Modeling and Verification of Real-Time Systems

Formalisms and Software Tools

Edited by
Stephan Merz
Nicolas Navet

iSTE

This page intentionally left blank

Modeling and Verification of Real-Time Systems

This page intentionally left blank

Modeling and Verification of Real-Time Systems

Formalisms and Software Tools

Edited by
Stephan Merz
Nicolas Navet

iSTE

First published in Great Britain and the United States in 2008 by ISTE Ltd and John Wiley & Sons, Inc.

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms and licenses issued by the CLA. Enquiries concerning reproduction outside these terms should be sent to the publishers at the undermentioned address:

ISTE Ltd
6 Fitzroy Square
London W1T 5DX
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

© ISTE Ltd, 2008

The rights of Stephan Merz and Nicolas Navet to be identified as the authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

Library of Congress Cataloging-in-Publication Data

Modeling and verification of real-time systems : formalisms and software tools / edited by Nicolas Navet, Stephan Merz.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-1-84821-013-4
1. Real-time data processing. 2. Computer software--Verification. 3. Formal methods (Computer science) I. Navet, Nicolas. II. Merz, Stephan.
QA76.54.M635 2008
004.01'51--dc22

2007045063

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN: 978-1-84821-013-4

Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire.

Contents

Preface 15
Stephan MERZ and Nicolas NAVET

Chapter 1. Time Petri Nets – Analysis Methods and Verification with TINA 19
Bernard BERTHOMIEU, Florent PERES and François VERNADAT

1.1. Introduction 19
1.2. Time Petri nets 20
1.2.1. Definition 20
1.2.2. States and the state reachability relation 20
1.2.3. Illustration 22
1.2.4. Some general theorems 23
1.3. State class graphs preserving markings and LTL properties 24
1.3.1. State classes 24
1.3.2. Illustration 25
1.3.3. Checking the boundedness property on-the-fly 26
1.3.4. Variations 27
1.3.4.1. Multiple enabledness 27
1.3.4.2. Preservation of markings (only) 28
1.4. State class graphs preserving states and LTL properties 28
1.4.1. Clock domain 28
1.4.2. Construction of the $SSCG$ 29
1.4.3. Variants 30
1.5. State class graphs preserving states and branching properties 32
1.6. Computing firing schedules 33
1.6.1. Schedule systems 33
1.6.2. Delays (relative dates) versus dates (absolute) 34
1.6.3. Illustration 35
1.7. An implementation: the `Tina` environment 35

1.8. The verification of $SE-LTL$ formulae in `Tina` 37
1.8.1. The temporal logic $SE-LTL$ 37
1.8.2. Preservation of LTL properties by `tina` constructions 39
1.8.3. `selt`: the $SE-LTL$ checker of `Tina` 39
1.8.3.1. Verification technique 39
1.8.3.2. The `selt` logic 40
1.9. Some examples of use of `selt` 42
1.9.1. John and Fred 42
1.9.1.1. Statement of problem 42
1.9.1.2. Are the temporal constraints appearing in this scenario consistent? 42
1.9.1.3. Is it possible that Fred took the bus and John the carpool? . . 44
1.9.1.4. At which time could Fred have left home? 44
1.9.2. The alternating bit protocol 44
1.10. Conclusion 47
1.11. Bibliography 48

Chapter 2. Validation of Reactive Systems by Means of Verification and Conformance Testing 51
Camille CONSTANT, Thierry JÉRON, Hervé MARCHAND and Vlad RUSU

2.1. Introduction 51
2.2. The IOSTS model 54
2.2.1. Syntax of IOSTS 54
2.2.2. Semantics of IOSTS 56
2.3. Basic operations on IOSTS 57
2.3.1. Parallel product 57
2.3.2. Suspension 58
2.3.3. Deterministic IOSTS and determinization 60
2.4. Verification and conformance testing with IOSTS 60
2.4.1. Verification 60
2.4.1.1. Verifying safety properties 62
2.4.1.2. Verifying possibility properties 63
2.4.1.3. Combining observers 63
2.4.2. Conformance testing 64
2.5. Test generation 64
2.6. Test selection 68
2.7. Conclusion and related work 70
2.8. Bibliography 73

Chapter 3. An Introduction to Model Checking 77
Stephan MERZ

3.1. Introduction 77
3.2. Example: control of an elevator 78

3.3. Transition systems and invariant checking 79
3.3.1. Transition systems and their runs 81
3.3.2. Verification of invariants 82
3.4. Temporal logic 84
3.4.1. Linear-time temporal logic 84
3.4.2. Branching-time temporal logic 87
3.4.3. ω-automata 89
3.4.4. Automata and PTL 92
3.5. Model checking algorithms 94
3.5.1. Local PTL model checking 95
3.5.2. Global CTL model checking 97
3.5.3. Symbolic model checking algorithms 99
3.6. Some research topics 103
3.7. Bibliography 105

Chapter 4. Model Checking Timed Automata 111
Patricia BOUYER and François LAROUSSINIE

4.1. Introduction 111
4.2. Timed automata 112
4.2.1. Some notations 112
4.2.2. Timed automata, syntax and semantics 113
4.2.3. Parallel composition 114
4.3. Decision procedure for checking reachability 115
4.4. Other verification problems 118
4.4.1. Timed languages 118
4.4.2. Branching-time timed logics 118
4.4.3. Linear-time timed logics 120
4.4.4. Timed modal logics 121
4.4.5. Testing automata 121
4.4.6. Behavioral equivalences 121
4.5. Some extensions of timed automata 121
4.5.1. Diagonal clock constraints 122
4.5.2. Additive clock constraints 123
4.5.3. Internal actions 124
4.5.4. Updates of clocks 125
4.5.5. Linear hybrid automata 126
4.6. Subclasses of timed automata 127
4.6.1. Event-recording automata 127
4.6.2. One-clock timed automata 128
4.6.3. Discrete-time models 129
4.7. Algorithms for timed verification 130
4.7.1. A symbolic representation for timed automata: the zones 130
4.7.2. Backward analysis in timed automata 131

4.7.3. Forward analysis of timed automata 132
4.7.4. A data structure for timed systems: DBMs 133
4.8. The model-checking tool Uppaal 134
4.9. Bibliography 135

Chapter 5. Specification and Analysis of Asynchronous Systems using CADP 141
Radu MATEESCU

