Object-Z/TCOZ and Timed Automata: Projection and Integration

HAO PING

(B.Sc. Huazhong University of Science and Technology, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

I am deeply indebted to my advisor, Professor Dong Jin Song, for his guidance, insight and encouragement throughout the course of my doctoral program and for his careful reading of and constructive criticisms and suggestions on drafts of this thesis and other works.

I owe thanks to Dr. Qin Shengchao, Professor Wang Yi, Professor Roger Duke, and Dr. Ana Cavalcanti for their suggestions and help on this thesis and other work. and I also owe thanks to Zhang Xian, Li Yuanfang, Sun Jun, Sun Jing, Wang Hai and other office-mates and friends for their help, discussions and friendship.

I would like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings. Their valuable comments have contributed to the clarification of many of the ideas presented in this thesis.

This study received financial support from the National University of Singapore. The School of Computing also provided the finance for me to present papers in several conferences overseas. For all this, I am very grateful.

I sincerely thank my parents Hao Xianqing and Yan Xiangrong and my sister Hao Xiaoping for their love and encouragement in my years of study. Finally, I wish to express my love and thanks to my husband Zhang Songhua for his continuing love, patience, and understanding.

Contents

1 Introduction **1**

1.1 Motivation and goals 1

1.2 Outline of This Thesis 7

1.3 Publications from this Thesis 7

2 OZ/TCOZ and Timed Automata **9**

2.1 Object-Z 10

2.2 Timed Communicating Sequential Process 12

2.3 Timed Communicating Object-Z 14

2.4 Timed Automata 21

3 Composable TA patterns **25**

3.1 Z definition of Timed Automata 26

3.2 TA patterns 28

3.3 Generating New Patterns 36

3.4 Guidelines for TA Design Using Patterns 37

3.5 Discussion and Conclusion 40

4 Projection from TCOZ to TA **43**

4.1 Introduction 44

4.2 Mapping Rules 45

4.3 Correctness 48

4.4 An Example: Railroad Crossing System 62

4.5 Tool Support 69

4.6 Conclusion 73

5 Case Study: Multi-terminal Railcar System **75**

5.1 TCOZ Model of MRS 77

5.2 Translation 84

5.3 Model-checking MRS 87

5.4 Conclusion 89

6 Integrating Object-Z with Timed Automata **91**

6.1 Introduction 92

6.2 Overview on Combining Object-Z and TA 95

6.3 Design Decisions 102

6.4 Composition and Communication 106

6.5 Operation Semantics 113

6.6 An Example: Electronic Key System 118

6.7 Conclusion 123

7 OZTA Semantics **125**

7.1 Introduction 126

7.2 The Syntax of OZTA 126

7.2.1 An example : Shunting Game 129

7.3 The Semantics of OZTA 131

7.3.1 The Automata Model 132

7.3.2 The Semantics of OZTA Automata with Patterns 135

7.3.3 The Semantics of Class 145

7.4 Conclusion 146

8 OZTA Tool Support and Case Study 147

8.1 Introduction 148

8.2 Modeling 149

8.3 Checking 150

8.3.1 Syntax and Type Checker 150

8.3.2 OZTA to UPPAAL 151

8.4 Case Study: A Frog Puzzle Game 152

8.4.1 Design of OZTA Models 153

8.4.2 Syntax and Type Check 155

8.4.3 Model Checking Using UPPAAL 156

8.4.4 Generation LaTeX Document 157

8.5 Conclusion 158

9 Conclusion and Future Directions 161

9.1 Summary and Contributions 162

9.2 Future Work 167

A 183

A.1 TCOZ Notation 183

A.2 Type Inference Rule 185

A.3 Screenshots of **HighSpec** 188

Summary

The design of complex real-time systems requires not only powerful mechanisms for modelling various aspects of a complex system but also tool support for developing the models, especially tool support for verifying the established models. While there are a variety of formal techniques and tools that have been proposed in the literature and each may has its unique strength in describing one or some aspects of a complex system, it has been realized that no single notation will ever be suitable to address all aspects of a complex system with tool support. The state-of-the-art formal modelling techniques, Z family languages OZ/TCOZ and state-machine based techniques Timed Automata, both of them have their unique strength and weakness on specifying high-level abstracted models for complex systems. This thesis investigates the possible links between the modelling techniques OZ/TCOZ and TA and research how to lend the strengths of different techniques to each other or how to integrate the strengths of different techniques together so that they can be utilized coherently for building and verifying models of complex real-time systems in a unified framework. Firstly, a set of composable timed automata patterns (reminiscent of 'design patterns' in object-oriented modelling) are defined based on TCOZ process constructs. These composable timed automata patterns not only provide a proficient interchange media for transforming TCOZ specifications into TA designs, which supports one possible engineering development process: TCOZ for high-level requirement specifications, then TA for design and timing analysis; but also provide a generic reusable framework for developing real-time systems in

TA alone. Secondly, based on the patterns, a set of transformation rules from TCOZ to TA are defined so that one possible engineering process for modelling and checking of complex real-time system can be supported: TCOZ for building high-level models and TA's tool support to be reused for verification of the models. We also investigate the semantic equivalence issue between TCOZ processes and timed automata and provide a proof for the correctness of the transformation. Lastly, inspired by this part of work, an interesting question is that: can we integrate Object-Z and Timed Automata directly? In this way, not only the wonderful tool support of TA can be reused straightforward, but also the timed composable patterns now can be directly utilized for systematic TA designs. Thus, rather than taking the transformation point of view, we also developed a novel integrated formal language which combines Object-Z with TA. The advantage of this approach lies in that by replacing TCSP with TA in TCOZ, the wonderful tool support of TA can be reused straightforward, moreover, comparing to CSP/TCSP which provide a fix topology for communications, this new formalism OZTA is injected with a novel concept of partial and sometime synchronization to capture various synchronization scenarios. Meanwhile, the OZTA notation is enhanced by introducing the set of timed patterns as language constructs to specify the dynamic and timing features of complex real-time systems in a systematic way. We also present a semantic model of OZTA in Unifying Theories of Programming which provides the semantic foundation for language understanding, reasoning and tool construction. Based on the semantic model, we constructed **HighSpec** an interactive system which can support editing, syntax and type checking of OZTA models as well as transforming

OZTA models into TA models so that we can utilize TA model-checkers, e.g., UPPAAL, for simulation and verification.

In summary, we built up the linkages of different modelling techniques, Object-Z/TCOZ and Timed Automata, and established a powerful unified framework using two alternative approaches for modelling and checking of complex real-time systems.

List of Figures

2.1 Timed Queue 22

4.1 Simulation 67

4.2 TCOZ to UPPAAL diagram 69

5.1 Terminal Panel 85

5.2 Car Handler 86

5.3 Car Panel 86

5.4 Controller 87

5.5 Simulation 89

5.6 Verification 90

6.1 Model 1 103

6.2 Model 2 104

6.3 Model 3 104

6.4 Model 4 105

6.5 The Automaton $s1$ 107

6.6 Timed Automata U, V and W 110

6.7 Timed Automata $U1$, $V1$ and $W1$ 111

6.8 Timed Automata $U2$, $V2$ and $W2$ 111

6.9 Timed Automata $U3$, $V3$ and $W3$ 112

6.10 Timed Automata $U4$ and $V4$ 113

6.11 Timed Automata $U5$ and $V5$ 113

7.1 The Shunting Game 132

8.1 Overview of **HighSpec** 148

8.2 Class Diagram of the type checker 151

8.3 Frog Puzzle Model in UPPAAL 156

A.1 The Main Window of **HighSpec** 189

A.2 The Object-Z Editing part 189

A.3 The Timed Automaton Editing Part 1: state definition 190

A.4 The Timed Automaton Editing Part 2: transition definition 190

A.5 The Timed Automaton Editing Part 3: pattern library 190

A.6 The Timed Automaton Editing Part 4: timing parameter of patterns 191

A.7 The Timed Automaton Editing Part 5: relating Object-Z operation with atomic states in the timed automaton 191

A.8 The Model of Frog Puzzle Game Example: the default abstracted automaton with recursive pattern as the outmost layer and external choice as its inside layer 191

A.9 The Model of Frog Puzzle Game Example 192

Chapter 1

Introduction

1.1 Motivation and goals

Formal methods are of global concern in software engineering. A formal method [32, 8, 9] in software development provides a formal language for describing a software artifact (e.g. specifications, designs, source code) such that formal proofs are possible, in principle, about the properties of the artifact so expressed. The use of formal methods can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected.

Over the last few decades, a variety of formal modelling techniques have been proposed in the literature. For example, VDM [2], Z [70], Object-Z [66], and B [3] are state-oriented formalisms; CSP [39, 38], Timed CSP [61], and CCS [43] are

process-oriented formalisms, and Timed Automata (TA) [4] and Petri-net [7] are state-machine based formalisms.

