat{=}\; & ibit, obit, last \in \{true, flase\} \\ & \wedge\, ival, oval \in Msg \\ & \wedge\, q \in Msg^* \end{aligned}$$

where Msg^* represents the set of all finite sequences of messages. Type-correctness is expressed by the formula $\Pi_Q \Longrightarrow \Box T_Q$ that has to be proved.

The formulae Π_Q and Φ_Q allow for behaviours that behave "well" in the beginning and then stop, i.e. they stutter forever. Such behaviours are omitted from the possible behaviour by imposing additional fairness constraints. The formula Φ_Q is then changed to:

$$\exists q, last : (Init_Q \wedge \Box[N_Q]_{\langle v \rangle} \wedge \mathcal{WF}_v(DeQ) \wedge \mathcal{SF}_v(EnQ))$$

As mentioned in Section 4.1 the formula $\mathcal{WF}_v(DeQ)$ expresses that if action DeQ is always enabled, then it is infinitely often executed. The formula $\mathcal{SF}_v(EnQ)$ describes that if action EnQ is infinitely often enabled, then it is infinitely often executed. If infinitely many messages are sent, then there are infinitely many messages inserted into the queue. Looking at both fairness constraints we can derive that infinitely many inputs result in infinitely many outputs although some inputs are lost.

B.1.1 Non-lossy Version

In this section we extend the lossy queue specification with real-time constraints that prevent the queue from losing input values. Technically this is achieved by

the introduction of timers (see Section 4.1). There are four constraints added to the specification of the lossy queue from Section B.2:

- Values are sent at most all δ_{snd} seconds. This constraint results in two timers, one for the input and one for the output interface. The timers are δ_{snd}–timer t_{Inp} and t_{DeQ} for the actions Inp and DeQ as lower bound timers.
- A value must be inserted into the queue Δ_{rcv} seconds after its sending. This constraint is expressed by the Δ_{rcv}–timer T_{EnQ} for the EnQ action. This timer is an upper bound timer.
- A value that is inserted into the queue must be output within Δ_{snd} later when it was inserted into the queue. This is expressed by the Δ_{snd}–timer T_{DeQ} as an upper–bound timer for the DeQ action.

The timed queue Π_Q^t is obtained by conjoining the corresponding timing requirements introduced to the specification Π_Q.

$$\begin{aligned}\Pi_Q^t \;\widehat{=}\; & \Pi_Q \wedge RT_v \\ & \wedge VTimer(t_{Inp}, Inp, \delta_{snd}, v) \wedge MinTime(t_{Inp}, Inp, v) \\ & \wedge VTimer(t_{DeQ}, DeQ, \delta_{snd}, v) \wedge MinTime(t_{DeQ}, DeQ, v) \\ & \wedge VTimer(T_{EnQ}, EnQ, \Delta_{rcv}, v) \wedge MaxTime(T_{EnQ}) \\ & \wedge VTimer(T_{DeQ}, DeQ, \Delta_{snd}, v) \wedge MaxTime(T_{DeQ})\end{aligned}$$

The formula

$$VTimer(T_{EnQ}, EnQ, \Delta_{rcv}, v) \wedge MaxTime(T_{EnQ})$$

expresses that a $\langle EnQ \rangle_v$ action cannot be continuously enabled for more than Δ_{rcv} seconds without being executed.

The formula

$$VTimer(t_{Inp}, Inp, \delta_{snd}, v) \wedge MinTime(t_{Inp}, Inp, v)$$

implies that an $\langle Inp \rangle_v$ action has to be enabled for at least δ_{snd} seconds before it can be executed. The meaning of the other timing requirements can be explained analogously.

The specification Π_Q^t can be transformed into the canonical form given in Figure B.3.

The specification Φ_Q^t including the hiding of the timers is obtained by existential quantifying the timers and the variables q and $last$.

An example for a behaviour of the timed non-lossy queue is given in Figure B.4 that clarifies that this version does not lose any values. The table in Figure B.4 describes in every row the execution of an action and the corresponding consequences.

$$
\begin{array}{lcl}
Init_Q^t & \widehat{=} & Init_Q \\
 & & \wedge\ now \in \mathbb{R} \\
 & & \wedge\ t_{Inp} = now + \delta_{snd} \\
 & & \wedge\ t_{DeQ} = T_{EnQ} = T_{DeQ} = \infty \\
Inp^t & \widehat{=} & Inp \\
 & & \wedge\ t_{Inp} \leq now \\
 & & \wedge\ t'_{Inp} = now' + \delta_{snd} \\
 & & \wedge\ T'_{EnQ} = \textbf{ if } last' \neq ibit' \textbf{ then } now' + \Delta_{rcv} \textbf{ else } \infty \\
 & & \wedge\ (t_{DeQ}, T_{DeQ})' = \textbf{ if } q = \ll\gg \textbf{ then } (\infty, \infty) \textbf{ else } (t_{DeQ}, T_{DeQ}) \\
 & & \wedge\ now' = now \\
EnQ^t & \widehat{=} & EnQ \\
 & & \wedge\ T'_{EnQ} = \infty \\
 & & \wedge\ (t_{DeQ}, T_{DeQ})' = \textbf{ if } q = \ll\gg \textbf{ then } (now + \delta_{snd}, now + \Delta_{snd}) \\
 & & \qquad\qquad \textbf{else } (t_{DeQ}, T_{DeQ}) \\
 & & \wedge\ (t_{Inp}, now)' = (t_{Inp}, now) \\
DeQ^t & \widehat{=} & DeQ \\
 & & \wedge\ t_{DeQ} \leq now \\
 & & \wedge\ (t_{DeQ}, T_{DeQ})' = \textbf{ if } q' = \ll\gg \textbf{ then } (\infty, \infty) \\
 & & \qquad\qquad \textbf{else } (now + \delta_{snd}, now + \Delta_{snd}) \\
 & & \wedge\ T'_{EnQ} = \textbf{ if } last' = ibit' \text{ then } \infty \textbf{ else } T_{EnQ} \\
 & & \wedge\ (t_{Inp}, now)' = (t_{Inp}, now) \\
QTick & \widehat{=} & now' \in]now, min(T_{DeQ}, T_{Enq})] \\
 & & \wedge\ (v, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ})' = (v, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ}) \\
vt & \widehat{=} & (v, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ}) \\
\Pi_Q^t & \widehat{=} & Init^t \\
 & & \wedge\ \Box[Inp^t \vee EnQ^t \vee DeQ^t \vee QTick]_{vt}
\end{array}
$$

Figure B.3: TLA Specification of the Non-Lossy Queue

Action	$QTick$	$QTick$	Inp^t	$QTick$	EnQ^t	$QTick$	Inp^t	EnQ^t	...
Act. No.	1	2	3	4	5	6	7	8	...
$ival$	$\in Msg$	$\in Msg$	$\in Msg$	1	1	1	1	2	...
$ibit$	t	t	t	f	f	f	f	t	...
q	$\ll\gg$	$\ll\gg$	$\ll\gg$	$\ll\gg$	$\ll\gg$	$\ll 1 \gg$	$\ll 1 \gg$	$\ll 1 \gg$	...
$last$	t	t	t	t	t	f	f	f	...
$oval$	$\in Msg$	$\in Msg$	$\in Msg$	$\in Msg$	$\in Msg$	$\in Msg$	$\in Msg$	$\in Msg$	...
$obit$	t	t	t	t	t	t	t	t	...
now	$now(=0)$	n_1	n_2	n_2	n_3	n_3	n_4	n_4	...
t_{Inp}	$now + \delta_{snd}$	δ_{snd}	δ_{snd}	$n_2 + \delta_{snd}$	$n_2 + \delta_{snd}$	$n_2 + \delta_{snd}$	$n_2 + \delta_{snd}$	$n_4 + \delta_{snd}$	...
t_{DeQ}	∞	∞	∞	∞	∞	$n_3 + \delta_{snd}$	$n_3 + \delta_{snd}$	$n_3 + \delta_{snd}$	...
T_{EnQ}	∞	∞	∞	$n_2 + \Delta_{rcv}$	$n_2 + \Delta_{rcv}$	∞	∞	$n_4 + \Delta_{rcv}$	...
T_{DeQ}	∞	∞	∞	∞	∞	$n_3 + \Delta_{snd}$	$n_3 + \Delta_{snd}$	$n_3 + \Delta_{snd}$	...