5.1. Introduction 141
5.2. The CADP toolbox 142
5.2.1. The LOTOS language 143
5.2.2. Labeled transition systems 143
5.2.3. Some verification tools 144
5.3. Specification of a drilling unit 147
5.3.1. Architecture 150
5.3.2. Physical devices and local controllers 152
5.3.2.1. Turning table 152
5.3.2.2. Clamp 153
5.3.2.3. Drill 154
5.3.2.4. Tester 154
5.3.3. Main controller – sequential version 155
5.3.4. Main controller – parallel version 157
5.3.5. Environment 158
5.4. Analysis of the functioning of the drilling unit 159
5.4.1. Equivalence checking 159
5.4.2. Model checking 161
5.5. Conclusion and future work 164
5.6. Bibliography 166

Chapter 6. Synchronous Program Verification with Lustre/Lesar 171
Pascal RAYMOND

6.1. Synchronous approach 171
6.1.1. Reactive systems 171
6.1.2. The synchronous approach 172
6.1.3. Synchronous languages 172
6.2. The Lustre language 173
6.2.1. Principles 173
6.2.2. Example: the beacon counter 174
6.3. Program verification 174
6.3.1. Notion of temporal property 175
6.3.2. Safety and liveness 175
6.3.3. Beacon counter properties 175
6.3.4. State machine 175

6.3.5. Explicit automata 176
6.3.6. Principles of model checking 176
6.3.7. Example of abstraction 177
6.3.8. Conservative abstraction and safety 178
6.4. Expressing properties 178
6.4.1. Model checking: general scheme 178
6.4.2. Model checking synchronous program 179
6.4.3. Observers 180
6.4.4. Examples 180
6.4.5. Hypothesis 180
6.4.6. Model checking of synchronous programs 181
6.5. Algorithms 182
6.5.1. Boolean automaton 182
6.5.2. Explicit automaton 182
6.5.3. The “pre ” and “post ” functions 183
6.5.4. Outstanding states 183
6.5.5. Principles of the exploration 184
6.6. Enumerative algorithm 184
6.7. Symbolic methods and binary decision diagrams 185
6.7.1. Notations 185
6.7.2. Handling predicates 186
6.7.3. Representation of the predicates 186
6.7.3.1. Shannon’s decomposition 186
6.7.3.2. Binary decision diagrams 187
6.7.4. Typical interface of a BDD library 188
6.7.5. Implementation of BDDs 188
6.7.6. Operations on BDDs 189
6.7.6.1. Negation 189
6.7.6.2. Binary operators 189
6.7.6.3. Cofactors and quantifiers 190
6.7.7. Notes on complexity 191
6.7.8. Typed decision diagrams 192
6.7.8.1. Positive functions 192
6.7.8.2. TDG 192
6.7.8.3. TDG implementation 193
6.7.8.4. Interest in TDGs 193
6.7.9. Care set and generalized cofactor 194
6.7.9.1. “Knowing that” operators 194
6.7.9.2. Generalized cofactor 194
6.7.9.3. Restriction 194
6.7.9.4. Algebraic properties of the generalized cofactor 195
6.8. Forward symbolic exploration 195
6.8.1. General scheme 196

6.8.2. Detailed implementation 196
6.8.3. Symbolic image computing 198
6.8.4. Optimized image computing 198
6.8.4.1. Principles 198
6.8.4.2. Universal image 199
6.8.4.3. Case of a single transition function 199
6.8.4.4. Shannon's decomposition of the image 200
6.9. Backward symbolic exploration 201
6.9.1. General scheme 201
6.9.2. Reverse image computing 202
6.9.3. Comparing forward and backward methods 203
6.10. Conclusion and related works 203
6.11. Demonstrations 204
6.12. Bibliography 205

Chapter 7. Synchronous Functional Programming with Lucid Synchrone 207
Paul CASPI, Grégoire HAMON and Marc POUZET

7.1. Introduction 207
7.1.1. Programming reactive systems 207
7.1.1.1. The synchronous languages 208
7.1.1.2. Model-based design 210
7.1.1.3. Converging needs 211
7.1.2. Lucid Synchrone 211
7.2. Lucid Synchrone 213
7.2.1. An ML dataflow language 213
7.2.1.1. Infinite streams as basic objects 213
7.2.1.2. Temporal operations: delay and initialization 213
7.2.2. Stream functions 214
7.2.3. Multi-sampled systems 216
7.2.3.1. The sampling operator `when` 217
7.2.3.2. The combination operator `merge` 218
7.2.3.3. Oversampling 219
7.2.3.4. Clock constraints and synchrony 220
7.2.4. Static values 222
7.2.5. Higher-order features 222
7.2.6. Datatypes and pattern matching 224
7.2.7. A programming construct to share the memory 225
7.2.8. Signals and signal patterns 227
7.2.8.1. Signals as clock abstractions 227
7.2.8.2. Testing presence and pattern matching over signals 228
7.2.9. State machines and mixed designs 229
7.2.9.1. Weak and strong preemption 229
7.2.9.2. ABRO and modular reset 230

7.2.9.3. Local definitions to a state 231
7.2.9.4. Communication between states and shared memory 232
7.2.9.5. Resume or reset a state 233
7.2.10. Parametrized state machines 233
7.2.11. Combining state machines and signals 234
7.2.12. Recursion and non-real-time features 236
7.2.13. Two classical examples 236
7.2.13.1. The inverted pendulum 236
7.2.13.2. A heater 237
7.3. Discussion 240
7.3.1. Functional reactive programming and circuit description languages 240
7.3.2. Lucid Synchrone as a prototyping language 241
7.4. Conclusion 242
7.5. Acknowledgment 243
7.6. Bibliography 243

Chapter 8. Verification of Real-Time Probabilistic Systems 249
Marta KWIATKOWSKA, Gethin NORMAN, David PARKER
and Jeremy SPROSTON

8.1. Introduction 249
8.2. Probabilistic timed automata 250
8.2.1. Preliminaries 250
8.2.2. Syntax of probabilistic timed automata 252
8.2.3. Modeling with probabilistic timed automata 254
8.2.4. Semantics of probabilistic timed automata 254
8.2.5. Probabilistic reachability and invariance 255
8.3. Model checking for probabilistic timed automata 258
8.3.1. The region graph 258
8.3.2. Forward symbolic approach 261
8.3.2.1. Symbolic state operations 261
8.3.2.2. Computing maximum reachability probabilities 263
8.3.3. Backward symbolic approach 266
8.3.3.1. Symbolic state operations 266
8.3.3.2. Probabilistic until 267
8.3.3.3. Computing maximum reachability probabilities 268
8.3.3.4. Computing minimum reachability probabilities 270
8.3.4. Digital clocks 273
8.3.4.1. Expected reachability 274
8.3.4.2. Integral semantics 276
8.4. Case study: the IEEE FireWire root contention protocol 277
8.4.1. Overview 277
8.4.2. Probabilistic timed automata model 278
8.4.3. Model checking statistics 281