While each formal specification technique may have its unique strength in describing the system or the aspects of the system which it fits well with, it has been realized that no single notation will ever be suitable to address all aspects of a complex system. As a result, combining different formal notations to capture various aspects of a complex system has become a recent research trend in software specification and design [6, 29, 12]. A popular approach has been the blending of Z/Object-Z with either CSP or CCS. For example, Fischer [28] and Smith [67] suggested independently a combination of Object-Z with CSP. Mahony and Dong [50] proposed Timed Communicating Object-Z (TCOZ), which combines Timed CSP and Object-Z. Comparing to Fischer and Smith's work, TCOZ is more novel in that building on Timed CSP, it includes primitives for treating timing issues. Timed CSP has strong process control modelling capabilities. Object-Z has strong data and state modelling capabilities. The two languages complement each other in their expressiveness. Building on the strengths of the Object-Z and Timed CSP notations, TCOZ is capable of modelling state, process and timing aspects of complex systems, thus it is well suited for presenting more complete and coherent requirement models for real-time complex systems.

However, the design of complex real-time systems usually requires not only powerful mechanisms for modelling various aspects of a complex system, but also tool support for building up the models, especially tool support for verifying the estab-

lished models.

One problem of integrated formal methods, is that very few of them has direct tool support, even just for editing, not to say verification. This is mainly because high-level specification languages such as Object-Z/TCOZ usually contain various kinds of abstracted system information, which makes the implementation of a verification tool from scratch difficult.

On the other hand, although the necessity of tool support for formal methods is widely accepted and some tools have been proposed and developed, such as Z/EVES [14], Alloy [54], PVS [56], SPIN [42], FDR [49], UPPAAL [58] and so on, one problem of the existing tool support for integrated formal methods is that much of them fails to exploit the advantages that formality brings. For example, Timed Automata (TA) [4] is powerful in designing real-time models with multiple clocks and has well developed automatic tool support for verification, e.g., UPPAAL [48], KRONOS [16], TEMPO [69], RED [74] and Timed COSPAN [72]. One weakness of TA, however, is the lack of high-level composable graphical patterns to support the systematic design of complex real-time systems.

This motivates us to research how integrating formal methods can lend the strengths of different techniques to each other and facilitate tool support for the development process. The Z family languages OZ/TCOZ and the state-machine based modelling techniques TA lie at each end of the spectrum of formal methods. Both of them are state-of-the-art modelling techniques. OZ/TCOZ is good at specifying high-level abstracted models for complex systems, while TA is good at designing low-level

abstracted timed models with multiple clocks but with well-developed tool support. Thus, it is of great interest and importance to investigate the possible links between OZ/TCOZ and TA so that they can be utilized coherently for building and verifying models of complex real-time systems in the development process.

The objective of this thesis is to construct a unified framework for using OZ/TCOZ and TA together for modelling and checking complex real-time systems in the development process, so that:

- not only powerful mechanisms are available for specification, to capture various aspects of a system such as data structure, concurrency, and real-time dynamic behaviors,
- but also tool support can be provided for design and verification either by integrating existing tools or exploiting the particular languages combined.

To construct such a unified framework, we will present two alternative approaches, i.e., a projection approach based on integrating existing notations and tools; and an integration approach based on creating a new combined language with direct tool support.

For the projection approach, we propose to use TCOZ for high-level requirement specification and then project TCOZ models to TA models so that TA's tool support UPPAAL can be reused for verification and analysis of the properties of TCOZ models such as timing issues. In this framework, we investigate the strengths and links between TCOZ and TA so that the two modelling techniques can benefit

each other. This leads us to an interesting research result, i.e., timed composable patterns (reminiscent of 'design patterns' in object-oriented modelling). These timed composbale patterns not only provide a proficient interchange media for projecting TCOZ specifications to TA designs for timing analysis, but also provide a generic reusable framework for systematically developing real-time systems in TA alone. Another important issue that needs to be addressed in language projection/translation is the consistency between the original TCOZ models and the translated TA models. For this, we will provide a correctness proof to demonstrate that our projection from TCOZ to Timed Automata is complete and sound. One interesting question may arise from this part of our work: can we integrate Object-Z and Timed Automata directly? In this way, not only the tool support of TA can be reused straightaway, but also the timed composable patterns now can be directly utilized for systematic TA designs. This motivate us to further our research on an integration approach.

For the integration approach, we propose a new integrated modelling language OZTA by combining Object-Z and Timed Automata. Besides its advantage of direct tool support from TA over TCOZ, comparing to CSP/TCSP which provide a fixed topology for communications, this new formalism OZTA is injected with a novel concept of partial and sometime synchronization to capture various synchronization scenarios. To achieve an effective combination of Object-Z and TA, the following issues will be explored:

- how to semantically and syntactically link the key language constructs so

that the two notations can be used in a cohesive way;

- how to clearly separate system functionality aspects from time control behavior patterns, so that separate tools can later be applied to check the related system properties;
- how to consistently unify the composition techniques from both Object-Z (class instantiation) and TA (automaton product) so that subsystem models can be easily and meaningfully composed;
- and how to systematically develop the communication mechanisms so that various concurrent interactions between system components can be precisely captured.

For this new integrated modelling technique OZTA, we also provide an operational semantics model using the Unified Theory of Programming [40]. Based on the semantic foundation, an interactive system **HighSpec** is developed to provide the tool support for developing OZTA models, and the TA's model-checker UPPAAL is integrated with **HighSpec** for verification.

In summary, these two approaches, the projection approach and the integration approach, represent two generic methods that can be adopted to take full advantage of various reciprocal formalisms altogether to build and verify models of complex systems.

1.2 Outline of This Thesis

The thesis is structured into 9 chapters. Chapter 2 covers the modelling languages and techniques used throughout the thesis. Chapter 3 investigates the links between TCOZ and Timed Automata and defines a set of composable timed patterns. Chapter 4 introduces the projection from TCOZ to Timed Automata. Chapter 5 demonstrates the projection using a railcar system. Chapter 6 proposes the new integrated language OZTA. chapter 7 further enhances OZTA and gives the semantic model for OZTA. Chapter 8 introduces an interactive system **HighSpec** we developed for building and checking of OZTA models and demonstrates the use of OZTA language and its tool support. Chapter 9 gives the conclusion of the thesis and future works.

1.3 Publications from this Thesis

Most chapters of the thesis have been accepted in international refereed conference proceedings or journals. Part of the work in Chapter 3, Chapter 4 and Chapter 5 has been presented at International Journal on Software Engineering and Knowledge Engineering [23] and the Sixth International Conference on Formal Engineering Methods ICFEM'04 (Nov 8-12, 2004, Seattle) [24], and a journal paper has been submitted to Transactions on Software Engineering [44]. The work in Chapter 6 has been presented at the 10th International Conference on Engineering of Complex Computer Systems ICECCS'05 (June 2005, Shanghai) [22]. The work in Chapter

7 has been presented at the Seventh International Conference on Formal Engineering Methods ICFEM'05 (1-4 November 2005, Manchester, UK) [25]. Chapter 8 has been presented at 28th International Conference on Software Engineering (ICSE'06), Research Demonstration (May 20-28, 2006. Shanghai, China) [26]. I also made partial contributions to other publications [78, 21] which are related to this thesis. They can be considered as side-stories to the impact of this thesis work.

Chapter 2

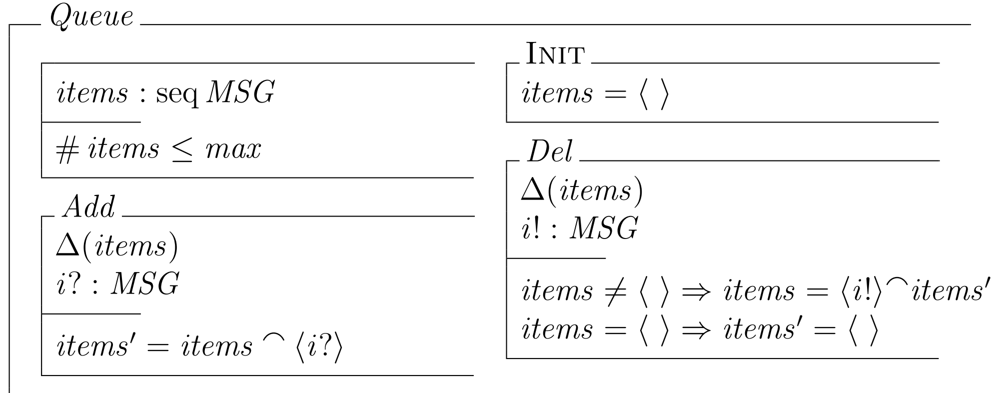
OZ/TCOZ and Timed Automata

In this chapter, we introduce the modelling techniques Object-Z, Timed Communicating Object-Z and Timed Automata.

2.1 Object-Z

Object-Z [66] is an extension of the Z [70] formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring.

The essential extension to Z given by Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class. Syntactically, a class definition is a named box. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas. To illustrate Object-Z, we consider a simple message queue system. The essential behaviors of this system is that it can receive a new message or remove a message. The message queue has a FIFO property.



The first schema in the left column of the class is called a state schema; it has an attribute *items* denoting a sequence of messages of the type *MSG*. The INIT schema describes the allowed initial values for the class attributes *items*. The remaining two schemas are operation schemas which describe the possible state change. The declaration parts of the operation schemas include a Δ- list of the primary attribute *items* whose value may change. By convention, no Δ-list means no attribute changes value. Note that '⁀' means concatenation. For sequences s and t, $s \frown t$ is the concatenation of s and t. It contains the elements of s followed by the elements of t.

Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Hence the class invariant holds at all times: in each possible initial state, and before and after each operation. In this example, the operation *Add* adds a given input *item*? to the end of the existing message sequence provided the sequence has not already reached its maximum size (an identifier ending in '?' denotes an input). The operation *Delete* outputs a value *item*! defined as one element of *items* and

reduces *items* by deleting the first one from the original queue (an identifier ending in '!' denotes an output).

The standard behavioral interpretation of Object-Z objects is as transition systems [65]. A behavior of a transition system consists of a series of state transitions each effected by one of the class operations. A *Queue* object starts with *items* empty then evolves by successively performing either *Add* or *Delete* operations. Operations in Object-Z are atomic, only one may occur at each transition, and there is no notion of time or duration. It is difficult to use the standard Object-Z semantics to model the real-time dynamic behaviors of a system composed by multi-threaded component objects whose operations have timing constraints.

2.2 Timed Communicating Sequential Process

Hoare's CSP [39] is an event based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization. Like CSP, Timed CSP adopts a symmetric view of process and environment. Events represent a cooperative synchronization between a process and its environment. Both the process and the environment may control the behavior of the other by *enabling* or *refusing* certain events and sequences of events.

The syntactic class of Timed CSP expressions is defined as follows:

$P ::= Stop \mid Skip \mid Run_{\Sigma}$

$\mid e \bullet t \rightarrow P(t) \mid e \xrightarrow{t}_1 P$

$\mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2$

$\mid P_1 \,|[\,\Sigma\,]|\, P_2 \mid P_1 \,|||\, P_2$

$\mid P_1;\ P_2 \mid P_1 \,\triangledown\, P_2 \mid P1 \triangleright\{d\}\ P2$

$\mid \textsc{Wait}[d] \mid P_1\ \triangledown\{d\}\ P_2 \mid \mu\, X \bullet P(X)$

Run_{Σ} is a process always willing to engage any event in Σ. *Stop* denotes a process that deadlocks and does nothing. A process that terminates immediately is written as *Skip*. A process which may participate in event e then act according to the process description P is written as $e \bullet t \rightarrow P(t)$. The (optional) timing parameter t records the time, relative to the start of the process, at which the event e occurs and allows the subsequent behavior P to depend on its value. The process $e \xrightarrow{t}_1 P$ delays the process P by t time units after engaging event e. The external choice operator ($\Box$) allows choice of behaviors according to what events are requested by the environment. Internal choice represents variation in behavior determined by the internal state of the process. The parallel composition of processes P_1 and P_2, synchronized on a common set of events Σ is written as $P_1 \,|[\,\Sigma\,]|\, P_2$. The sequential composition of P_1 and P_2, written as $P_1;\ P_2$, acts as P_1 until P_1 terminates by communicating a distinguished event $\checkmark$ and then proceeds to act as P_2. The interrupt process $P_1 \,\triangledown\, P_2$ behaves as P_1 until the first occurrence of an event in P_2, then the control passes to P_2. The timed interrupt process $P_1\ \triangledown\{d\}\ P_2$ behaves similarly, except that P_1 is interrupted as soon as d time units have elapsed. A process which allows no communications for a period of d time units, and then terminates

is written as $\textsc{Wait}[d]$. The timeout construct written as $P_1 \triangleright\{d\}\ P_2$ passes control to an exception handler P_2 if no event occurs in the primary process P_1 by d time units. Recursion is used to give finite representation of non-terminating processes. The process expression $\mu X \bullet P(X)$ describes processes which repeatedly act as $P(X)$.

In general, Timed CSP is superior to Object-Z as a means of describing process control. However, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal state. The syntactic treatment of internal state is complex and unwieldy, distracting strongly from the basically elegant treatment of the delay and timeout issues. Timed CSP has yet no standard support for state modelling in the form of mathematical toolkits and libraries, nor are there modular techniques for constructing and reasoning about complex internal state.

2.3 Timed Communicating Object-Z

Timed Communicating Object Z (TCOZ) [50] is essentially a blending of Object-Z with Timed CSP, for the most part preserving them as proper sub-languages of the blended notation. The essence of this blending is the identification of Object-Z operation specification schemas with terminating CSP processes. Thus operation

schemas and CSP processes occupy the same syntactic and semantic category; operation schema expressions can appear wherever processes appear in CSP and CSP process definitions can appear wherever operation definitions appear in Object-Z. In this section we briefly consider various aspects of TCOZ. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [50]. The formal semantics of TCOZ (presented in Z) is also documented [51].

Timing and Channels

In TCOZ, all timing information is represented as real-valued measurements. TCOZ adopts all Timed CSP timing operators, for instance, *timeout* and *wait*. In order to describe the timing requirements of operations and sequences of operations, a deadline command has been introduced. If OP is an operation specification (defined through any combination of CSP process primitives and Object-Z operation schemas) then $OP \bullet \textsc{Deadline}\, t$ describes the process which has the same effect as OP, but is constrained to terminate no later than t (relative time). If it cannot terminate by time t, it deadlocks. The $\textsc{WaitUntil}$ operator is a dual to the deadline operator. The process $OP \bullet \textsc{WaitUntil}\, t$ performs OP, but will not terminate until at least time t. If it were to normally terminate before time t, it idles. In this thesis, when the term TCOZ timing constructs is mentioned, it means Timed CSP constructs with the extensions.

CSP channels are given an independent, first class role in TCOZ. In order to support the role of CSP channels, the state schema convention is extended to allow

declaration of communication channels. Contrary to the conventions adopted for internal state attributes, channels are viewed as shared (global) rather than as encapsulated entities. This is an essential consequence of their role as communication interfaces *between* objects. The introduction of channels in TCOZ reduces the need to reference other classes in class definitions, thereby enhancing the modularity of system specifications.

Active Objects and Passive Objects

Active objects have their own thread of control, while passive objects are controlled by other objects in a system. In TCOZ, an identifier MAIN (non-terminating process) is used to determine the behavior of active objects of a given class. The MAIN operation is optional in a class definition. It only appears in a class definition when the objects of that class are active objects. Classes for defining passive objects will not have the MAIN definition, but may contain CSP process constructors. If ob_1 and ob_2 are active objects of the class C, then the independent parallel composition of the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\mathrm{MAIN} \mid\mid\mid ob_2.\mathrm{MAIN}$.

Semantics of TCOZ

The details of the blended state/event process model form the basis for the TCOZ denotational semantics [51]. In brief, the semantic approach is to identify the notions of operation and process by providing a process interpretation of the Z

operation schema construct. TCOZ differs from many other approaches to blending Object-Z with a process algebra in that it does not identify operations with events. Instead an unspecified, fine-grained, collection of state-update events is hypothesized. Operation schemas are modelled by the collection of those sequences of update events that achieve the state change described by the schema. This means that there is no semantic difference between a Z operation schema and a CSP process. It therefore makes sense to also identify their syntactic classes.

The process model used by TCOZ consists of sets of tuples consisting of: an *initial* state; a *trace* (a sequence of time stamped events, including update-events), a *refusal* (a record of what and when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is if a termination event ✓ occurs), then the state at the time of termination is called the *final* state.

The process model of an operation schema consists of all initial states and update traces (terminated with a ✓) such that the initial state and the final state satisfy the relation described by the schema. If no legal final state exists for a given initial state, the operation diverges immediately. An advantage of this semantics is that it allows CSP process refinement to agree with Z operation refinement.

Network Topologies

The syntactic structure of the CSP synchronization operator is suitable for the case of pipeline-like communication topologies. When expressing more complex communication topologies, it generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology [50]. For example, consider that processes A and B communicate privately through the interface ab, processes A and C communicate privately through the interface ac, and processes B and C communicate privately through the interface bc.

One CSP expression for such a network communication system is

$$(A[bc'/bc] \,|[\, ab, ac \,]|\, (B[ac'/ac] \,|[\, bc \,]|\, C[ab'/ab]) \setminus ab, ac, bc)[ab, ac, bc/ab', ac', bc'].$$

The hiding and renaming is necessary in order to cover cases such as C not being able to communicate on channel ab.

The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology. This network topology of A, B and C may be described by

$$\| (A \overset{ab}{\longleftrightarrow} B;\; B \overset{bc}{\longleftrightarrow} C;\; C \overset{ca}{\longleftrightarrow} A).$$

Other forms of stylized usage allow network connections with common nodes to be run together, for example

$$\| (A \overset{ab}{\longleftrightarrow} B \overset{bc}{\longleftrightarrow} C \overset{ca}{\longleftrightarrow} A),$$

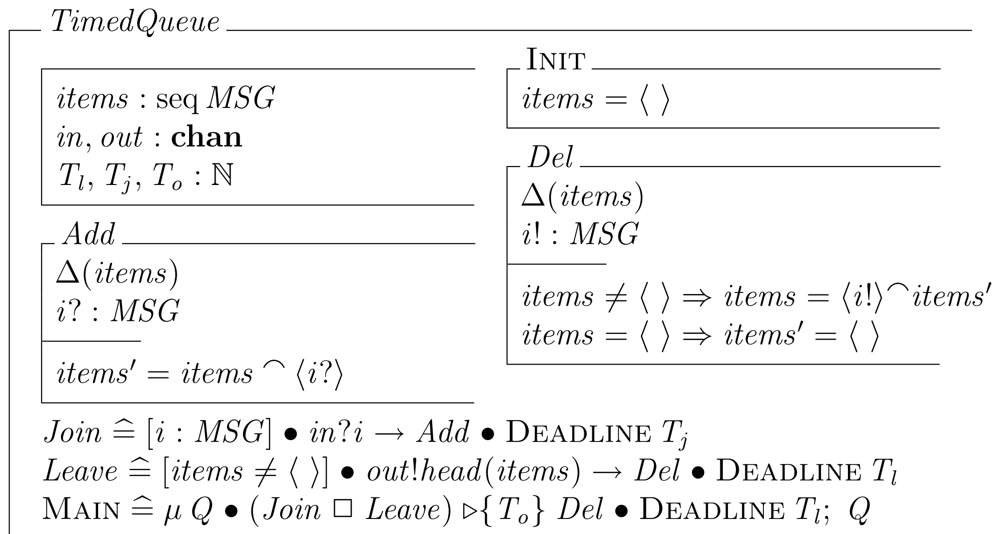
and multiple channels above the arrow, for example if processes D and F communicate privately through the channels df_1 and df_2, then

$$\|(D \xleftrightarrow{df_1, df_2} F).$$

A Timed Message Queue Example

The use of TCOZ is illustrated by the message queue example in section 2.1 tempered with timed constraints, which can receive a new message through an input channel '*in*' within a time duration 'T_j' or remove the first message in the queue and send it through an output channel '*out*' within a time duration 'T_l'. If there is no interaction with environment within a certain time 'T_o', then the head message will be removed from the current list.