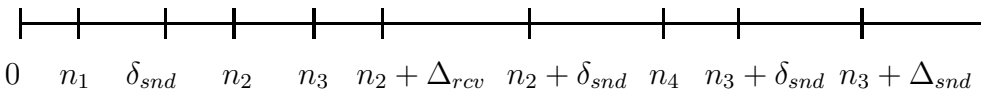

Figure B.4: Example Behaviour of the Timed Queue

In Figure B.4 a time line gives the ordering of the different timers used in the example with respect to the time variable *now*.

Some remarks on the specification of the non-lossy queue.

- There is only one operation enabled in the initial state, $QTick$. Inp^t is enabled if the value of the time variable *now* is greater than the timer δ_{snd}, $now > \delta_{snd}$. This is a difference to the lossy version. In this version the input action Inp was always enabled. In the non-lossy version Inp must be enabled for at least δ_{snd} before it can be taken.
- The action $QTick$ changes only the value of the variable *now* and leaves all the other variables unchanged.
- After action 4 (see Figure B.4) the Inp cannot take place since the condition $t_{Inp} \leq now$ is not fulfilled right after the $QTick$ action.
- In action 6 the value of the variable *now* is not allowed to be increased over $n_3 + \Delta_{snd}$.
- If the timer Δ_{snd} is less than the timer δ_{snd}, then after action 8 a time blocked state is reached. This means that the action $QTick$ can be taken but there is an upper bound for the enlargement of *now*, $n_3 + \Delta_{snd}$. The only actions that can change the value of the timer T_{DeQ} are Inp^t and DeQ^t. But these actions are not enabled because of the fact that t_{Inp} as well as t_{DeQ} are greater as the actual value of the variable *now*. The action $QTick$ has to increase the value of *now* in order to enable one of these actions, which is in this case impossible since the upper bound for *now* is T_{DeQ} and it holds that $T_{DeQ} < t_{DeQ}$.

  A specification as Π_Q^t is called *Zeno* which means that it allows the system only to change the variable *now*, but with an upper bound for *now*. Usually such behaviours indicate an error in the specification of the system. The error in this case was that we have imposed a time constraint on the output with the intention to block the input of values.
- Generally it is possible that the action $QTick$ is the only action ever taken since the timers that restrict the variable *now* are set to ∞ in the initial state.
- If $\Delta_{rcv} \geq \delta_{snd}$, then a state can be reached where an old message can be overwritten by a new one without having inserted the old message into the queue. The consequence of that is that the timed queue would also be lossy. Therefore, the condition $\Delta_{rcv} < \delta_{snd}$ must be required.

The formula Π_Q^t is a TLA specification that fulfils each maximum delay constraint by preventing *now* from advancing before the constraint has been satisfied. That way one could think that the specification implements timing constraints by stopping the time. But this is not the case. The formula Π_Q^t only says that an action cannot be executed when its timing requirements are not satisfied, i.e. an action can only take place if *now* has advanced to a certain value.

B.1.2 Reasoning about Time

A property that the timed queue should satisfy is that under certain timing conditions no input values are lost. Since TLA has no past temporal operators, the past has to be specified explicitely. This is done with the history-determined variables *hin* with

$$\begin{aligned} H_{in} \;\widehat{=}\; & hin = \ll\gg \\ & \wedge\Box[hin' = hin \circ \ll ival' \gg \wedge (ival, ibit)' \neq (ival, ibit)]_{\langle hin, ival, ibit\rangle} \end{aligned}$$

for the inputs and *hout* with

$$\begin{aligned} H_{out} \;\widehat{=}\; & hout = \ll\gg \\ & \wedge\Box[hout' = hout \circ \ll oval' \gg \wedge (oval, obit)' \neq (oval, obit)]_{\langle hout, oval, obit\rangle} \end{aligned}$$

for the outputs. As can be seen in the definition of the variables *hin* and *hout*, *hin* collects all the inputs and *hout* stores all the outputs ever made. The property that the timed queue does not lose any value is expressed by the following formula:

$$\Pi_Q^t \wedge H_{in} \wedge H_{out} \Longrightarrow (hout \preceq hin)$$

where $\alpha \preceq \beta$ holds if and only if α is an initial prefix of β.

Trying to prove this property we have to find an invariant that helps to prove it. The following invariant is taken from [8].

$$\begin{aligned} & T_Q \wedge (t_{Inp}, now \in \mathbb{R}) \wedge (T_{EnQ}, t_{DeQ}, T_{DeQ} \in \mathbb{R} \cup \{\infty\}) \\ & \wedge\ now \leq min(T_{EnQ}, T_{DeQ}) \\ & \wedge\ (last = ibit) \Longrightarrow ((T_{EnQ} = \infty) \wedge (hin = hout \circ q)) \\ & \wedge\ (last \neq ibit) \Longrightarrow ((T_{EnQ} < t_{Inp}) \wedge (hin = hout \circ q \circ \ll ival \gg)) \\ & \wedge\ (q = \ll\gg) \equiv (T_{DeQ} = \infty) \end{aligned}$$

In order to prove the mentioned property the assumption $\Delta_{rcv} < \delta_{snd}$ is needed. Having proved the invariant it immediately follows that $\Box(hout \preceq hin)$ holds and we are done. The reader who is interested in the TLA proof is best referred to [8].

B.1.3 The Lossy Queue in VSE-II

In this section we present the specification of the previous described lossy queue done in VSE-II. We show only the temporal logic part of the specification since it is the most interesting one in this context. It defines the behaviour of the whole system whereas the underlying datatypes define the types, functions or predicates

```
TLSPEC LossyQueue
 PURPOSE
 " Specification of the lossy queue"
 PARAMS QueueData
 USING Queue[QueueDataType]
 DATA INTERNAL q : QueueType;
               last : bool
      OUT ibit, obit : bool;
          ival, oval : QueueDataType
 ACTIONS
 Init ::= last = ibit AND
          q = emptyqueue
 Inp ::= ibit' /= ibit AND
         UNCHANGED(obit, oval, q, last)
 Enq ::= last /= ibit AND
         q' = enqueue(ival, q) AND
         last' = ibit AND
         UNCHANGED(ibit, obit, ival, oval)
 Deq ::= q /= emptyqueue AND
         oval' = head(q) AND
         q' = tail(q) AND
         obit' = ~ obit AND
         UNCHANGED(ibit, ival, last)
 SPEC INITIAL Init
      TRANSITIONS [Inp, Enq, Deq] {ibit, obit, ival, oval, q, last}
 HIDE q, last
TLSPECEND
```

Figure B.5: Specification of the Lossy Queue in VSE-II

used in this specification. The complete specification can be found in Appendix B.2.