8.4.4. Performance analysis 282
8.5. Conclusion 285
8.6. Bibliography 285

Chapter 9. Verification of Probabilistic Systems Methods and Tools 289
Serge HADDAD and Patrice MOREAUX

9.1. Introduction 289
9.2. Performance evaluation of Markovian models 290
9.2.1. A stochastic model of discrete event systems 290
9.2.2. Discrete-time Markov chains 292
9.2.2.1. Presentation 292
9.2.2.2. Transient and steady-state behaviors of DTMC 293
9.2.3. Continuous-time Markov chains 294
9.2.3.1. Presentation 294
9.2.3.2. Transient and steady-state behaviors of CTMC 295
9.3. High level stochastic models 297
9.3.1. Stochastic Petri nets with general distributions 297
9.3.1.1. Choice policy 298
9.3.1.2. Service policy 298
9.3.1.3. Memory policy 298
9.3.2. GLSPN with exponential distributions 299
9.3.3. Performance indices of SPN 300
9.3.4. Overview of models and methods in performance evaluation . . . 300
9.3.5. The GreatSPN tool 301
9.3.5.1. Supported models 302
9.3.5.2. Qualitative analysis of Petri nets 302
9.3.5.3. Performance analysis of stochastic Petri nets 302
9.3.5.4. Software architecture 302
9.4. Probabilistic verification of Markov chains 303
9.4.1. Limits of standard performance indices 303
9.4.2. A temporal logic for Markov chains 303
9.4.3. Verification algorithms 305
9.4.4. Overview of probabilistic verification of Markov chains 306
9.4.5. The ETMCC tool 307
9.4.5.1. Language of system models 307
9.4.5.2. Language of properties 307
9.4.5.3. Computed results 308
9.4.5.4. Software architecture 308
9.5. Markov decision processes 308
9.5.1. Presentation of Markov decision processes 308
9.5.2. A temporal logic for Markov decision processes 309
9.5.3. Verification algorithms 309
9.5.4. Overview of verification of Markov decision processes 313

9.5.5. The PRISM tool 314
9.5.5.1. Language of system models 314
9.5.5.2. Properties language 314
9.5.5.3. Computed results 314
9.5.5.4. Software architecture 314
9.6. Bibliography 315

Chapter 10. Modeling and Verification of Real-Time Systems using the IF Toolset 319
Marius BOZGA, Susanne GRAF, Laurent MOUNIER and Iulian OBER

10.1. Introduction 320
10.2. Architecture 322
10.3. The IF notation 324
10.3.1. Functional features 324
10.3.2. Non-functional features 326
10.3.3. Expressing properties with observers 328
10.4. The IF tools 329
10.4.1. Core components 329
10.4.2. Static analysis 332
10.4.3. Validation 333
10.4.4. Translating UML to IF 334
10.4.4.1. UML modeling 334
10.4.4.2. The principles of the mapping from UML to IF 334
10.5. An overview on uses of IF in case studies 336
10.6. Case study: the Ariane 5 flight program 337
10.6.1. Overview of the Ariane 5 flight program 337
10.6.2. Verification of functional properties 339
10.6.3. Verification of non-functional properties 343
10.6.4. Modular verification and abstraction 344
10.7. Conclusion 345
10.8. Bibliography 347

Chapter 11. Architecture Description Languages: An Introduction to the SAE AADL 353
Anne-Marie DÉPLANCHE and Sébastien FAUCOU

11.1. Introduction 353
11.2. Main characteristics of the architecture description languages 356
11.3. ADLs and real-time systems 357
11.3.1. Requirement analysis 357
11.3.2. Architectural views 359
11.4. Outline of related works 360
11.5. The AADL language 362
11.5.1. An overview of the AADL 363

11.6. Case study 365
11.6.1. Requirements 365
11.6.2. Architecture design and analysis with AADL 366
11.6.2.1. High-level design 366
11.6.2.2. Thread and communication timing semantics 369
11.6.2.3. Technical overview of the flow latency analysis algorithm . 373
11.6.2.4. Modeling fault-tolerance mechanisms 374
11.6.3. Designing for reuse: package and refinement 377
11.7. Conclusion 380
11.8. Bibliography 381

List of Authors 385

Index 389

Preface

The study of real-time systems has been recognized over the past 30 years as a discipline of its own whose research community is firmly established in academia as well as in industry. This book aims at presenting some fundamental problems, methods, and techniques of this domain, as well as questions open for research.

The field is mainly concerned with the control and analysis of dynamically evolving systems for which requirements of timeliness are paramount. Typical examples include systems for the production or transport of goods, materials, energy or information. Frequently, controllers for these systems are "embedded" in the sense that they are physically implemented within the environment with which they interact, such as a computerized controller in a plane or a car. This characteristic imposes strong constraints on space, cost, and energy consumption, which limits the computational power and the available memory for these devices, in contrast with traditional applications of computer science where resources usually grow exponentially according to Moore's law. The design of real-time systems relies on techniques that originate in several disciplines, including control theory, operations research, software engineering, stochastic process analysis and others.

Software supporting real-time systems needs not only to compute the correct value of a given function, but it must also deliver these values at the right moment in order to ensure the safety and the required performance level of the overall system. Usually, this is implemented by imposing constraints (or *deadlines*) on the termination of certain activities. The verification techniques presented in this volume can help to ensure that deadlines are respected.

Chapter written by Stephan MERZ and Nicolas NAVET.

The chapters of this book present basic concepts and established techniques for modeling real-time systems and for verifying their properties. They concentrate on functional and timing requirements; the analysis of non-functional properties such as schedulability and Quality of Service guarantees would be a useful complement, but would require a separate volume. Formal methods of system design are based on mathematical principles and abstractions; they are a cornerstone for a "zero-default" discipline. However, their use for the development of real-world systems requires the use of efficient support tools. The chapters therefore emphasize the presentation of verification techniques and tools associated with the different specification methods, as well as the presentation of case studies that illustrate the application of the formalisms and the tools. The focus lies on model checking approaches, which attempt to provide a "push-button" approach to verification and integrate well into standard development processes. The main obstacle for the use of model checking in industrial-sized developments is the state-explosion problem, and several chapters describe techniques based on abstraction, reduction or compression that stretch the limits of the size of systems that can be handled.