As mentioned, TCOZ varies from Object-Z in the structure of class definitions, which may include Timed CSP channel and process definitions.



The schema expressions are same as those in the Object-Z model, which capture

the data structure and functionalities. T_l, T_j, T_o denote the time constants; and *in* and *out* denote the communication channels. The dynamic behavior modelled in TCOZ is expressed in the form of Timed CSP process definitions. Note that '$\widehat{=}$' means 'is defined as'. The *Join* operation specifies that after the parameter *i* has been input on channel *in*, the state-calculation *Add* is performed, and this *Add* operation is guarded by a DEADLINE expression, which constrains that the *Add* operation shall be finished in T_j time units. The *Leave* operation specifies that if the sequence *items* is not empty, it will output *head*(*items*), i.e., the first element of *items*, through channel *out* and remove this element from *items* in T_l time units. The behavior of this timed queue system is specified by the *Main* process which defines a repeated timeout process. More information on the TCOZ notation can be found in Appendix A.1.

Building on the strengths of the Object-Z and Timed CSP notations, TCOZ is capable of modelling state, process and timing aspects of complex systems, thus is well suited for presenting more complete and coherent requirement models for real-time complex systems. However, one problem of existing formal methods, especially for integrated formal methods like TCOZ, is that very few of them has direct tool support. This is mainly because high-level specification languages such as Object-Z/TCOZ usually contain various kinds of abstracted system information, which makes the implementation of a verification tool from scratch difficult.

2.4 Timed Automata

Timed Automata (TA) [4] are finite state machines with clocks. It was introduced as a formal notation to model the behavior of real-time systems. Its definition provides a general way to annotate state-transition graphs with timing constraints using finitely many real-valued clock variables. Another interesting aspect is that there exist model checking methods for temporal logics with quantitative temporal operators which are directly applicable to TA. Thus a variety of tools such as UPPAAL [48], KRONOS [16] etc., are available for specification and verification of real-time systems modelled in TA. A timed automaton A is a tuple (S, Σ, C, I, T), where

- S is a finite set of states.
- Σ is a set of actions/events.
- C is a finite set of clocks.
- I is a mapping that labels each state s in S with some clock constraint $\Phi(C)$. $\Phi(C)$is a set of clock constraints which is defined by the following grammar:

  $$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2$$

  where $x \in C$, and c is a constant.
- T, a subset of $S \times S \times \Sigma \times 2^C \times \Phi(C)$, is the set of transitions. A switch $\langle s, s', a, \lambda, \delta \rangle$ represents a transition from state s to state s' on input symbol a. The set λ gives the clocks to be reset with this transition, and δ is a clock constraint over C that specifies when the switch is enabled.

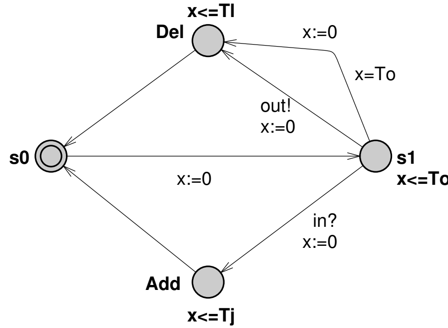


Figure 2.1: Timed Queue

As usual for automata, an initial or set of initial states can also be specified, as well as terminal state(s). We also assume that the special event τ, denoting an internal transition, is included in the events Σ. For example, the timed message queue example can be designed as shown in Figure 2.1. The model has four states: *Add*, *Del*, $s0$ and $s1$. The initial state $s0$ is denoted as a double circle. The control is passed from $s0$ to state $s1$ by an internal transition on which a clock x is reset to 0. Then the automaton waits for *To* time units to respond to the events from its environment through channels *in* and *out* to add or delete a certain message. If no event occurs during the *To* time units, the automaton will do the deletion and then return to its initial state.

UPPAAL

UPPAAL [58] is a tool for modelling, simulation and verification of real-time systems, developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. The tool is appropriate for systems that

can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. It consists of three main parts: a system editor, a simulator and a model checker. The system editor provides a graphical interface for the tool. Its output is an XML representation of the timed automata. Typically, a system description will consist of a set of instances of timed automata declared from the process templates, and of some global data, such as global clocks, variables, synchronization channels.

The simulator is a validation tool which enables examination of possible dynamic executions of a system. The model checker checks invariant and bounded liveness properties by exploring the symbolic state space of a system, i.e., reachability analysis in terms of symbolic states represented by constraints.

The model checking engine of UPPAAL is designed to check a subset of TCTL [5] formulas for networks of timed automata. The formulas contain no nested quantifiers and should be one of the following forms:

$\mathbf{A}[]\phi$ — Invariantly ϕ

$\mathbf{E} <> \phi$ — Possibly ϕ

$\mathbf{A} <> \phi$ — Always Eventually ϕ

$\mathbf{E}[]\phi$ — Potentially Always ϕ

$\phi \rightarrow \psi$ — Shorthand for $\mathbf{A}[\,](\phi \Rightarrow A <> \psi)$

where ϕ, ψ are local properties that can be checked locally on a state, i.e., Boolean

expressions over predicates on the states and integer variables, and clock constraints.

In summary, as one of the most widely studied modelling languages for real-time systems, Timed Automata has well developed tool support for model-checking such as UPPAAL. However, one problem of TA is that it lacks of high-level composable patterns to support systematic design for complex real-time systems. On the other hand, TCOZ, built on the strengths of Object-Z and TCSP, has powerful composable language constructs to directly capture common timing constraints. However it has no tool support. In the next chapter, we will discuss how these modelling languages can be linked together to complement each other.

Chapter 3

Composable TA patterns

High-level real-time system requirements often need to state the system timing constraints in terms of *deadline*, *timeout*, and *waituntil* commands which can be regarded as common timing constraint patterns. For example, "task A must complete within t time units" is a typical one (*deadline*). TCOZ was developed for modelling the high-level real-time system requirements; it has the composable language constructs that directly capture those common timing patterns. On the other hand, if TA is used to capture real-time requirements, then one often needs to manually cast those timing patterns into a set of clock variables with carefully calculated clock constraints, which is a process that is very much closer to design rather than specification. One interesting question is the following: can we build a set of TA patterns that correspond to the TCOZ timing constructs? If such a set of TA patterns can be formulated, then not only the transformation from TCOZ to TA can be readily achieved, but also TA alone can be more applicable for capturing high-level requirements if those TA patterns are utilized.

In this chapter, we introduce a set of composable Timed Automata patterns defined based on TCOZ process constructs and discuss the use of these patterns.

3.1 Z definition of Timed Automata

Since the current published semantics of TCOZ [51] is specified in Z, we define a set of composable TA patterns also in the same Z-meta notation. First of all, we

give the definition of TA in Z as follows.

$$[State, Event, Clock]$$
$$\mathbb{T} == \{r : \mathbb{R} \mid r >= 0\}$$
$$\Phi ::= (_ <= _)\langle\langle Clock \times \mathbb{T}\rangle\rangle \mid (_ >= _)\langle\langle Clock \times \mathbb{T}\rangle\rangle \mid (_ < _)\langle\langle Clock \times \mathbb{T}\rangle\rangle \mid (_ > _)\langle\langle Clock \times \mathbb{T}\rangle\rangle \mid (_ \wedge _)\langle\langle \Phi \times \Phi\rangle\rangle \mid true$$
$$Label \mathrel{\widehat=} Event \times \mathbb{P}\, Clock \times \Phi$$
$$Transition \mathrel{\widehat=} State \times Label \times State$$

$$clk : \Phi \rightarrow \mathbb{P}\, Clock$$

$$\forall x : Clock;\ t : \mathbb{T} \bullet clk(true) = \varnothing$$
$$clk(x <= t) = \{x\} \wedge clk(x >= t) = \{x\}$$
$$clk(x < t) = \{x\} \wedge clk(x > t) = \{x\}$$
$$\forall \varphi_1, \varphi_2 : \Phi \bullet clk(\varphi_1 \wedge \varphi_2) = clk(\varphi_1) \cup clk(\varphi_2)$$