As can be seen in Figure B.5 the lossy queue has `ibit, obit, ival, oval` and `q` as flexible variables where `q` and `last` are internal variables and hidden from the outside world.

The formula in the `SPEC` slot describes the behaviour of the specification. As can easily be recognised the specification is very similar to the TLA specification of the lossy queue and therefore no further explanations are needed.

B.1.4 The Non–Lossy Real-time Version of the Queue in VSE-II

In this section a real-time version of the Lossy Queue is specified. The added timing requirements prohibit the losing of values.

B.1.4.1 Time

In the following example we use discrete time in the specification. As the domain for the time we choose the natural numbers. In this way we avoid the temporal version of Zeno's paradox. We represent time as a variable `now` in the specification. Because of the fact that time advances forever we have specified it in a separate temporal logic specification that is shown in Figure B.6.

```
TLSPEC Time
 USING HelpTheory
 DATA OUT now : natinf
 ACTIONS
 tick ::= now' = now + 1
 SPEC INITIAL now = 0
      TRANSITIONS [tick] {now}
      FAIRNESS WF(tick) {now}
TLSPECEND
```

Figure B.6: Specification of Time.

The `TLSPEC Time` consists of one action that increments `now`. We could have chosen as well a predefined *timestep* that is specified as a rigid variable in the `TLSPEC Time`. This rigid variable defines the difference between `now` and `now'`. For simplicity we have taken the constant 1. The `tick` action has no explicit enabling condition and therefore it can happen at any time. The behaviour of the specification is described in the `SPEC` slot. Initially the value assigned to `now` is 0 and the only action that can be taken is the `tick` action. The variable `now` is advanced forever which is expressed by a weak fairness condition.

The type `natinf` used in the specification of `Time` consists of the natural numbers together with an infinity symbol `infinity` (see Figure B.7).

```
BASIC NatInf
 USING natural
 natinf = infinity WITH infinityp |
          nat2natinf(natinf2nat : nat) WITH natinfp
BASICEND

THEORY NatInfTheory
 USING NatInf
 FUNCTIONS _ + _ : natinf,natinf -> natinf
 PREDICATES _<_ : natinf, natinf;
            _<=_ : natinf, natinf
 VARS n1, n2 : natinf
 AXIOMS FOR + : DEFFUNC n1 + n2 =
                IF natinfp(n1) AND natinfp(n2)
                THEN nat2natinf(natinf2nat(n1) + natinf2nat(n2))
                ELSE infinity
                FI
        FOR < : DEFPRED n1 < n2 <->
                IF natinfp(n1) AND natinfp(n2)
                THEN natinf2nat(n1) < natinf2nat(n2)
                ELSE IF infinityp(n1)
                     THEN FALSE
                     ELSE TRUE
                     FI
                FI
        FOR <= : DEFPRED n1 <= n2 <->
                 n1 < n2 OR
                 n1 = n2
THEORYEND
```

Figure B.7: Specification of the Naturals with an Infinity Element.

We have two possibilities to insert time into the lossy queue specification.

- We can include the `TLSPEC` Time in the specification of the lossy queue. The semantics of the new specification is that there is a time represented by `now` and time keeps advancing forever. Additionally, all behaviours of the so specified queue must fulfil the queue specification as well as the time specification. We can refer in the actions of the queue to the variable `now`. This is done in such a way that `now` is not changed by the queue itself (there are no primed occurrences of now) but only by the `tick` action.

- We can combine the queue and the time specification. Again in this case the behaviour of the composed system must be a behaviour of the queue and the time system. This is achieved without inconsistencies with the combine operator. The variable `now` is an `IN` variable for the queue specification and an `OUT` variable for the time specification. The queue is not able to change the time, since it is an `IN` variable for the queue.

We have taken the second possibility to specify the timed queue, since this seemed to be more adequate in this case. The specification in VSE-II is given in Figure B.8 and B.9.

```
TLSPEC TimedQueue
 PURPOSE
 " Specification of the timed Non--Lossy Queue "
 PARAMS QueueData
 USING Queue[QueueDataType];
       HelpTheory[QueueDataType]
 DATA INTERNAL T_Deq, T_Enq, tInp, tDeq : natinf;
               q : QueueType;
               last : bool
      IN now : natinf
      OUT ibit, obit : bool;
          ival, oval : QueueDataType
 VARS D_rcv, D_snd, dsnd : natinf
 ACTIONS
 Init_t ::= last = ibit AND q = emptyqueue AND
            tInp = now + dsnd AND tDeq = infinity AND
            T_Enq = infinity AND T_Deq = infinity
 Inp_t ::= ibit' = ~ ibit AND tInp < now AND
           tInp' = now + dsnd AND
           T_Enq' = fT_Enq(last', ibit', now, D_rcv) AND
           tDeq' = ftTDeq(q, tDeq) AND T_Deq' = ftTDeq(q, T_Deq)
 Enq_t ::= last /= ibit AND q' = enqueue(ival, q) AND
           last' = ibit AND T_Enq' = infinity AND
           tDeq' = ftDeq2(q, now, dsnd, tDeq) AND
           T_Deq' = fT_Deq2(q, now, Dsnd, T_Deq) AND
           UNCHANGED(ibit, tInp, obit, ival, oval)
```

Figure B.8: Specification of the Timed Queue (1)

Inspecting the specification reveals that the first part of every action is the same as in the specification of the lossy queue. The remaining parts of the actions contain the change to the variables that are called *timers* in TLA (`tInp`, `T_Enq`, etc). With these timers we determine what actions are enabled and how far time

```
 Deq_t ::= q /= emptyqueue AND oval' = head(q) AND
           q' = tail(q) AND obit' = ~ obit AND
           tDeq <= now AND
         (q' = emptyqueue -> (tDeq' = infinity AND
                              T_Deq' = infinity)) AND
         (q' /= emptyqueue -> (tDeq' = now + dsnd AND
                              T_Deq' = now + D_snd)) AND
         (last' = ibit' -> T_Enq' = infinity) AND
         (last' /= ibit' -> T_Enq' = T_Enq) AND
         UNCHANGED(tInp, ibit, ival, last)

 Limit_now ::= now <= min(T_Deq, T_Enq)

 SPEC INITIAL Init_t
      TRANSITIONS [Inp_t, Enq_t, Deq_t]
                  {ibit, obit, ival, oval,
                   q, last, tInp, tDeq, T_Deq, T_Enq}
      ;
      INITIAL TRUE
      TRANSITIONS [Limit_now]
                  {ibit, obit, ival, oval, q, last, tInp, tDeq,
                   T_Deq, T_Enq}
 HIDE tInp, tDeq, q,
      last, T_Deq, T_Enq
TLSPECEND
```

Figure B.9: Specification of the Timed Queue (2).

can be advanced without violating the specification. This seems as if we would stop advancing of time with the action `Limit_now`, but this is not the case. To understand that put it the other way around. If there is a behaviour which is not accepted by our specification, because of the behaviour of `now`, then this is not a behaviour of the specified system.

The semantics of the timers used in the specification B.8 and B.9 is:

- $\texttt{t}_{\texttt{Inp}}$: A value must be put on the input wire `ival` at most once every `dsnd` seconds. This is a timer on the input action.
- $\texttt{t}_{\texttt{Deq}}$: A value must be put on the output wire `oval` at most once every `dsnd` seconds. This is a timer on the dequeue action.
- $\texttt{T}_{\texttt{Enq}}$: A value must be added to the queue at most `Drcv` seconds after it appears on the input wire. This is a timer on the enqueue action.
- $\texttt{T}_{\texttt{Deq}}$: A value must be added to the queue at most `Drcv` seconds after it appears on the input wire. This is a timer on the enqueue action.