Before verification can be applied, the system must be modeled in a formal description language such as (timed) Petri nets, timed automata or process algebra. The properties expected of a system are typically expressed in temporal logic or using automata as observers. Two main classes of properties are *safety* properties that, intuitively, express that nothing bad ever happens, and *liveness* properties that assert that something good eventually happens. The third step is the application of the verification algorithm itself to decide whether the properties hold over the model of the system or not; in the latter case, model checking generates a counter-example exhibiting a run of the system that violates the property.

Beyond *verification*, which compares two formal objects, the model should also be *validated* to ensure that it faithfully represents the system under development. One approach to validation is to decide healthiness properties of the model (for example, ensure that each component action can occur in a system run), and model checking is again useful here. In general, it is helpful to narrow the gap between the system description and its formal model, for example by writing a model in a high-level executable language or in a notation familiar to designers such as UML. The chapters of this book, written by researchers active in the fields, present different possible approaches to the problems of modeling, verification and validation, as well as open research questions.

Chapter 1, written by Bernard Berthomieu, Florent Peres and François Vernadat, explains the analysis of real-time systems based on timed Petri nets. It illustrates the high expressiveness of that formalism and the different, complementary verification techniques that are implemented in the Tina tool.

In Chapter 2, Camille Constant, Thiery Jéron, Hervé Marchand and Vlad Rusu describe an approach that combines verification and conformance testing (on the actual implementation platform) of input/output symbolic transition systems. Disciplined approaches to testing are indeed a very valuable complement to formal verification for ensuring the correctness of an implementation. This is true in particular when the complexity of the models makes exhaustive verification impossible.

Chapters 3 and 4 are devoted to the presentation of model checking techniques. Starting with the canonical example of a lift controller, Stephan Merz presents the basic concepts and techniques of model checking for discrete state transition systems: temporal logics, the principles of model checking algorithms and their complexity, and strategies for mastering the state explosion problem. Patricia Bouyer and François Laroussinie focus on model checking for timed automata, the main semantic formalism for modeling real-time systems. They describe the formalism itself, timed modal and temporal logics, as well as some extensions and subclasses of timed automata. Finally, algorithms and data structures for the representation and verification of timed automata are introduced, and the modeling and verification environment Uppaal is described in some detail.

Using a model of an industrial drilling station as a running example, Radu Mateescu presents in Chapter 5 the functionalities of the CADP toolbox for modeling and verification. CADP is designed to model arbitrary asynchronous systems whose components run in parallel and communicate by message passing. The toolbox accepts models written in different formalisms, including networks of communicating automata or higher-level models written in Lotos. It implements a set of model transformations, simulation and verification algorithms, and offers the possibility to generate conformance tests for the implementation.

Chapter 6, written by Pascal Raymond, is devoted to the verification of programs written in the synchronous language Lustre with the help of the model checker Lesar. Synchronous languages enjoy ever more success for the development of reactive systems, of which real-time systems are a particular instance. Based on mathematical models of concurrency and time, synchronous languages provide a high-level abstraction for the programmer and are well-suited to formal verification.

In Chapter 7, Paul Caspi, Grégoire Hamon and Marc Pouzet go on to describe the language Lucid Synchrone that extends Lustre with constructs borrowed from functional languages, further augmenting expressiveness. The authors start by asking why synchronous languages are relevant for the design of critical systems. They give an account of the development of the Lucid language, and present in detail its primitives and the underlying theoretical concepts, illustrating them by several examples.

One of the most exciting developments over the past 15 years has been the emergence of techniques for the verification of probabilistic systems, intimately coupled

with work on stochastic processes carried out in the realm of performance evaluation. Probabilistic models are very useful because they add quantitative information above the non-deterministic representation of the behavior of system components and the environment. They can also be used to determine system parameters such as queue sizes, as a function of the desired failure guarantees. Marta Kwiatkowska, Gethin Norman, David Parker and Jeremy Sproston lay the bases in Chapter 8 by defining probabilistic timed automata and extending the model checking algorithms for ordinary timed automata to handle probabilistic models. The case study of the IEEE FireWire Root Contention Protocol illustrates the application of these techniques. In Chapter 9, Serge Haddad and Patrice Moreaux give an overview of verification techniques for probabilistic systems: discrete and continuous-time Markov chains, stochastic Petri nets, Markov decision processes and associated temporal logics. They also cover some of the main tools used in this domain, including GreatSPN, ETMCC and Prism.

Chapter 10, written by Marius Bozga, Susanne Graf, Laurent Mounier and Iulian Ober, presents the IF toolset, a tool environment for modeling and verifying real-time systems centered around a common internal description language based on communicating timed automata. User-level specifications written in languages such as SDL or UML are translated into this internal representation and can be subject to analysis using algorithms of static analysis, reduction and model checking. An extended case study from the aerospace domain based on joint work with EADS concludes the chapter.

Chapter 11, written by Anne-Marie Déplanche and Sébastien Faucou, is dedicated to the architecture description language AADL, originally designed and standardized for the avionic and aerospace domains, but which is an excellent candidate for arbitrary real-time systems. Architectural descriptions can serve as a reference for all actors involved in system design; they contain the information needed for simulation, formal verification and testing. The authors examine the specific requirements for describing real-time systems and then present the AADL and its support tools. Their use is illustrated with the help of a case study of a closed-loop control system.

We would like to express our gratitude to all of the authors for the time and energy they have devoted to presenting their topic. We are also grateful to ISTE Ltd. for having accepted to publish this volume and for their assistance during the editorial phase.

We hope that you, the readers of this volume, will find it an interesting source of inspiration for your own research or applications, and that it will serve as a reliable, complete and well-documented source of information on real-time systems.

Stephan MERZ and Nicolas NAVET
INRIA Nancy Grand Est and LORIA
Nancy, France

Chapter 1

Time Petri Nets – Analysis Methods and Verification with TINA

1.1. Introduction

Among the techniques proposed to specify and verify systems in which time appears as a parameter, two are prominent: Timed Automata (see Chapter 4) and Time Petri nets, introduced in [MER 74].