$\mathcal{S}_{TA}$

$$S : \mathbb{P}\, State;\quad i, e : State$$
$$\Sigma : \mathbb{P}\, Event$$
$$C : \mathbb{P}\, Clock$$
$$I : State \nrightarrow \Phi$$
$$T : \mathbb{P}\, Transition$$

$$\{i, e\} \subseteq S \wedge \operatorname{dom} I = S$$
$$\forall \varphi : \operatorname{ran} I \bullet clk(\varphi) \subseteq C$$
$$\forall s, s' : State;\ l : Label \bullet (s, l, s') \in T \Rightarrow \{s, s'\} \subseteq S \wedge \pi_1(l) \in \Sigma \wedge \pi_2(l) \subseteq C$$

There are three basic types, i.e., *State*, *Event* and *Clock*; $\mathbb{T}$ is a set of positive real numbers; Φ defines the types of clock constraints, in which a *true* type is added here to represent the empty set of clock constraints; Φ specifies a clock constraint; the function $clk(\varphi)$ returns the set of clocks used in φ; a timed automaton $\mathcal{S}_{TA}$ is defined as a binding (binding is a partial function from variables to values), in which S models states; i and e respectively represent the initial state and terminal state; σ is a set of events; C is a set of clock variables; I defines local invariants which give a clock constraint to each state; and T models transitions;

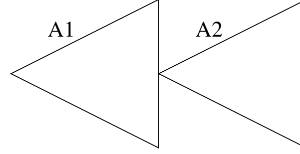
the second element of T, i.e., *Label*, models transition conditions, in which *Event* is an enabling event, $\mathbb{P}\, Clock$ represents the set of clocks reset on a transition.

To simplify the composition of automata, we define one initial/terminal state here instead of a set of initial/terminal states for a timed automaton. In case there are multiple initial/terminal states, we assume there is a unique initial/terminal state which is connected to all initial/terminal states through an internal transition τ. Note that the terminal state may be identical to the initial state, and in some cases it may be unreachable, and thus omitted from a concrete automaton.

3.2 TA patterns

In the following, a set of TA patterns according to TCOZ constructs are defined in Z together with their graphic presentations. In these graphical TA patterns, an automaton A is abstracted as a triangle, the left vertex of this triangle or a circle attached to the left vertex represents the initial state of A, and the right edge represents the terminal state of A. Those timed patterns together with their formal definitions can be readily understood. The timed composable patterns can be seen as a reusable high-level library that may facilitate a systematic engineering process when TA is used to design timed systems. Furthermore, these patterns provide an interchange media for transforming TCOZ specifications into TA designs.

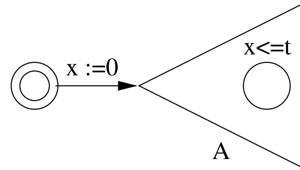
$$
\begin{array}{|l}
seq : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall A_1, A_2 : \mathcal{S}_{TA} \bullet \\
\quad seq(A_1, A_2) = \langle\!| \\
\quad S \mathrel{\widehat=} A_1.S \cup A_2.S,\ i \mathrel{\widehat=} A_1.i,\ e \mathrel{\widehat=} A_2.e, \\
\quad \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma,\ C \mathrel{\widehat=} A_1.C \cup A_2.C,\ I \mathrel{\widehat=} A_1.I \cup A_2.I, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(A_1.e, (\tau, \varnothing, true), A_2.i)\} \ |\!\rangle
\end{array}
$$



D1. Sequential Composition

The *sequential composition* pattern: there are two timed automata A_1, A_2. By linking the terminal state of A_1 with the initial state of A_2, the resulting automaton passes control from A_1 to A_2 immediately when A_1 goes to its terminal state.

$$
\begin{array}{|l}
deadline : \mathcal{S}_{TA} \times \mathbb{T} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall A : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists x : Clock;\ i_0 : State \mid x \notin A.C \wedge i_0 \notin A.S \bullet \\
\quad deadline(A, t) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0\},\ i \mathrel{\widehat=} i_0,\ e \mathrel{\widehat=} A.e, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma,\ C \mathrel{\widehat=} A.C \cup \{x\}, \\
\quad I \mathrel{\widehat=} \{s : A.S \bullet (s, x <= t \wedge A.I(s))\} \cup \{i_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (\tau, \{x\}, true), A.i)\} \ |\!\rangle
\end{array}
$$

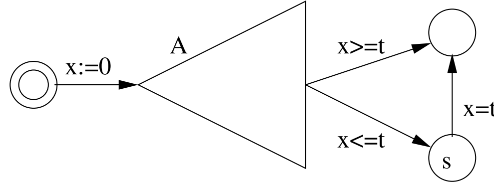


D2. Deadline

The *deadline* pattern: there is a single fresh clock x. When the system switches to the automaton A, the clock x gets reset to 0. The local invariant $x <= t$ covers

each state of the timed automaton A and specifies the requirement that a switch must occur before t time units for every state of A. Thus the timing constraint expressed by this automaton is that A should terminate no later than after t time units.

$$
\begin{array}{|l}
\mathit{waituntil} : \mathcal{S}_{TA} \times \mathbb{T} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall A : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists x : \mathit{Clock};\ i_0, e_0, s : \mathit{State} \mid \\
x \notin A.C \wedge \{i_0, s, e_0\} \cap A.S = \varnothing \bullet \\
\quad \mathit{waituntil}(A, t) = \langle\!| \\
\quad S \mathrel{\widehat=} A.S \cup \{i_0, s, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma, C \mathrel{\widehat=} A.C \cup \{x\}, \\
\quad I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto \mathit{true}, s \mapsto \mathit{true}, e_0 \mapsto \mathit{true}\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (\tau, \{x\}, \mathit{true}), A.i), \\
\qquad\qquad (A.e, (\tau, \varnothing, x >= t), e_0), (A.e, (\tau, \varnothing, x <= t), s), \\
\qquad\qquad (s, (\tau, \varnothing, x = t), e_0)\} \ |\!\rangle
\end{array}
$$

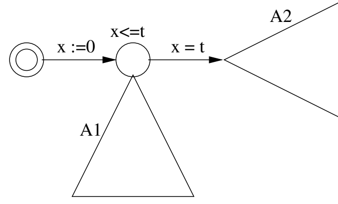


D3. Waituntil

The *waituntil* timed pattern: the automaton is constrained to finish its process not earlier than t time units. Two situations are captured here: if the process of A finishes earlier than t time units, then the automaton idles until t time units and if the process of A takes more than t time units, then the automaton terminates as A terminates.

$timeout : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times \mathbb{T} \rightarrow \mathcal{S}_{TA}$

$$\begin{array}{l}
\forall A_1, A_2 : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists x : Clock;\ i_0, e_0 : State;\ \varphi : \Phi \mid x \notin (A_1.C \cup A_2.C) \\
\wedge \{i_0, e_0\} \cap (A_1.S \cup A_2.S) = \varnothing \wedge \varphi = I(A_1.i) \bullet timeout(A_1, A_2, t) = \langle\!| \\
\quad S \mathrel{\widehat{=}} A_1.S \cup A_2.S \cup \{i_0, e_0\}, i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, \\
\quad \Sigma \mathrel{\widehat{=}} A_1.\Sigma \cup A_2.\Sigma, C \mathrel{\widehat{=}} A_1.C \cup A_2.C \cup \{x\}, \\
\quad I \mathrel{\widehat{=}} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\} \oplus \{A_1.i \mapsto x <= t \wedge \varphi\}, \\
\quad T \mathrel{\widehat{=}} A_1.T \cup A_2.T \cup \\
\qquad \{(i_0, (\tau, \{x\}, true), A_1.i), (A_1.i, (\tau, \varnothing, x = t), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \,|\!\rangle
\end{array}$$

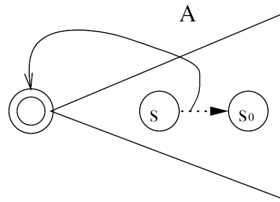


D4. Timeout

The *timeout* pattern: there are two timed automata A_1 and A_2. If no transition has been triggered for t time units in timed automaton A_1, then A_1 will timeout and the control will be passed to A_2.

$recursion : \mathcal{S}_{TA} \times State \rightarrow \mathcal{S}_{TA}$

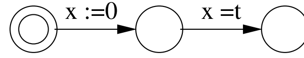
$$\begin{array}{l}
\forall A : \mathcal{S}_{TA};\ s_0 : State \mid s_0 \in A.S \bullet \\
\quad recursion(A, s_0) = \langle\!| \\
\quad S \mathrel{\widehat{=}} A.S, i \mathrel{\widehat{=}} A.i, e \mathrel{\widehat{=}} A.e, \\
\quad \Sigma \mathrel{\widehat{=}} A.\Sigma, C \mathrel{\widehat{=}} A.C, I \mathrel{\widehat{=}} A.I, \\
\quad T \mathrel{\widehat{=}} A.T \cup \{s : State, l : Label \mid (s, l, s_0) \in A.T \bullet (s, l, i)\} \\
\qquad - \{s : State, l : Label \mid (s, l, s_0) \in A.T \bullet (s, l, s_0)\} \,|\!\rangle
\end{array}$$



D5. Recursion

The *recursion* pattern: given a timed automaton A and a state s_0, which is a fixed point. The recursion is achieved by diverting all the transitions from pointing to s_0 to the initial state of A. The dotted arrow represents the transitions which are originally pointing to s_0. In the resultant automaton these transitions are replaced by transitions which points to the initial state of A.