Analysing these timers we see that the queue is non-lossy if $\texttt{Drcv} < \texttt{dsnd}$. In this case the sending of values is slower than the insertion of values into the queue.

The composition of the specification of the timed queue and the specification of time is shown in Figure B.10.

Comparing the timed and the untimed specifications of the queue, it is clear that the timed queue is non–lossy. This can be proved by introducing history variables to keep track of the values sent to the queue and removed from the queue. Comparing the results in every step of each system leads to the desired result that the lossy queue can lose values and the timed queue does not.

Furthermore, it is clear that the timed queue is a refinement of the untimed queue. To prove this, we have to map the visible variables of the timed queue to the variables of the lossy queue. The mapping is given in Figure B.11.

Depending on this mapping VSE-II generates proof obligations that can easily be proved.

In the next section we list the complete specification of the lossy and the timed queue in VSE-II.

B.2 Specification of the Lossy and the Timed Queue

```
MAPPING Timed_Lossy_Queue
 EXPORTSPEC LossyQueue
 IMPLEMENTATION Combined_Time_TimedQueue
```

```
 TLSPEC Combined_Time_TimedQueue
 PURPOSE
 " Composition of the TLSPECS Time and TimedQueue "
 PARAMS QueueData
 USING HelpTheory[QueueDataType]
 DATA OUT ibit, obit : bool;
          ival, oval : QueueDataType
      OUT now : natinf
 COMBINE TimedQueue [TimedQueue.ibit ->
                      Combined_Time_TimedQueue.ibit,
                     TimedQueue.obit ->
                      Combined_Time_TimedQueue.obit,
                     TimedQueue.ival ->
                      Combined_Time_TimedQueue.ival,
                     TimedQueue.oval ->
                      Combined_Time_TimedQueue.oval,
                     TimedQueue.now <-
                      Time.now] ;
         Time [Time.now -> Combined_Time_TimedQueue.now]
TLSPECEND
```

Figure B.10: Combined Specification of Time and the Timed Queue

```
MAPPING Timed_Lossy_Queue
 EXPORTSPEC LossyQueue
 IMPLEMENTATION Combined_Time_TimedQueue
 MAPS Combined_Time_TimedQueue.ibit IMPLEMENTS LossyQueue.ibit;
      Combined_Time_TimedQueue.obit IMPLEMENTS LossyQueue.obit;
      Combined_Time_TimedQueue.ival IMPLEMENTS LossyQueue.ival;
      Combined_Time_TimedQueue.oval IMPLEMENTS LossyQueue.oval
MAPPINGEND
```

Figure B.11: Refinement Mapping

```
 MAPS Combined_Time_TimedQueue.ibit IMPLEMENTS LossyQueue.ibit;
      Combined_Time_TimedQueue.obit IMPLEMENTS LossyQueue.obit;
      Combined_Time_TimedQueue.ival IMPLEMENTS LossyQueue.ival;
      Combined_Time_TimedQueue.oval IMPLEMENTS LossyQueue.oval
MAPPINGEND

TLSPEC TimedQueue
 PURPOSE
 " Specification of the timed Non Lossy Queue "
 PARAMS QueueData
 USING Queue[QueueDataType];
       HelpTheory[QueueDataType]
 DATA INTERNAL T_Deq, T_Enq, tInp, tDeq : natinf;
               q : QueueType;
               last : bool
      IN now : natinf
      OUT ibit, obit : bool;
          ival, oval : QueueDataType
 VARS D_rcv, D_snd, dsnd : natinf
 ACTIONS
 Init_t ::= last = ibit AND
            q = emptyqueue AND
            tInp = now + dsnd AND
            tDeq = infinity AND
            T_Enq = infinity AND
            T_Deq = infinity
 Inp_t ::= ibit' = ~ ibit AND
           tInp < now AND
           tInp' = now + dsnd AND
           T_Enq' = fT_Enq(last', ibit', now, D_rcv) AND
           tDeq' = ftTDeq(q, tDeq) AND
           T_Deq' = ftTDeq(q, T_Deq)
 Enq_t ::= last /= ibit AND
           q' = enqueue(ival, q) AND
           last' = ibit AND
           T_Enq' = infinity AND
           tDeq' = ftDeq2(q, now, dsnd, tDeq) AND
           T_Deq' = fT_Deq2(q, now, Dsnd, T_Deq) AND
           UNCHANGED(ibit, tInp, obit, ival, oval)
 Deq_t ::= q /= emptyqueue AND
           oval' = head(q) AND
           q' = tail(q) AND
           obit' = ~ obit AND
```

```
           tDeq <= now AND
           (q' = emptyqueue ->
            (tDeq' = infinity AND
             T_Deq' = infinity)) AND
           (q' /= emptyqueue ->
            (tDeq' = now + dsnd AND
             T_Deq' = now + D_snd)) AND
           (last' = ibit' ->
            T_Enq' = infinity) AND
           (last' /= ibit' ->
            T_Enq' = T_Enq) AND
           UNCHANGED(tInp, ibit, ival, last)
 Limit_now ::= now <= min(T_Deq, T_Enq)
 SPEC INITIAL Init_t
      TRANSITIONS [Inp_t, Enq_t, Deq_t]
      {ibit, obit, ival, oval, q, last,
       tInp, tDeq, T_Deq, T_Enq}
      ;
      INITIAL TRUE
      TRANSITIONS [Limit_now]
      {ibit, obit, ival, oval, q, last,
       tInp, tDeq, T_Deq, T_Enq}
 HIDE tInp, tDeq, q, last, T_Deq, T_Enq
TLSPECEND

TLSPEC Time
 PARAMS QueueData
 USING HelpTheory[QueueDataType]
 DATA OUT now : natinf
 ACTIONS
 tick ::= now' = now + nat2natinf(1)
 SPEC INITIAL now = nat2natinf(0)
      TRANSITIONS [tick] {now}
      FAIRNESS WF(tick) {now}
TLSPECEND


THEORY HelpTheory
 PARAMS QueueData
 USING boolean;
       NatInfTheory;
       Queue[QueueDataType]
 FUNCTIONS ftDeq2, fT_Deq2 : QueueType, natinf, natinf, natinf ->
```

```
                              natinf;
           ftTDeq : QueueType, natinf -> natinf;
           fT_Enq : bool, bool, natinf, natinf -> natinf;
           min : natinf, natinf -> natinf
 VARS q : QueueType;
      n1, n2, n3 : natinf;
      b1, b2 : bool
 AXIOMS FOR fT_Deq2 : DEFFUNC fT_Deq2(q, n1, n2, n3) =
                      IF q = emptyqueue
                      THEN n1 + n2
                      ELSE n3
                      FI
        FOR ftDeq2 : DEFFUNC ftDeq2(q, n1, n2, n3) =
                     IF q = emptyqueue
                     THEN n1 + n2
                     ELSE n3
                     FI
        FOR ftTDeq : DEFFUNC ftTDeq(q, n1) =
                     IF q = emptyqueue
                     THEN infinity
                     ELSE n1
                     FI
        FOR fT_Enq : DEFFUNC fT_Enq(b1, b2, n1, n2) =
                     IF b1 /= b2
                     THEN n1 + n2
                     ELSE infinity
                     FI
        FOR min : DEFFUNC min(n1, n2) =
                  IF n1 = infinity
                  THEN n2
                  ELSE IF n2 = infinity
                       THEN n1
                       ELSE IF n1 <= n2
                            THEN n1
                            ELSE n2
                            FI
                       FI
                  FI
THEORYEND


THEORY Queue
 PURPOSE
```

```
 " Specification of the datatype queue"
 PARAMS QueueData
 USING boolean
 TYPES QueueType =
       FREELY GENERATED BY emptyqueue |
                           enqueue(head : QueueDataType,
                                   tail : QueueType)
 FUNCTIONS last : QueueType -> QueueDataType
 VARS queue : QueueType
 AXIOMS FOR last : DEFFUNC last(queue) =
                   IF queue /= emptyqueue
                   THEN IF tail(queue) /= emptyqueue
                        THEN last(tail(queue))
                        ELSE head(queue)
                        FI
                   FI
THEORYEND


THEORY QueueData
 PURPOSE
 " Definition of the types of the elements of the queue"
 TYPES QueueDataType
THEORYEND


TLSPEC LossyQueue
 PURPOSE
 " Specification of the lossy queue"
 PARAMS QueueData
 USING Queue[QueueDataType]
 DATA INTERNAL q : QueueType;
               last : bool
      OUT ibit, obit : bool;
          ival, oval : QueueDataType
 ACTIONS
 Init ::= last = ibit AND
          q = emptyqueue
 Inp ::= ibit' /= ibit AND
         UNCHANGED(obit, oval, q, last)
 Enq ::= last /= ibit AND
         q' = enqueue(ival, q) AND
         last' = ibit AND
```

```
         UNCHANGED(ibit, obit, ival, oval)
 Deq ::= q /= emptyqueue AND
         oval' = head(q) AND
         q' = tail(q) AND
         obit' = ~ obit AND
         UNCHANGED(ibit, ival, last)
 Nq ::= Inp OR
        Enq OR
        Deq
 SPEC INITIAL Init
      TRANSITIONS [Nq] {ibit, obit, ival, oval, q, last}
 HIDE q, last
TLSPECEND


BASIC NatInf
 USING natural
 natinf = infinity WITH infinityp |
          nat2natinf(natinf2nat : nat) WITH natinfp
BASICEND


THEORY NatInfTheory
 USING NatInf
 FUNCTIONS _ + _ : natinf,natinf -> natinf
 PREDICATES _<_ : natinf, natinf;
            _<=_ : natinf, natinf
 VARS n1, n2 : natinf
 AXIOMS FOR + : DEFFUNC n1 + n2 =
                IF natinfp(n1) AND natinfp(n2)
                THEN nat2natinf(natinf2nat(n1)
                                + natinf2nat(n2))
                ELSE infinity
                FI
        FOR < : DEFPRED n1 < n2 <->
                IF natinfp(n1) AND natinfp(n2)
                THEN natinf2nat(n1) < natinf2nat(n2)
                ELSE IF infinityp(n1)
                     THEN FALSE
                     ELSE TRUE
                     FI
                FI
        FOR <= : DEFPRED n1 <= n2 <->
```

```
                n1 < n2 OR
                n1 = n2
THEORYEND


TLSPEC Combined_Time_TimedQueue
 PURPOSE
 " Composition of the TLSPECS Time and TimedQueue "
 PARAMS QueueData
 USING HelpTheory[QueueDataType]
 DATA OUT ibit, obit : bool;
          ival, oval : QueueDataType
      OUT now : natinf
 COMBINE
  TimedQueue [TimedQueue.ibit -> Combined_Time_TimedQueue.ibit,
              TimedQueue.obit -> Combined_Time_TimedQueue.obit,
              TimedQueue.ival -> Combined_Time_TimedQueue.ival,
              TimedQueue.oval -> Combined_Time_TimedQueue.oval,
              TimedQueue.now <- Time.now] ;
  Time [Time.now -> Combined_Time_TimedQueue.now]
TLSPECEND
```