Time Petri nets are obtained from Petri nets by associating two dates $\min(t)$ and $\max(t)$ with each transition t. Assuming t became enabled for the last time at date θ, t cannot fire (cannot be taken) before the date $\theta+\min(t)$ and must fire no later than date $\theta+\max(t)$, except if firing another transition disabled t before then. Firing a transition takes no time. Time Petri nets naturally express specifications "in delays". By making explicit the beginnings and ends of actions, they can also express specifications "in durations"; their applicability is thus broad.

We propose in this chapter a panorama of the analysis methods available for Time Petri nets and discuss their implementation. These methods, based on the technique of state classes, were initiated in [BER 83, BER 91]. State class graphs provide finite abstractions for the behavior of bounded Time Petri nets. Various abstractions have been proposed in [BER 83, BER 01, BER 03b], preserving various classes of properties. In this chapter, we will discuss in addition the practical problem of the verification of formulae (*model checking*) of certain logics on the graphs of state classes available. Using these techniques requires software tools, both for the construction of the state

---

Chapter written by Bernard BERTHOMIEU, Florent PERES and François VERNADAT.

space abstractions and for the actual verification of the properties. The examples discussed in this chapter are handled with the tools available in the `Tina` environment [BER 04].

The basic concepts of Time Petri nets are reviewed in section 1.2. Sections 1.3 to 1.5 introduce various state class graph constructions, providing finite abstractions of the infinite state spaces of Time Petri nets. Sections 1.3 and 1.4 present the constructions preserving the properties of linear-time temporal logics such as LTL; in addition to traces, the construction of section 1.3 preserves markings while that exposed in section 1.4 also preserve states (markings and temporal information). Section 1.5 discusses preservation of the properties expressible in branching-time temporal logics. Section 1.6 discusses the analysis of firing schedules and presents a method to characterize exactly the possible firing dates of transitions along any finite sequence. The toolbox `Tina`, implementing all state space abstractions reviewed and a model checker, is presented in section 1.7. The following sections are devoted to the verification of LTL formulae on the graphs of state classes. Section 1.8 presents the logic selected, $SE{-}LTL$, an extension of the LTL logic, and the implementation of a verifier for that logic, the module `selt` of `Tina`. Section 1.9 discusses two application examples and their verification.

1.2. Time Petri nets

1.2.1. *Definition*

Let $\mathbf{I}^+$ be the set of non-empty real intervals with non-negative rational endpoints. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left endpoint and $\uparrow i$ its right endpoint (if i is bounded) or ∞ (otherwise). For all $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ denotes the interval $\{x - \theta \mid x \in i \wedge x \geqslant \theta\}$.

DEFINITION 1.1.– *A* Time Petri net *(TPN for short) is a tuple* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s\rangle$*, in which* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0\rangle$ *is a Petri net and* $I_s : T \rightarrow \mathbf{I}^+$ *is a function called the* Static Interval *function.*

Application I_s associates a temporal interval $I_s(t)$ with each transition of the net. The rationals $Eft_s(t) = \downarrow I_s(t)$ and $Lft_s(t) = \uparrow I_s(t)$ are called the static earliest firing time, and static latest firing time of transition t, respectively. A Time Petri net is represented in Figure 1.1.

1.2.2. *States and the state reachability relation*

DEFINITION 1.2.– *A state of a Time Petri net is a pair* $e = (m, I)$ *in which* m *is a marking and* $I : T \rightarrow \mathbf{I}^+$ *a function that associates a temporal interval with each transition enabled at* m*.*

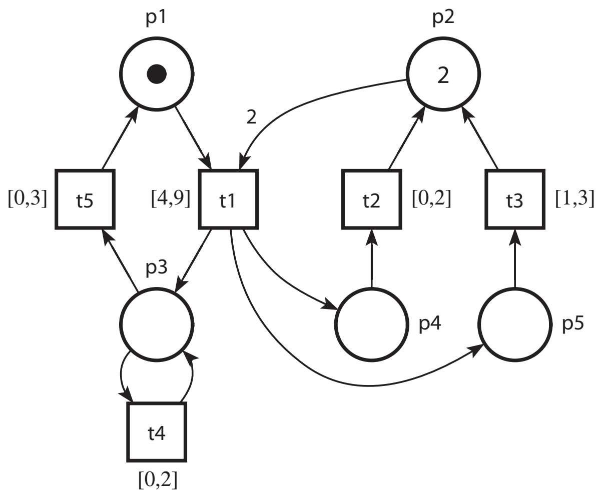


Figure 1.1. *A Time Petri net*

The initial state is $e_0 = (m_0, i_0)$, where I_0 is the restriction of I_s to the transitions enabled at m_0. Any transition enabled must fire in the time interval associated with it.

Firing t at date θ from $e = (m, I)$ (or, equivalently, waiting θ units of time then firing t instantly) is thus allowed if and only if:

$$m \geqslant \mathbf{Pre}(t) \wedge \theta \in I(t) \wedge (\forall k \neq t)(m \geqslant \mathbf{Pre}(k) \Rightarrow \theta \leqslant \uparrow(I(k))).$$

The state $e' = (m', I')$ reached from e by firing t at θ is determined by:

1) $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$ (as in Petri nets).

2) For each transition k enabled at m':

$$I'(k) = \text{if } k \neq t \text{ and } m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k) \text{ then } I(k) \dot{-} \theta \text{ else } I_s(k).$$

Let us note by $\xrightarrow{t@\theta}$ the timed reachability relation defined above, and let $e \xrightarrow{t} e'$ stand for $(\exists\theta)(e \xrightarrow{t@\theta} e')$. A *firing schedule* is a sequence of timed transitions $t_1@\theta_1 \cdots t_n@\theta_n$. It is firable from e if the transitions in sequence $\sigma = t_1 \cdots t_n$ are successively firable at the relative dates they are associated with in the schedule. σ is called the *support* of the schedule. A sequence of transitions is firable if and only if it is the support of some firable schedule.

Let us note that, in Time Petri nets, and contrarily to Timed Automata, the elapsing of time can only increase the set of firable transitions from a state, but can in no case

reduce it. Relation $\xrightarrow{t}$ characterizes exactly the "discrete" behavior of a TPN (bisimilarity after abstraction of time) when interpreting time-elapsing as non-determinism. Alternatively, time-elapsing could be interpreted as for Timed Automata. In that case, we should add to the transitions of relation $\xrightarrow{t@\theta}$ those representing the elapsing of time, of form $(m, I) \xrightarrow{r} (m, I \dot{-} r)$ where, for any k, r is not larger than $\uparrow(I(k))$.