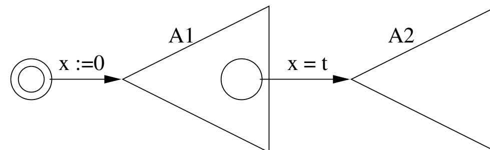
$$\begin{array}{|l}
wait : \mathbb{T} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, t : \mathbb{T};\ \exists\, x : Clock;\ i_0, w_0, e_0 : State \bullet \\
\quad wait(t) = \langle\!| \ S \mathrel{\widehat=} \{i_0, w_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \\
\quad I \mathrel{\widehat=} \{i_0 \mapsto true, w_0 \mapsto true, e_0 \mapsto true\}, \\
\quad \Sigma \mathrel{\widehat=} \varnothing, C \mathrel{\widehat=} \{x\}, \\
\quad T \mathrel{\widehat=} \{(i_0, (\tau, \{x\}, true), w_0), (w_0, (\tau, \varnothing, x = t), e_0)\} \ |\!\rangle
\end{array}$$



D6. Wait

The *wait* pattern: Time idles at its second state for t time units then terminates.

$$\begin{array}{|l}
tinterrupt : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times \mathbb{T} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, A_1, A_2 : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists\, x : Clock;\ i_0, e_0 : State \mid \\
x \notin A_1.C \cup A_2.C \wedge i_0 \notin A_1.S \cup A_2.S \wedge e_0 \notin A_1.S \cup A_2.S \bullet \\
\quad tinterrupt(A_1, A_2, t) = \langle\!| \ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma, \\
\quad C \mathrel{\widehat=} A_1.C \cup A_2.C \cup \{x\}, I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(i_0, (\tau, \{x\}, true), A_1.i)\} \\
\qquad \cup \{s : A_1.S \bullet (s, (\tau, \varnothing, x = t), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \ |\!\rangle
\end{array}$$

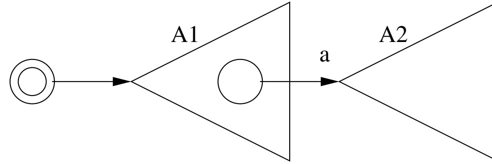


D7. Timed Interrupt

The *timed interrupted* pattern: it is composed of two automata, A_1 and A_2, the control will be passed from A_1 to A_2 immediately if A_1 cannot finish its process within t time units, that is, when the value of clock x increases to t, there will be a transition to the initial state of A_2 from any state of A_1.

$$einterrupt : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \times Event \rightarrow \mathcal{S}_{TA}$$

$$\begin{array}{l}
\forall A_1, A_2 : \mathcal{S}_{TA};\ a : Event;\ \exists i_0, e_0 : State \mid \\
i_0 \notin A_1.S \cup A_2.S \wedge e_0 \notin A_1.S \cup A_2.S \bullet \\
\quad einterrupt(A_1, A_2, a) = \langle\!| \ S \mathrel{\widehat=} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A_1.\Sigma \cup A_2.\Sigma \cup \{a\}, \\
\quad C \mathrel{\widehat=} A_1.C \cup A_2.C, I \mathrel{\widehat=} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A_1.T \cup A_2.T \cup \{(i_0, (\tau, \varnothing, true), A_1.i)\} \\
\qquad \cup \{s : A_1.S \bullet (s, (a, \varnothing, true), A_2.i)\} \\
\qquad \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \ |\!\rangle
\end{array}$$

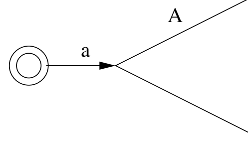


D8. Event Interrupt

The *event interrupted* pattern: it is composed of two automata, A_1 and A_2, the control will be passed from A_1 to A_2 immediately when event a happens.

$$eprefix : Event \times \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA}$$

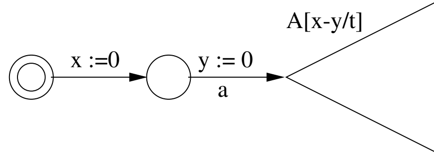
$$\begin{array}{l}
\forall a : Event;\ A : \mathcal{S}_{TA};\ \exists i_0 : State \mid i_0 \notin A.S \bullet \\
\quad eprefix(a, A) = \langle\!| \ S \mathrel{\widehat=} A.S \cup \{i_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} A.e, \\
\quad \Sigma \mathrel{\widehat=} A.\Sigma \cup \{a\}, C \mathrel{\widehat=} A.C, I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto true\}, \\
\quad T \mathrel{\widehat=} A.T \cup \{(i_0, (a, \varnothing, true), A.i)\} \ |\!\rangle
\end{array}$$



D9. Event Prefix

The *event prefix* pattern: if event a happens, then the control will be passed to A immediately.

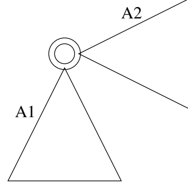
$$
\begin{array}{l}
\mathit{tprefix} : \mathit{Event} \times \mathbb{T} \times \mathcal{S}_{TA} \rightarrow \mathcal{S}_{TA} \\
\hline
\forall\, a : \mathit{Event};\ A : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists\, x, y : \mathit{Clock};\ i_0, s_0 : \mathit{State} \mid \\
t = x - y \wedge \{x, y\} \cap A.C = \varnothing \wedge \{i_0, s_0\} \cap A.S = \varnothing \bullet \\
\quad \mathit{tprefix}(a, t, A) = \langle\!|\ S \mathrel{\widehat=} A.S \cup \{i_0, s_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} A.e, \Sigma \mathrel{\widehat=} A.\Sigma \cup \{a\}, \\
\quad I \mathrel{\widehat=} A.I \cup \{i_0 \mapsto \mathit{true}, s_0 \mapsto \mathit{true}\}, C \mathrel{\widehat=} A.C \cup \{x, y\}, \\
\quad T \mathrel{\widehat=} A.T[x - y/t] \cup \{(i_0, (\tau, \{x\}, \mathit{true}), s_0)\} \cup \{(s_0, (a, \{y\}, \mathit{true}), A.i)\}\ |\!\rangle
\end{array}
$$



D10. Timed Event Prefix

The *timed event prefix* pattern: there are two clocks, x is reset at the initial state, y is reset when event a happens. The time difference of x and y records the time point at which a happens and substitutes the variable t in the timed automaton enabled by event a for further use.

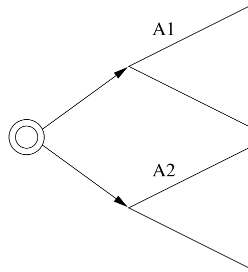
$$
\begin{array}{l}
extchoice : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \to \mathcal{S}_{TA} \\
\hline
\forall A_1, A_2 : \mathcal{S}_{TA};\ \exists i_0, e_0 : State \mid \{i_0, e_0\} \cap (A_1.S \cup A_2.S) = \varnothing \bullet \\
\quad extchoice(A_1, A_2) = \langle\!| \ S \mathrel{\widehat{=}} A_1.S \cup A_2.S \cup \{i_0, e_0\} - \{A_1.i, A_2.i\}, \\
\quad i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, C \mathrel{\widehat{=}} A_1.C \cup A_2.C, \\
\quad I \mathrel{\widehat{=}} A_1.I \cup A_2.I \cup \{(i_0, I(A_1.i) \wedge I(A_2.i))\} - \{(A_1.i, I(A_1.i)), (A_2.i, I(A_2.i))\}, \\
\quad T \mathrel{\widehat{=}} A_1.T \cup A_2.T \cup \{(A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \\
\qquad \cup\{l : Label, k : State \mid (A_1.i, l, k) \in A_1.T \vee (A_2.i, l, k) \in A_2.T \bullet (i_0, l, k)\} \\
\qquad \cup\{l : Label, k : State \mid (k, l, A_1.i) \in A_1.T \vee (k, l, A_2.i) \in A_2.T \bullet (k, l, i_0)\} \\
\qquad -\{t : Transition \mid \pi_1(t) = A_1.i \vee \pi_1(t) = A_2.i \vee \pi_3(t) = A_1.i \\
\qquad \vee\ \pi_3(t) = A_2.i\} \ |\!\rangle
\end{array}
$$



D11. External Choice

The *external choice* pattern: timed automata A_1 and A_2 share a common initial state and the environment has the choice to trigger one of them by different external events.

$$
\begin{array}{l}
intchoice : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \to \mathcal{S}_{TA} \\
\hline
\forall A_1, A_2 : \mathcal{S}_{TA};\ \exists i_0, e_0 : State \mid \{i_0, e_0\} \cap (A_1.S \cup A_2.S) = \varnothing \bullet \\
\quad intchoice(A_1, A_2) = \langle\!| \ S \mathrel{\widehat{=}} A_1.S \cup A_2.S \cup \{i_0, e_0\}, \\
\quad i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, \Sigma \mathrel{\widehat{=}} A_1.\Sigma \cup A_2.\Sigma, \\
\quad C \mathrel{\widehat{=}} A_1.C \cup A_2.C, I \mathrel{\widehat{=}} A_1.I \cup A_2.I \cup \{i_0 \mapsto true, e_0 \mapsto true\}, \\
\quad T \mathrel{\widehat{=}} A1.T \cup A2.T \cup \{(i_0, (\tau, \varnothing, true), A_1.i), (i_0, (\tau, \varnothing, true), A_2.i), \\
\qquad (A_1.e, (\tau, \varnothing, true), e_0), (A_2.e, (\tau, \varnothing, true), e_0)\} \ |\!\rangle
\end{array}
$$



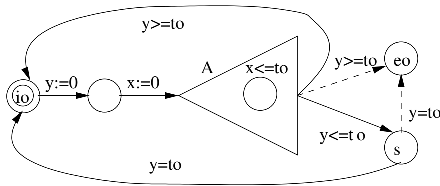
D12. Internal Choice

The *internal choice* pattern: there are two timed automata A_1 and A_2 and an initial state. The choice of which automaton to be triggered is decided by internal events.