C

VSE-II Proof Rules

There are many rules implemented in the deduction component of the VSE-II tool, namely rules for *dynamic logic*, first-order logic and temporal logic. We confine ourselves to the rules of temporal logic. The description of these rules is taken from the VSE-II manual.

C.1 Basic Rules

C.1.1 Axiom

$$\frac{}{\exists v.x = v, \Gamma \vdash \Delta}\ \textbf{flex range left}$$

- x is a flexible variable
- v is a rigid variable

C.1.2 □ Rules

$$\frac{\phi, \bigcirc\phi, \Gamma \vdash \Delta}{\Box\phi, \Gamma \vdash \Delta}\ \textbf{always left} \qquad \frac{Al(\Gamma) \vdash \phi, Ev(\Delta)}{\Gamma \vdash \Box\phi, \Delta}\ \textbf{always right}$$

- *Al* and *Ev* select invariant formulae.

$$\frac{\Gamma \vdash \phi, \Delta \quad \phi, Al(\Gamma) \vdash \bigcirc\phi, Ev(\Delta)}{\Gamma \vdash \Box\phi, \Delta}\ \textbf{always right induction}$$

- *Al* and *Ev* select invariant formulae (see C.2.2).

C.1.3 $\Diamond$ Rules

$$\frac{\phi, Al(\Gamma) \vdash Ev(\Delta)}{\Diamond\phi, \Gamma \vdash \Delta}\ \textbf{eventually left} \qquad \frac{\Gamma \vdash \phi, \bigcirc\phi, \Delta}{\Gamma \vdash \Diamond\phi, \Delta}\ \textbf{eventually right}$$

- *Al* and *Ev* select invariant formulae.

C.1.4 $\mathcal{U}$ Rules

$$\frac{\phi_2, \Gamma \vdash \Delta \quad \phi_1, \bigcirc(\phi_1 \textbf{ unless } \phi_2), \Gamma \vdash \phi_2, \Delta}{\phi_1 \textbf{ unless } \phi_2, \Gamma \vdash \Delta}\ \textbf{unless left}$$

$$\frac{\Gamma \vdash \phi_2, \Delta}{\Gamma \vdash \phi_1 \textbf{ unless } \phi_2, \Delta}\ \textbf{unless right exit}$$

$$\frac{\Gamma \vdash \phi_1, \Delta \quad \Gamma \vdash \bigcirc(\phi_1 \textbf{ unless } \phi_2), \Delta}{\Gamma \vdash \phi_1 \textbf{ unless } \phi_2, \Delta}\ \textbf{unless right step}$$

$$\frac{\Gamma \vdash \phi_1, \phi_2, \Delta \quad \phi_1, AlUnl_{\phi_2}(\Gamma) \vdash \phi_2, \bigcirc(\phi_1 \vee \phi_2), Ev(\Delta)}{\Gamma \vdash \phi_1 \textbf{ unless } \phi_2, \Delta}\ \textbf{unless right induction}$$

- $AlUnl_{\phi_2}$ selects invariant formulae, and also preserves $\mathcal{U}$ formulae.
- *Ev* selects invariant formulae of the succedent.

C.1.5 Step Rule

$$\frac{Al_m(\Gamma) \vdash Ev_m(\Delta)}{\Gamma \vdash \Delta}\ \textbf{step}$$

- m maps all free flexible, non-primed variables onto new rigid variables.
- Al_m and Ev_m select invariant formulae, and transfer formulae which are not invariant to next state.