Finally, let us note that the concept of state introduced in this section associates exactly one temporal interval with each enabled transition, whether or not that transition is multi-enabled (t is multi-enabled at m if there is an integer $k > 1$ such that $m \geqslant k.\mathbf{Pre}(t)$). An alternative interpretation of multi-enabledness is discussed in [BER 01], in which several temporal intervals may be associated with transitions; this interpretation will be briefly discussed in section 1.3.4.1.

1.2.3. *Illustration*

The states can be represented by pairs (m,D), in which m is a marking and D is a set of vectors of dates called a firing domain. The ith projection of domain D is the firing interval $I(t_i)$ associated with the ith enabled transition. Firing domains can be described by systems of linear inequalities with one variable per enabled transition (noted like the transitions).

The initial state $e_0 = (m_0, D_0)$ of the net represented in Figure 1.1 is written:

$$\begin{aligned} m_0 &: p_1, p_2 * 2 \\ D_0 &: 4 \leqslant t_1 \leqslant 9 \end{aligned}$$

Firing t_1 from e_0 at relative time $\theta_1 \in [4, 9]$ leads to state $e_1 = (m_1, D_1)$ given by:

$$\begin{aligned} m_1 &: p_3, p_4, p_5 \\ D_1 &: 0 \leqslant t_2 \leqslant 2 \\ &\quad 1 \leqslant t_3 \leqslant 3 \\ &\quad 0 \leqslant t_4 \leqslant 2 \\ &\quad 0 \leqslant t_5 \leqslant 3 \end{aligned}$$

Firing t_2 from e_1 at relative time $\theta_2 \in [0, 2]$ leads to $e_2 = (m_2, D_2)$, where:

$$\begin{aligned} m_2 &: p_2, p_3, p_5 \\ D_2 &: \max(0, 1 - \theta_2) \leqslant t_3 \leqslant 3 - \theta_2 \\ &\quad 0 \leqslant t_4 \leqslant 2 - \theta_2 \\ &\quad 0 \leqslant t_5 \leqslant 3 - \theta_2 \end{aligned}$$

Since θ_2 may take any real value in $[0, 2]$, state e_1 admits an infinity of successors.

1.2.4. *Some general theorems*

A Petri net or Time Petri net is *bounded* if, for some integer b, the marking of each place is smaller than b. Let us recall an undecidability result.

THEOREM 1.1.– *The problems of marking reachability, of state reachability, of boundedness and of liveness are undecidable for Time Petri nets.*

Proof. It is shown in [JON 77] that the marking reachability problem for $TPNs$ is reducible to that, undecidable, of the termination of a two-counter machine. Undecidability of the other problems follows.

Representing the behavior of a Time Petri net by its state reachability graph (as the behavior of a Petri net is represented by its marking reachability graph) is in general impossible: the transitions being able to fire at any time in their firing interval, states typically admit an infinity of successors. The purpose of the state classes defined thereafter are precisely to provide finite representation for this infinite state space, when the network is bounded, by grouping certain sets of states. However, there are two remarkable subclasses of Time Petri nets admitting finite state graphs if and only if they are bounded. □

THEOREM 1.2.– *Consider a* TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, M_0, I_s\rangle$. *If all transitions* $t \in T$ *have static interval* $[0, \infty[$*, then the state graph of the net is isomorphic with the marking graph of the underlying Petri net.*

Proof (by induction). If each transition carries interval $[0, \infty[$, then the firing conditions for the transitions are reduced to that in Petri nets (without temporal constraints). In addition, the firing rule preserves the shape of firing intervals. □

THEOREM 1.3.– *Consider a* TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, M_0, I_s\rangle$. *If* I_s *associates with each transition a punctual interval (reduced to one point), then:*
(i) the state graph of the net is finite if and only if the net is bounded;
(ii) if, in addition, all transitions bear equal static firing intervals, then its state graph is isomorphic with the marking graph of the underlying Petri net.

Proof (by induction). (i) By the firing rule, if all static intervals are punctual, then a state has at most as many successor states as the number of transitions enabled at it. In addition, the firing rule preserves the punctual character of firing intervals. For (ii), note that the firing condition reduces then to that in Petri nets. □

Theorems 1.2 and 1.3 make it possible to interpret Petri nets as particular Time Petri nets, in various ways. The most frequent interpretation is to regard them as Time Petri nets in which each transition carries static interval $[0, \infty[$.

1.3. State class graphs preserving markings and LTL properties

1.3.1. *State classes*

The set of states of a Time Petri net may be infinite for two reasons: on one hand because a state may admit an infinity of successors and, on the other hand, because a TPN can admit schedules of infinite length going through states whose markings are all different. The second case will be discussed in section 1.3.3. To manage the first case, we will gather certain sets of states into state classes. Several grouping methods are possible; the construction reviewed in this section is that introduced in [BER 83, BER 91].

For each firable sequence σ, let us note by C_σ the set of states reached from the initial state by firing schedules of support σ. For any such set C_σ, let us define its marking as that of the states it contains (all these states have necessarily the same marking) and its firing domain as the union of the firing domains of the states it contains. Finally, let us note by $\cong$ the relation satisfied by two sets C_σ and $C_{\sigma'}$ when they have equal markings and equal firing domains. If two sets of states are related by $\cong$, then any schedule firable from some state in one of these sets is firable from some state in the other set.

The graph of state classes of states of [BER 83], or SCG, is the set of sets of states C_σ, for any firable sequence σ, considered modulo relation $\cong$, and equipped with the transition relation: $C_\sigma \xrightarrow{t} X$ if and only if $C_{\sigma.t} \cong X$. The initial state class is the equivalence class of the singleton set of states containing the initial state.

The SCG is built as follows. State classes are represented by pairs (m, D), where m is a marking and D a firing domain described by a system of linear inequalities $W\underline{\phi} \leqslant \underline{w}$. The variables $\underline{\phi}$ are bijectively associated with the transitions enabled at m. The equivalence $(m, D) \cong (m', D')$ holds if and only if $m = m'$ and $D = D'$ (i.e. the systems describing D and D' have equal solution sets).