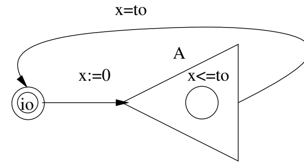
3.3 Generating New Patterns

New patterns can be composed from the existing ones. For example, the new timing pattern *PeriodicRepeat*, which specifies "Task A is repeated every t_0 time units provided that A is guaranteed to terminate before t_0 time units", can be composed by three existing patterns - *deadline*, *waituntil* and *recursion*, as shown in Figure (a). Assuming A is the automaton which performs the task A, clocks x and y are generated to respectively give the time constraints for the *deadline* and *waituntil* pattern. The terminal state of the automaton A_0 ($A_0 = \mathit{waituntil}(\mathit{deadline}(A, t_0), t_0)$), e_0, is the fix point for the recursion pattern, thus the transitions of A_0 which were originally pointing to e_0 are diverted and point to the initial state of A_0.

$$
\begin{array}{|l}
\mathit{PeriodicRepeat} : \mathcal{S}_{TA} \times \mathbb{T} \to \mathcal{S}_{TA} \\
\hline
\forall\, A, A_0 : \mathcal{S}_{TA};\ t_0 : \mathbb{T};\ e_0 : \mathit{State} \mid e_0 = A_0.e \\
\wedge\ A_0 = \mathit{waituntil}(\mathit{deadline}(A, t_0), t_0) \bullet \\
\qquad \mathit{PeriodicRepeat}(A, t_0) = \mathit{recursion}(A_0, e_0)
\end{array}
$$



(a)



(b)

The resultant automaton can be simplified as shown in Figure (b), in which any consecutive initial, terminal and intermediate states linked with a τ transition label are incorporated into one state to simplify the resultant automaton. To further reduce the state space of the resultant timed automaton, clock y is removed by reusing clock x. The two diverted transitions are merged into one transition since they can only be enabled when the value of clock x equals t_0.

$$PeriodicRepeat(A, t_0) = \langle\!| \; S \mathrel{\widehat=} A.S \cup \{i_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, \Sigma \mathrel{\widehat=} A.\Sigma,$$

$$C \mathrel{\widehat=} A.C \cup \{x\}, I \mathrel{\widehat=} \{s : A.S \bullet (s, x <= t_0 \wedge A.I(s))\},$$

$$T \mathrel{\widehat=} \{(i, (\tau, \{x\}, true), A.i)\} \cup \{A.e, (\tau, \varnothing, x = t_0), i)\} \cup \; A.T \; |\!\rangle$$

3.4 Guidelines for TA Design Using Patterns

Many complex real-time systems can be naturally modelled as collections of small processes lying in different layers of the systems, operating and interacting sequentially or concurrently. Our generic TA patterns, in a way, provide a set of templates to decompose a complex real-time system into different layers and smaller components.

There are certain guidelines for the engineers to use these generic timed patterns for systematic TA design:

- Decide the layers of the complex system. The abstracted triangle automaton can be seen as an outside layer, which will be substituted by its inside layer.

The TA model of a system can finally be generated in a top-down way by continuously embodying its inside layers using appropriate patterns.

  - For most reactive systems, usually the outmost layer of such systems can be modelled by a recursion pattern.
  - For systems which run one time only, a sequential pattern can be used to model its outmost layer.

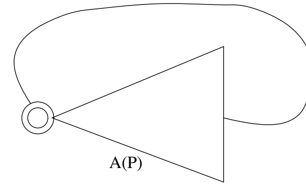
- Decompose the complex processes of one layer into smaller processes with simpler behaviors. Those smaller processes mostly are composed together by sequential composition or alternation.
  - To describe a process which has different behaviors, usually the *external choice* or the *internal choice* pattern should be applied.
  - To capture a process which has exceptions, usually the *timed interrupt* or the *event interrupt* patterns should be chosen.
  - To specify requirement constraints, e.g., timing constraints, the *deadline* and the *waituntil* pattern can be utilized.
  - Sequential behaviors can be captured by utilizing the sequential composition pattern.

An Example: Timed Queue System

In the following, we will use the timed queue example in chapter 2.3 to demonstrate how the timed patterns can be applied to facilitate TA designs in a systematic way.

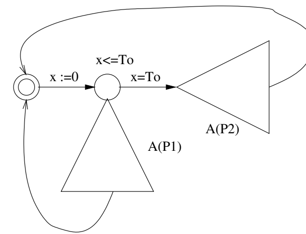
From its specification, four kinds of patterns in total can be identified in the timed queue class, i.e., *recursion*, *timed out*, *external choice* and *event prefix*. Assume, for any process P, that $\mathcal{A}(P)$ is the TA model of P. The timed queue automaton can be generated step by step in a top-down way as follows:

step 1. The MAIN process is a recursive process, say P, which stands for $(Join \ \Box \ Leave) \triangleright \{T_o\} \ Del \bullet$ DEADLINE T_l. According to the *recursion* pattern, $\mathcal{A}$(MAIN) can be projected as shown in figure (a).



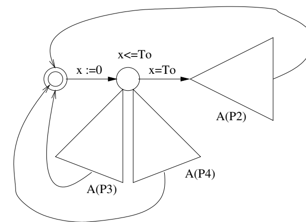
(a)

step 2. Assume $P1$ stands for $(Join \ \Box$ $Leave)$ and $P2$ stands for $Del \bullet$ DEADLINE T_l, then according to the *timeout* pattern, $\mathcal{A}$(MAIN) can be further derived as shown in figure (b).



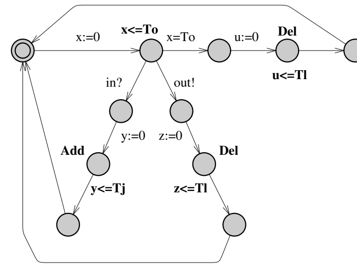
(b)

step 3. According to the *external choice* pattern, $\mathcal{A}(P1)$ can be derived as shown in the figure (c), in which $P3$ represents the process *Join*, and $P4$ represents the process *Leave*.



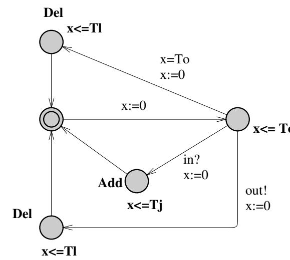
(c)

step 4. The processes *Join* and *Leave* both match the timed event prefix pattern and the deadline pattern. The atomic operations *Add* and *Del* are projected as states. Clocks x, y, z and u are generated to give these operations the timing constraints. Now the MAIN process has been translated into an automaton as shown in figure (d).



(d)

step 5. The last step of this translation is to simplify the resultant automaton. Any two consecutive initial and terminal states are merged into one state. In order to reduce the state space of the automaton, the clock y, z and u are replaced by clock x after resetting its value to 0.



(e)

3.5 Discussion and Conclusion

The set of patterns we presented are defined according to TCOZ process constructs, i.e., Timed CSP constructs with the extensions, thus preserves the expressiveness

of Timed CSP in describing dynamic real-time behaviors. Moreover, these timed composable patterns:

- not only provide a proficient interchange media for transforming TCOZ specifications into TA designs, which supports one possible engineering development process: TCOZ for high-level requirement specifications, then TA for design and timing analysis,
- but also provide a generic reusable framework for developing real-time systems in TA alone.

In the following chapters, this concept of timed patterns will be constantly applied to facilitate building up the framework of modelling and checking of complex real-time systems.

Chapter 4

Projection from TCOZ to TA

4.1 Introduction

The specification of complex real-time systems requires powerful mechanisms for modelling state, concurrency and real-time behavior as well as tool support for verifying the established system models. Integrated formal modelling techniques are well suited for presenting complete and coherent requirement models for complex systems. Timed Communicating Object Z (TCOZ) is an integrated high-level abstracted formal specification language for modelling both the state, process and timing aspects of complex systems. However, the challenge is how to verify the TCOZ models with tool support, and especially to analyze timing properties. CSP has a good model checker FDR [60] but not Timed CSP[1]. Rather than develop a tool for TCOZ from scratch, we believe a better approach is to reuse any existing tools. The graph-based modelling technique, Timed Automata (TA) has well developed automatic tool support, e.g., UPPAAL [48], KRONOS [16], TEMPO [69], RED [74] and Timed COSPAN [72].