C.1.6 Induction Rule

$$\frac{\Box\forall v.(v \ll v_0 \rightarrow \phi[v]), Al(\Gamma) \vdash \phi[v_0], Ev(\Delta)}{\Gamma \vdash \forall v.\phi[v], \Delta}\ \textbf{temporal induction 1}$$

- a size function $\ll$ for v must exist
- v_0 is a new rigid variable, and
- $\phi[v]$ indicates that the variable v occurs in the formula ϕ.

C.1.7 Quantifiers

$$\frac{\phi[x], \Gamma \vdash \Delta}{\exists y.\phi[y], \Gamma \vdash \Delta}\ \textbf{hide left}$$

- ϕ must be invariant under stuttering
- x is a new flexible variable

$$\frac{\Gamma \vdash \phi[\tau], \Delta}{\Gamma \vdash \exists y.\phi[y], \Delta}\ \textbf{hide right}$$

- $\phi[\tau]$ is the substitution of y by term τ; τ may contain flexible and primed variables; substituting terms in temporal logic is different from substitution in predicate logic (s. [6])

C.2 Definitions

C.2.1 Mapping into Next State

Nx_m is defined as follows:

$$\begin{array}{lcll}
Nx_m(\bigcirc\phi) & = & \phi & \\
Nx_m(\phi[(\xi,\xi')]) & = & \phi[(m(\xi),\xi)] & \phi \text{ a predicate logic formula} \\
Nx_m(\forall v.\phi) & = & \forall v.Nx_m(\phi) & \text{if } Nx_m(\phi) \neq nil \\
Nx_m(\exists v.\phi) & = & \exists v.Nx_m(\phi) & \text{if } Nx_m(\phi) \neq nil \\
Nx_m(\phi) & = & nil & \text{otherwise}
\end{array}$$

C.2.2 Filters selecting Invariant Formulae

If a temporal rule is applied, it is often necessary to modify also the context of the formula, which was the focus of the rule application. If a premise states properties about an arbitrary future step in the temporal trace, only invariant properties of the context are preserved.

C.2.2.1 Standard Filters

Al is defined as follows:

$$\begin{array}{lcll} Al(\Box\phi) & = & \Box\phi & \\ Al(SF(\phi)\{\overline{x}\}) & = & SF(\phi)\{\overline{x}\} & \\ Al(\,WF(\phi)\{\overline{x}\}) & = & WF(\phi)\{\overline{x}\} & \\ Al(\phi) & = & \phi & \phi \text{ is rigid} \\ Al(\phi) & = & nil & \text{otherwise} \end{array}$$

Ev is defined als follows:

$$\begin{array}{lcll} Ev(\Diamond\phi) & = & \Diamond\phi & \\ Ev(\phi) & = & \phi & \phi \text{ is rigid} \\ Ev(\phi) & = & nil & \text{otherwise} \end{array}$$

C.2.2.2 Filters mapping into Next State

Al_m is defined as follows:

$$\begin{array}{lcll} Al_m(\Box\phi) & = & \Box\phi & \\ Al_m(SF(\phi)\{\overline{x}\}) & = & SF(\phi)\{\overline{x}\} & \\ Al_m(\,WF(\phi)\{\overline{x}\}) & = & WF(\phi)\{\overline{x}\} & \\ Al_m(\phi) & = & Nx_m(\phi) & \text{otherwise} \end{array}$$

Ev_m is defined as follows:

$$\begin{array}{lcll} Ev_m(\Diamond\phi) & = & \Diamond\phi & \\ Ev_m(\phi) & = & Nx_m(\phi) & \text{otherwise} \end{array}$$

C.2.2.3 Filter preserving $\mathcal{U}$ Formulae

$AlUnl_\chi$ is defined as follows:

$$
\begin{array}{lcll}
AlUnl_\chi(\Box\phi) & = & \Box\phi & \\
AlUnl_\chi(SF(\phi)\{\overline{x}\}) & = & SF(\phi)\{\overline{x}\} & \\
AlUnl_\chi(WF(\phi)\{\overline{x}\}) & = & WF(\phi)\{\overline{x}\} & \\
AlUnl_\chi(\phi) & = & \phi & \phi \text{ is rigid} \\
AlUnl_\chi(\psi_1 \textbf{ unless } \psi_2) & = & (\Box(\psi_2 \rightarrow \chi)) \rightarrow (\psi_1 \textbf{ unless } \psi_2) & \\
AlUnl_\chi(\phi) & = & nil & \text{otherwise}
\end{array}
$$

C.2.2.4 Filters respecting Stuttering Variables

$Al_{\overline{s}}$ is defined as follows:

$$
\begin{array}{lcll}
Al_{\overline{s}}(\phi \odot \Gamma) & = & Al_{\overline{s}}(\phi) \odot Al_{\overline{s}}(\Gamma) & \\
Al_{\overline{s}}(\Box\phi) & = & \Box\phi & \\
Al_{\overline{s}}(SF(\phi)\{\overline{x}\}) & = & SF(\phi)\{\overline{x}\} & \\
Al_{\overline{s}}(WF(\phi)\{\overline{x}\}) & = & WF(\phi)\{\overline{x}\} & \\
Al_{\overline{s}}(\phi) & = & \phi & \phi \text{ a predicate logic formula} \\
 & & & \text{and } \mathit{flexvars}(\phi) \subseteq \overline{s} \\
Al_{\overline{s}}(\phi) & = & nil & \text{otherwise}
\end{array}
$$

with $\odot \in \{\wedge, \vee, \ldots\}$.

$Ev_{\overline{s}}$ is defined as follows:

$$
\begin{array}{lcll}
Ev_{\overline{s}}(\phi \odot \Gamma) & = & Ev_{\overline{s}}(\phi) \odot Ev_{\overline{s}}(\Gamma) & \\
Ev_{\overline{s}}(\Diamond\phi) & = & \Diamond\phi & \\
Ev_{\overline{s}}(\phi) & = & \phi & \phi \text{ a predicate logic formula} \\
 & & & \text{and } \mathit{flexvars}(\phi) \subseteq \overline{s} \\
Ev_{\overline{s}}(\phi) & = & nil & \text{otherwise}
\end{array}
$$

C.2.3 Progress Conditions

$\mathcal{F}_{\overline{s}}(\mathcal{P}, \Gamma)$ is defined as

$$
\begin{array}{lcll}
\mathcal{F}_{\overline{s}}(\mathcal{P}, \Gamma) & = & \mathcal{F}_{\overline{s}}(\mathit{Steps}(\mathcal{P}), \Gamma) & \\
\mathcal{F}_{\overline{s}}(\phi, \Gamma) & = & \mathit{nil} & \text{if } WF(\phi)\{\overline{x}\} \notin \Gamma \\
 & & & \text{and } SF(\phi)\{\overline{x}\} \notin \Gamma \\
\mathcal{F}_{\overline{s}}(\phi, \Gamma) & = & \mathit{Enabled}\ (\phi \wedge \overline{x}' \neq \overline{x}) & \text{if } \mathit{flexvars}(\phi) \subseteq \overline{s} \\
\mathcal{F}_{\overline{s}}(\phi, \Gamma) & = & \Box\Diamond(\mathit{Enabled}\ (\phi \wedge \overline{x}' \neq \overline{x})) & \text{if } SF(\phi)\{\overline{x}\} \in \Gamma \\
\mathcal{F}_{\overline{s}}(\phi, \Gamma) & = & \Diamond\Box(\mathit{Enabled}\ (\phi \wedge \overline{x}' \neq \overline{x})) & \text{if } WF(\phi)\{\overline{x}\} \in \Gamma
\end{array}
$$

Bibliography

[1] Common criteria for information technology security evaluation (CC), U. S. National Institute of Standards and Technology, Version 2.1, August 1999.