ALGORITHM 1.1.– *Construction of the SCG, state classes*
For any firable sequence σ, let L_σ be the class computed as explained below. Compute the smallest set C of classes including L_ϵ and, whenever $L_\sigma \in C$ and $\sigma.t$ is firable, then $(\exists X \in C)(X \cong L_{\sigma.t})$:

– *The initial class is $L_\epsilon = (m_0, \{Eft_s(t) \leqslant \underline{\phi}_t \leqslant Lft_s(t) \mid t \in T \wedge m_0 \geqslant \mathbf{Pre}(t)\})$.*

– *If σ is firable and $L_\sigma = (m, D)$, then $\sigma.t$ is firable if and only if:*

(i) $m \geqslant \mathbf{Pre}(t)$ (t is enabled at m) and

(ii) the system $D \wedge \{\underline{\phi}_t \leqslant \underline{\phi}_i \mid i \in T \wedge i \neq t \wedge m \geqslant \mathbf{Pre}(i)\}$ is consistent.

– *If $\sigma \cdot t$ is firable, then $L_{\sigma.t} = (m', D')$ is computed as follows from $L_\sigma = (m, D)$:*

$m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$,

D' obtained as follows:

(a) the above firability constraints (ii), of t from L_σ, are added to D;

(b) for each k enabled at m', a new variable $\underline{\phi}'_k$ is introduced, obeying:

$\underline{\phi}'_k = \underline{\phi}_k - \underline{\phi}_t$, *if $k \neq t$ and $m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k)$,*

$Eft_s(k) \leqslant \underline{\phi}'_k \leqslant Lft_s(k)$, *otherwise;*

(c) variables $\underline{\phi}$ are eliminated.

L_σ is the equivalence class by relation $\cong$ of the set C_σ of states reached from s_0 by firing schedules of support σ. Equivalence $\cong$ is checked by putting the systems representing firing domains into canonical forms. These systems are systems of differences. Computing canonical form for them reduces to a problem of "all-pairs shortest path", solved in polynomial time using, for example, Floyd/Warshall's algorithm.

Remark: two sets C_σ and $C_{\sigma'}$ can be equivalent by $\cong$ while having different contents in terms of states. The notation of the state classes by a pair (m, D) canonically represents an equivalence class of sets of states for relation $\cong$, but we cannot tell from such a class if it contains some given state.

1.3.2. *Illustration*

As an illustration, let us build some state classes of the TPN represented in Figure 1.1. The initial class c_0 is described in the same way as the initial state e_0 (see section 1.2.3). Firing t_1 from c_0 leads to a class c_1 described like state e_1 (since the target state does not depends on the time at which t_1 fired). Firing t_2 from c_1 leads to $c_2 = (m_2, D_2)$, with $m_2 = (p_2, p_3, p_5)$ and D_2 computed in three steps:

(a) D_1 is augmented with the firability conditions for t_2, given by system:

$$t_2 \leqslant t_3$$
$$t_2 \leqslant t_4$$
$$t_2 \leqslant t_5$$

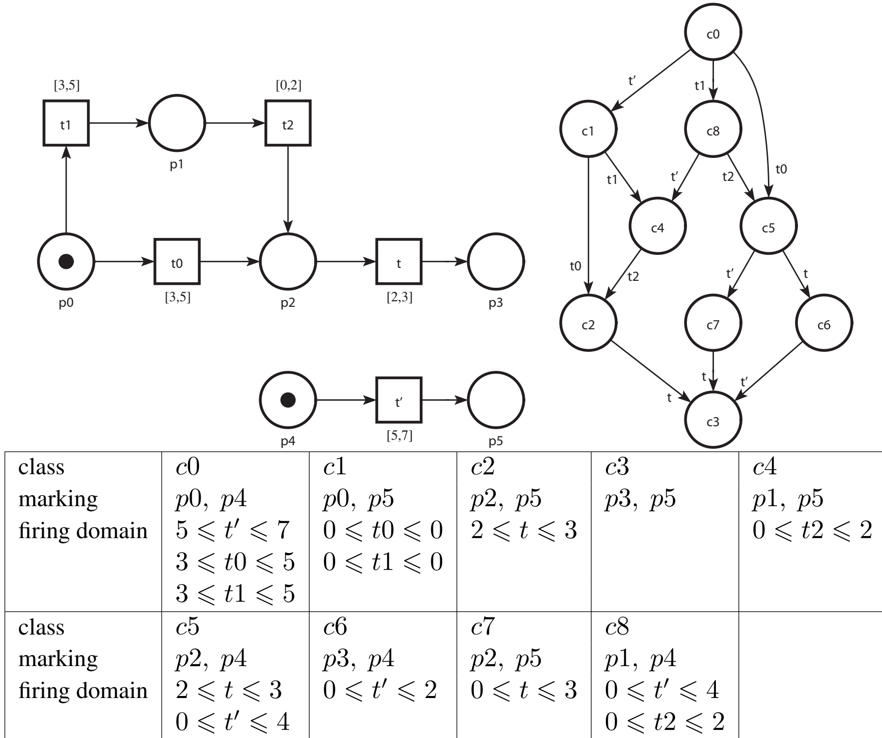
(b) No transition is newly enabled, and t_3, t_4, t_5 remain enabled while t_2 fires. So we simply add equations $t'_i = t_i - t_2$, for $i \in \{3, 4, 5\}$.

(c) The variables t_i are eliminated, yielding system:

$$0 \leqslant t'_3 \leqslant 3 \quad t'_4 - t'_3 \leqslant 1$$
$$0 \leqslant t'_4 \leqslant 2 \quad t'_5 - t'_3 \leqslant 2$$
$$0 \leqslant t'_5 \leqslant 3$$

The graph of state classes of the net in Figure 1.1 admits 12 classes and 29 transitions, which can be found in [BER 01].