[2] Stankovic J. A. and K. Ramamritham. What is predictability for realtime system. volume 2, pages 247–254. Kluwer Academic Publishers, 1990.

[3] K. M. Hansen A. P. Ravn, H. Rischel. Specifying and verifying requirements of realtime systems. *IEEE Transactions on software Engineering*, 19(1):41–55, January 1993.

[4] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, pages 1–27. Springer Verlag, New York, LNCS 600, 1992.

[5] M. Abadi and L. Lamport. Composing specifications. In *ACM Transactions on Programming Languages and Systems*, volume 15(1), pages 73 – 132, 1993.

[6] M. Abadi and S. Merz. On tla as a logic. In M. Broy, editor, *Deductive Program Design*, NATO ASI series, pages 235–272. Springer, 1996.

[7] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[8] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[9] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[10] J. Abrial, E. Boerger, 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.

[11] J-R. Abrial. The B tool. In G. Goos and J. Hartmanis, editors, *VDM – The Way Ahead. Proc. 2nd VDM-Europe Symposium*, volume 328 of *Lecture Notes in Computer Science*, pages 86–87. VDM-Europe, Springer-Verlag, 1988.

[12] R. Alur. *Techniques for Authomatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[13] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 414–425, 1990.

[14] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifaksi, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[15] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 209–229. Springer Verlag, Lecture Notes in Computer Science, vol. 736, 1993.

[16] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[17] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *ICALP 90: Automata, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 322–355, 1990.

[18] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. In *Proceedings of the 10th Annual Symposium on Priciples of Distributed Computing*, pages 139–152. ACM, 1991.

[19] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, New York, 1990.

[20] R. Alur and T. A. Henzinger. Logics and models of real-time: A survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, pages 74–106. Springer Verlag, New York, LNCS 600, 1992.

[21] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the Association for Computing Machinery*, 41(1):181–204, 1994.

[22] R. Alur, T. A. Henzinger, and E. Sontag, editors. *Hybrid Systems III*. Lecture Notes in Computer Science, Springer Verlag, 1996.

[23] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[24] Rajeev Alur, Thomas A. Henzinger, and Howard Wong-Toi. Symbolic analysis of hybrid systems. In *Proceedings of the 36th International IEEE Conference on Decision and Control (CDC 1997)*, pages 702–707, 1997.

[25] Anonymous. Reprogramming capability proves key to extending voyager 2's journey. In *Aviation Week and Space Technology*, page 72, August 1989.

[26] Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors. *Algebraic foundations of systems specification*. IFIP state-of-the-art reports. Springer, Berlin, 1999.

[27] B Core UK Ltd. *B-Tool manual*, 1994.

[28] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of uppaal. In *International Workshop on Software Tools for Technology Transfer, Aalborg, Denmark*, 1998.

[29] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[30] A. Bernstein and P. K. Jr. Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th Annual Symposium on Priciples of Programminig Languages*, pages 1–11. ACM Press, 1981.

[31] D. Bert. B'98: Recent advances in the development and use of the B method. In *Second International B Conference*, volume 1393 of *LNCS*. Springer-Verlag, April 1998.

[32] Z. Chaochen, M. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Symposium on Theoretical Aspects of Computer Science (STACS 93)*, volume 665 of *Lecture Notes in Computer Science*, pages 58–68. Springer-Verlag, 1993.

[33] Zhao Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

[34] Brian F. Chellas. *Modal logic an introduction*. Cambridge University Press, 1980.

[35] E. M. Clarke, , E. A. Emerson, and A. P. Sisla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):244–263, 1986.

[36] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings CAV 89: Automatic Verification Systems for Finite-State Systems*, pages 197–212. Springer Verlag, Lecture Notes in Computer Science, vol. 407, 1989.

[37] E. A. Emerson, A. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *CAV 90: Computer Aided Verification*, pages 163–145. Lecture Notes in Computer Science, vol. 531, Springer Verlag, New York, 1990.

[38] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[39] U. Engberg, P. Grnning, and L. Lamport. Mechanical verification of concurrent systems with tla.

[40] J. Garman. The bug heard 'round the world. In *ACM Software Engineering Notes*, volume 6-5, pages 3–10, 1981.

[41] Hafer and Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Annual International Colloquium on Automata, Languages and Programming*, 1987.

[42] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, New York, 1990.

[43] M. Heisel and C. Sühl. Formal specification of safety-critical software with z and real-time csp. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 31–45. Springer Verlag, 1996.

[44] C. L. Heitmeyer, B. G. Labaw, P. C. Clements, and A. K. Mok. Engineering CASE tools to support formal methods for real-time software development. In G. Forte, N. H. Madhavji, and H. A. Mueller, editors, *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, Canada, 1992. IEEE Computer.

[45] T. A. Henzinger. Half-order modal logic: How to prove real-time properties. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 281–286. ACM, New York, 1990.

[46] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford University, Stanford, Ca., 1991.

[47] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th LICS*, pages 278–292. IEEE Comp. Soc. Press, 1996.

[48] T. A. Henzinger and P.-H. Ho. HyTech: The cornell hybrid technology tool. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, pages 265–293. Springer Verlag, Lecture Notes in Computer Science, vol. 999, 1995.

[49] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[50] G. Holzmann. The model checker spin. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295, 1997.

[51] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, , and A. Wolpers. Verification support environment (vse). In *Journal of High Integrity Systems*, 1996.

[52] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the Verification Support Environment (VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings Formal Methods Europe 1996: Industrial Benefits and Advances in Formal Methods*. SPRINGER, 1996.

[53] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Verification support environment (vse). *High Integrity Systems*, 1(6):523–530, 1996.

[54] Dieter Hutter, Heiko Mantel, Georg Rock, Werner Stephan, Andreas Wolpers, Michael Balser, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. VSE: Controlling the complexity in formal software developments. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proceedings Current Trends in Applied Formal Methods, FM-Trends 98*, Boppard, Germany, 1999. Springer-Verlag, LNCS 1641.

[55] Dieter Hutter, Georg Rock, Jörg H. Siekmann, Werner Stephan, and Roland Vogt. Formal Software Development in the Verification Support Environment (VSE). In Bill Manaris Jim Etheredge, editor, *FLAIRS-2000: Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference*, pages 367–376. AAAI-Press, 2000.

[56] Dieter Hutter and Axel Schairer. Towards an evolutionary formal software development. In *Proceedings 16th IEEE International Conference on Automated Software Engineering, ASE-2001*, San Diego, USA, 2001. IEEE Computer Society.

[57] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification*

CAV, volume 1102, pages 244–256, New Brunswick, NJ, USA, 1996. Springer Verlag.

[58] Y. Kesten, O. Lichtenstein, and A. Pnueli. A complete axiomatization of ptl. Technical report, Weizmann Institute, 1995.

[59] R. Koymans and W. P. Deroever. Examples of a real-time temporal logic specification. In B. T. Denvir, W. T. Harwood, M. I. Jackson, and M. J. Wray, editors, *Analysis of Concurrent Systems*, volume 207, pages 231–251. Springer-Verlag, Berlin-Heidelberg-New York, 1985.

[60] R. Koymans, J. Vytopyl, and W. de Roever. Real-time programming and asynchronous message-passing. In *Proceedings of the Second Annual Symposium on Priciples of Distributed Computing*, pages 187–197. ACM Press, 1983.