Figure 1.2 shows another Time Petri net, which will be used to compare the various state class graph constructions, together with its SCG graph.



class marking firing domain	$c0$ $p0,\ p4$ $5 \leqslant t' \leqslant 7$ $3 \leqslant t0 \leqslant 5$ $3 \leqslant t1 \leqslant 5$	$c1$ $p0,\ p5$ $0 \leqslant t0 \leqslant 0$ $0 \leqslant t1 \leqslant 0$	$c2$ $p2,\ p5$ $2 \leqslant t \leqslant 3$	$c3$ $p3,\ p5$	$c4$ $p1,\ p5$ $0 \leqslant t2 \leqslant 2$
class marking firing domain	$c5$ $p2,\ p4$ $2 \leqslant t \leqslant 3$ $0 \leqslant t' \leqslant 4$	$c6$ $p3,\ p4$ $0 \leqslant t' \leqslant 2$	$c7$ $p2,\ p5$ $0 \leqslant t \leqslant 3$	$c8$ $p1,\ p4$ $0 \leqslant t' \leqslant 4$ $0 \leqslant t2 \leqslant 2$	

Figure 1.2. *A TPN and its SCG. The variable $\underline{\phi}_t$ is written t*

1.3.3. *Checking the boundedness property on-the-fly*

It remains to examine the conditions under which the set of state classes of a TPN is finite. Let us recall that a Petri net or Time Petri net is bounded if the marking of any place admits an upper bound; boundedness implies finiteness of the set of reachable markings. It was proven in [BER 83] that the set of firing domains of a Time Petri net is finite; its SCG is thus finite if and only if the net is bounded. This property is undecidable for arbitrary TPN (see Theorem 1.1), but sufficient conditions can be applied. The simplest such sufficient condition is that the underlying Petri net is bounded (which is decidable), but weaker conditions can be stated. The following theorem reviews some of them.

THEOREM 1.4.– *[BER 83] A Time Petri net is bounded if its SCG does not contain a pair of classes $c = (m, D)$ and $c' = (m', D')$ such that:*

i) c' is reachable from c,

ii) $m' \ngeqslant m$,

iii) $D' = D$,

iv) $(\forall p)(m'(p) > m(p) \Rightarrow m'(p) \geqslant \max_{(t \in T)}\{\mathbf{Pre}(p,t)\})$.

Properties (i) to (iv) are necessary for the boundedness property, but not sufficient. This theorem, for example, ensures that the nets in Figures 1.1, 1.3 (left) and 1.3 (middle) are bounded, but it cannot be used to prove that the net represented in Figure 1.3 (right) is bounded, even though this net only admits 48 classes. If we omit (iv), then boundedness of the net in Figure 1.3 (middle) cannot be proven anymore. Omitting (iii), in addition, we cannot infer boundedness for the net in Figure 1.3 (left). The condition obtained then ((i) and (ii)) is similar to the boundedness condition for the underlying (untimed) Petri net [KAR 69].

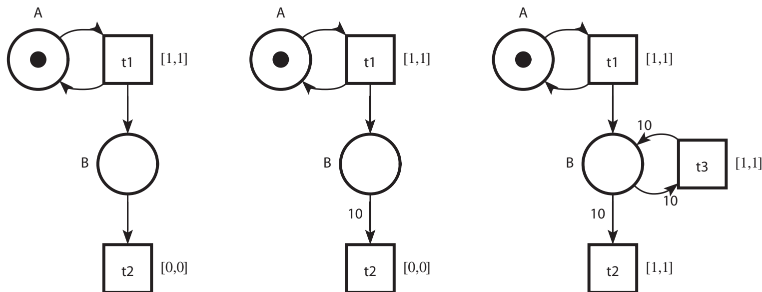


Figure 1.3. *Three bounded Time Petri nets*

1.3.4. *Variations*

1.3.4.1. *Multiple enabledness*

A transition t is multi-enabled at a marking m if there is some integer $k > 1$ such that $m \geqslant k.\mathbf{Pre}(t)$. In the classes of the graph introduced in section 1.3, each enabled transition is associated with exactly one temporal variable, whether or not the corresponding transition is multi-enabled; the various enabling dates for a transition are systematically identified with the largest of these dates.

In some applications, this interpretation may be found to be too restrictive; we may want to distinguish the different enabledness instances. The issue is investigated in [BER 01], where several operational interpretations of multi-enabledness are discussed. We can naturally see the enabledness instances as independent, or as ordered

according to their creation dates. In any case, Algorithm 1.1 is easily adapted to produce state class graphs "with multi-enabledness" [BER 01].

1.3.4.2. *Preservation of markings (only)*

It is possible to compact further the state class graph SCG by not storing a class when it is included in an already built class. More precisely, let $L_\sigma = (m, D)$ and $L_{\sigma'} = (m', D')$ be two classes, and $L_\sigma \sqsubseteq L_{\sigma'}$ if and only if $m = m'$ and $D \subseteq D'$. Then, rather than proceeding as in Algorithm 1.1, we build a set C of classes such that, when $L_\sigma \in C$ and $\sigma \cdot t$ is firable, then $(\exists X \in C)(L_{\sigma \cdot t} \sqsubseteq X)$.

Intuitively, if such a class X exists, any schedule firable from a state of $L_{\sigma \cdot t}$ is also firable from a state in X. We will thus not find new markings by storing $L_{\sigma \cdot t}$. Of course, this construction does not preserve the firing sequences, and therefore LTL properties; it only preserves markings, but it typically produces smaller graphs. This construction is convenient when correctness requirements can be reduced to marking reachability properties or to absence of deadlocks.

1.4. State class graphs preserving states and LTL properties

As already mentioned, the classes built using Algorithm 1.1 canonically represent equivalence classes of sets of states by $\cong$, but not the sets C_σ defined in section 1.3.1. A consequence is that the SCG cannot be used to prove or disprove reachability of a particular state of a TPN.

The *strong state classes* (also called *state zones* by some authors) reviewed in this section, introduced in [BER 03b], coincide exactly with these sets of states C_σ. The graph of strong state classes ($SSCG$ for short) preserves LTL properties and states in a way we will make clear.

1.4.1. *Clock domain*

To build the SSCG, it is necessary to represent the sets of states C_σ in a canonical way. An adequate representation is provided by clock domains.

With any firing schedule we can associate a clock function γ as follows: with any transition enabled after firing the schedule, function γ associates the time elapsed since that transition was last enabled. Clock functions can also be seen as vectors $\underline{\gamma}$, indexed by the enabled transitions.

The set of states described by a marking m and a clock system $Q = \{G\underline{\gamma} \leqslant \underline{g}\}$ is the set $\{(m, \phi(\underline{\gamma})) \mid \underline{\gamma} \in Sol(Q)\}$, where $Sol(q)$ is the solution set of system Q, and the firing domain $\phi(\underline{\gamma})$ is the set of solutions in $\underline{\phi}$ of system:

$$\underline{0} \leqslant \underline{\phi}\ ,\ \underline{e} \leqslant \underline{\phi} + \underline{\gamma} \leqslant \underline{l} \ \text{ where } \ \underline{e}_k = Eft_s(k) \text{ and } \underline{l}_k = Lft_s(k).$$