[61] Saul A. Kripke. Semantical analysis of modal logic I. In *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, volume 9, pages 67–96, 1963.

[62] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Proceedings of the Fifth International Conference, CAV'93, Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179. Springer-Verlag, June 1993.

[63] S. Merz L. Lamport. Specifying and verifying fault-tolerant systems. In J. Vytopil H. Langmaack, W. P. de Roever, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 1994, Lübeck: 3rd International Symposium, organized jointly with the Working Group Provably Correct Systems ProCoS, Lübeck, Germany, September 19-23, 1994*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, 1994.

[64] L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer-Verlag, 1993.

[65] L. Lamport. Specifying concurrent systems with tla+. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, number 173 in F: Computer and Systems Sciences, pages 183–247. IOS Press, Amsterdam, 1999.

[66] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[67] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[68] C. D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, pages 37–53, 1992.

[69] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Teubner, Chichester;New York;Brisbane, 1996.

[70] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, New York, 1992.

[71] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, 2000. IEEE Computer Society Press.

[72] K. McMillan. The smv system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[73] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[74] Jonathan K. Millen. CAPSL: Common authentication protocol specification language. The MITRE Corporation, Technical Report MP 97B48, 1997. http://www.csl.sri.com/ millen/capsl.

[75] R. Milner. *Communicating and Concurrency*. Prentice Hall, New York, 1989.

[76] A. Nonnengart. A resolution-based calculus for temporal logics. In *PhD Thesis*. Universität Saarbrücken, December 1995.

[77] A. Nonnengart, G. Rock, and W. Stephan. Expressing Realtime Properties in VSE-II. In *ESA Workshop on On-Board Autonomy*, volume WPP-191, pages 447–454, October 2001.

[78] Andreas Nonnengart. A deductive model checking approach for hybrid systems. Research Report MPI-I-1999-2-006, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, November 1999.

[79] Andreas Nonnengart. Hybrid systems verification by location elimination. In Nancy Lynch and Bruce H. Krogh, editors, *Proceedings of the 3rd International Workshop HSCC 2000*, pages 352–365. Springer Verlag, LNCS 1790, 2000.

[80] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verication of linear hybrid systems. In *Proceedings of the 6th Computer-Aided Verication, CAV*, volume 818 of *LNCS*, pages 81–94, 1994.

[81] J. S. Ostroff. *Temporal Logic of Real-Time Systems*. Research Studies Press, Taunton, England, 1990.

[82] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, New York, 1977.

[83] A. Pnueli. Applications of temporal logic to the specication and verication of reactive systems: a survey of current trends. In J. W. Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510 – 584. Springer-Verlag, 1986.

[84] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–93. Lecture Notes in Computer Science, vol. 331, Springer Verlag, New York, 1988.

[85] Koymans R. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[86] W. Reif. Correctness of generic modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*, volume 620 of *LNCS*. Springer, 1992.

[87] Georg Rock, Werner Stephan, and Andreas Wolpers. Tool support for the compositional development of distributed systems. In *Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, number 315 in GMD Studien. GMD, 1997.

[88] Georg Rock, Werner Stephan, and Andreas Wolpers. Assumption–Commitment Specifications and Safety-Critical Systems. In Hartmut Knig and Peter Langendrfer, editors, *FBT'98. Formale Beschreibungstechniken für verteilte Systeme*, pages 125–135. Shaker Verlag, Aachen, 1998. 8. GI/ITG-Fachgespräch.

[89] Georg Rock, Werner Stephan, and Andreas Wolpers. Modular Reasoning about Structured TLA Specifications. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 217–229. Springer, WienNewYork, 1999.

[90] Axel Schairer and Dieter Hutter. Proof transformations for evolutionary formal software development. In *Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002*. Springer-Verlag, LNCS 2422, 2002.

[91] S. Schneider. Using csp for protocol analysis: the needham-schroeder publickey protocol. Technical Reprot CSD-TR-96-14, University of London, 1996.

[92] S. Schneider, J. Davies, D. Jackson, G. Reed, J. Reed, and A. Roscoe. Timed csp: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *Lecture Notes of Computer Sience*, pages 640–675, June 1992.

[93] Steve Schneider. *Concurrent and real-time systems : the CSP approach*. Worldwide series in computer science. Wiley, 1999.

[94] J. M. Spivey, editor. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, second edition, 1992.

[95] Johan van Benthem. *The Logic of Time*. Reidel, Dordrecht, 1990.

[96] X. Yong. A justification assistant for duration calculus. Technical Report 126, United Nations University, 1997.

[97] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

Index

Symbols

Γ_δ 96
$\breve{}$ 114
$\downarrow$ 156
$\hat{}$ 113
$\lceil$ 156
$\mathbb{Q}$ 76
π 85
$\star$ 76
$\tilde{=}$ 108

A

Abstract Data Type 38
admissible 74
ADT 38
always left rule 225
always right induction rule 226
always right rule 225
aperiodic 7
assumption 42
asymmetry 15

B

behaviour 28
behaviour predicate 25
bounded temporal operator 10
branching time 9, 10

C

canonical form 31
clock 12
combine 41
concurrency 40
constraint formula 71
constraint term 71
continuous time 9
copy automaton 119
CSP 155

D

DC, induction rule 151
deadline 7
density 15
directedness 15
discrete time 9
discretisation 96
Duration Calculus 147
Duration Calculus, formula 148
Duration Calculus, gasburner 152
Duration Calculus, semantics 148
Duration Calculus, state expr. . . . 148
Duration Calculus, syntax 148
Duration Calculus, term 148

E

enabled 29
equal up to 28
event extraction 157
eventually left rule 226
eventually right rule 226

F

fairness 30, 206
flex range left rule 225
formal methods 1
freely generated 38
freeze quantification 11
future operator 15

G

gasburner 19
granularity change 75
guarantee 42

H

history-determined variable 31
Hybrid Automata 71

I

include 41
integral 149
integration 83
interleaving 41
interpretation 27

K

Kripke structure 15

L

leads-to 119
linear time 9
linearity 15

M

message extraction 156
model checker 35

N

nested 118
non-zeno 75
now 32

O

observer model 126

P

past operator 15
periodic 7
position 75
projection 156
Proof rules, VSE-II 225
property 31
Property Specification Language . . 77
PSL, semantics 77
PSL, syntax 77

Q

qualitative, time 8
quantitative, time 9
queue, lossy 203
queue, non-lossy 206

R

reachable 74
real-time 71
requirements engineering 60
run 74

S

safety property 31
seriality 15
similar up to 27, 28
state 27, 73
state function 25
state predicate 25
step rule 226
strong fairness 30, 206
stuttering 28
stuttering equivalence 28
stuttering invariant 31
suffix 75
SVKO 60
SVKO, specification 177

T

temporal induction rule 227
temporal logic 9
temporal operator 10
temporal quantifier rules 227
Timed CSP 155
timely-reachable 74
TLA, lower-bound timer 33

TLA, MaxTime 33
TLA, MinTime 33
TLA, pyramid 23
TLA, real-time 32
TLA, semantics 26
TLA, syntax 23
TLA, Temporal Logic of Actions . . 23
TLA, timer 33
TLA, TLC 35
TLA, upper-bound timer 33
TLC 35
transition formula 25
transition function 25
transition predicate 25
transition-reachable 74
transitivity 15

U

unless rules 226

V

variable, history-determined 31
Verification Support Environment . 37
VSE-I 37
VSE-II 37, 225

W

weak fairness 30, 206