In this chapter, we will define a set of rules for mapping TCOZ to Timed Automata based on the timed patterns presented in the previous chapter, and provide a correctness proof for this transformation. The core of the transformation is performed by analyzing the dynamic view of the TCOZ model, i.e., the timed process associated with each class. A Java tool to automate the transformation process is implemented and illustrated.

[1] To our knowledge, the only tool support for Timed CSP is the preliminary PVS encoding of Timed CSP in Brooke's PhD thesis [10].

4.2 Mapping Rules

Since the timed composable patterns are defined according to TCOZ process constructs, the transformation rules are straightforward:

Definition 4.2.1 *We define the mapping function $\mathcal{A}$ from TCOZ processes to TA as follows.*

- **R1** : *If* $P = \text{Skip}$*, then* $\exists\ i_0, e_0 : State$*, so that* $\mathcal{A}(P) = \langle\!| S \mathrel{\widehat=} \{i_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, I \mathrel{\widehat=} \varnothing, T \mathrel{\widehat=} \{(i, (\tau, \varnothing, true), e)\} |\!\rangle$
- **R2** : *If* $P = \text{Stop}$*, then* $\exists\ i_0, e_0 : State$*, so that* $\mathcal{A}(P) = \langle\!| S \mathrel{\widehat=} \{i_0, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, I \mathrel{\widehat=} \varnothing, T \mathrel{\widehat=} \varnothing |\!\rangle$
- **R3** : *If* $P = a \rightarrow P_0$*, then* $\mathcal{A}(P) = eprefix(a, \mathcal{A}(P_0))$
- **R4** : *If* $P = a@t \rightarrow P_0$*, then* $\mathcal{A}(P) = tprefix(a, t, \mathcal{A}(P_0))$
- **R5** : *If* $P = \text{Wait}\ t$*, then* $\mathcal{A}(P) = wait(t)$
- **R6** : *If* $P = P_0 \bullet \text{WaitUntil}\ t$*, then* $\mathcal{A}(P) = waituntil(\mathcal{A}(P_0), t)$
- **R7** : *If* $P = P_0 \bullet \text{Deadline}\ t$*, then* $\mathcal{A}(P) = deadline(\mathcal{A}(P_0), t)$
- **R8** : *If* $P = P_1 \rhd \{t\} P_2$*, then* $\mathcal{A}(P) = timeout(\mathcal{A}(P_1), \mathcal{A}(P_2), t)$
- **R9** : *If* $P = P_1 \nabla \{t\} P_2$*, then* $\mathcal{A}(P) = tinterrupt(\mathcal{A}(P_1), \mathcal{A}(P_2), t)$
- **R10** : *If* $P = P_1 \nabla a \rightarrow P_2$*, then* $\mathcal{A}(P) = einterrupt(\mathcal{A}(P_1), \mathcal{A}(P_2), a)$
- **R11** : *If* $P = \mu N \bullet P(N)$*, then* $\mathcal{A}(P) = recursion(\mathcal{A}(P(N)), N)$
- **R12** : *If* $P = P_1;\ P_2$*, then* $\mathcal{A}(P) = seqcom(\mathcal{A}(P_1), \mathcal{A}(P_2))$

- **R13** : *If* $P = P_1 \sqcap P_2,$ *then* $\mathcal{A}(P) = intchoice(\mathcal{A}(P_1), \mathcal{A}(P_2))$
- **R14** : *If* $P = P_1 \Box P_2,$ *then* $\mathcal{A}(P) = extchoice(\mathcal{A}(P_1), \mathcal{A}(P_2))$
- **R15** : *If* $P = P_1 \, |[X]| \, P_2,$ *then* $\mathcal{A}(P) = \mathcal{A}(P_1) \parallel \mathcal{A}(P_2)$

In these mapping rules, channels, events and guards in a TCOZ model are viewed as triggers which cause the state transitions. They match the definition of actions and timed constraints in Timed Automata, thus, they are directly projected as transition conditions. Clock variables will be generated in the target automaton to guard its transition if the process of TCOZ to be translated has any timing constraints such as deadlines.

Consider the translation rule for the DEADLINE primitive. $P_0 \bullet \text{DEADLINE } t$ describes the process which has the same effect as P_0, but is constrained to terminate no later than t. This process can be translated into timed automaton $\mathcal{A}(P)$ according to the *deadline* pattern. The resultant automaton differs from $\mathcal{A}(P_0)$ in that its process time is limited to t time units, which is equivalent to the TCOZ expression $P_0 \bullet \text{DEADLINE } t$. Other mapping rules can be explained in the same way.

The above rules apply to all the TCOZ time primitives and its basic composition of events, guards and processes, through which all the important dynamic information about time constraints in a TCOZ specification can be completely translated into Timed Automata. The following provides the transformation rules for TCOZ class/objects:

Every object in a TCOZ model is projected as a timed automaton. The Init schema in TCOZ class is used to appoint one of those identified states to be an initial state. It will not be projected as a new state because it does not trigger any transition.

An operation schema in a TCOZ class, say P_{op}, is projected as a timed automaton $\mathcal{A}(P_{op})$. The projection rule is provided as follow,

- **R0** : *If* $P = P_{op}$*, then* $\exists\, i_0, op, e_0 : State$*, so that* $\mathcal{A}(P_{op}) = \langle\!| S \mathrel{\widehat=} \{i_0, op, e_0\}, i \mathrel{\widehat=} i_0, e \mathrel{\widehat=} e_0, I \mathrel{\widehat=} \varnothing, T \mathrel{\widehat=} \{(i_0, (\tau, \varnothing, true), op), (op, (e_{op}, \varnothing, true), e_0)\} |\!\rangle$

Where e_{op} represents the data change event.

Note that by using the term "projection", we mean only a subset (i.e., the TCSP process part) of TCOZ has been translated into TA. Namely, we mainly focus on how to reuse TA's tool-support to verify and analyze the timing behaviors of TCOZ models, i.e., how to formally translate the process part (i.e. TCSP part) of a TCOZ model into a TA model. Some constructs of TCOZ that cannot be mapped onto any concept of Timed Automata, such as predicate expressions, containment and etc. These are more relevant with the OZ part of TCOZ and are not considered in the projection not only because they are beyond of the expressiveness of TA (other Z tools i.e., Z/Eve tool can be used to check those properties), but also because they can often be separated out from the timing analysis, and of little relevance with the model-checking issues, thus are abstracted away here in our transformation.

4.3 Correctness

This section is devoted to the soundness proof for our mapping rules from TCOZ processes to Timed Automata.

Firstly two labelled transition systems (LTSs) are developed to represent the operational semantic models for TCOZ processes and Timed Automata. To show the soundness of the mapping rules, we prove that any source process in TCOZ and its corresponding target Timed Automaton preserve the same semantics under a bisimulation equivalence relation between the two LTSs.

The operational semantics for TCOZ processes is captured by the labelled transition system

$$TS^1_{\text{TCOZ}} \mathrel{\widehat{=}} (\mathcal{C},\ \Sigma^\tau \cup \mathbb{T},\ \ \longrightarrow_1)$$

Note that

$\mathcal{C} \mathrel{\widehat{=}} \mathcal{P} \times \mathbb{T}$ is the set of possible configurations, where $\mathcal{P}$ is the set of processes, and $\mathbb{T}$ is the time domain. A configuration $c = \langle P, t \rangle$ comprising process P and time t denotes a state in the transition system.

Σ^τ is the set of possible communication events including the silent event τ.

$\longrightarrow_1\ \subseteq\ (\mathcal{C}\ \times\ \Sigma^\tau \cup \mathbb{T}\ \times\ \mathcal{C})$ is the set of possible transitions.

A complete set of transition rules for TCOZ processes are provided in the following. Note that most of these rules can be referred in Schneider's book [63] except some

new constructs introduced into TCOZ later such as DEADLINE and WAITUNTIL.

r1 : $\langle \textsc{Skip}, t\rangle \xrightarrow{\tau}_1 \langle \epsilon, t\rangle$

r2 : $\langle \textsc{Stop}, t\rangle \xrightarrow{\delta}_1 \langle \textsc{Stop}, t+\delta\rangle$

r3 : $\langle a \rightarrow P, t\rangle \xrightarrow{a}_1 \langle P, t\rangle$

r4a : $\langle a@u \rightarrow P, t\rangle \xrightarrow{\delta}_1 \langle a@u \rightarrow P[u+\delta/u], t+\delta\rangle$

r4b : $\langle a@u \rightarrow P, t\rangle \xrightarrow{a}_1 \langle P[0/u], t\rangle$

r5a : $\langle \textsc{Wait}\, t_0, t\rangle \xrightarrow{\delta}_1 \langle \textsc{Wait}(t_0-\delta), t+\delta\rangle,\ \delta < t_0$

r5b : $\langle \textsc{Wait}\, t_0, t\rangle \xrightarrow{t_0}_1 \langle \epsilon, t+t_0\rangle$

r6a : $\dfrac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P \bullet \textsc{Deadline}\, t_0,t\rangle \xrightarrow{a}_1 \langle P' \bullet \textsc{Deadline}\, t_0,t\rangle}$

r6b : $\dfrac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle,\ \delta<t_0}{\langle P \bullet \textsc{Deadline}\, t_0,t\rangle \xrightarrow{\delta}_1 \langle P' \bullet \textsc{Deadline}(t_0-\delta),t'\rangle}$

r7a : $\dfrac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P \bullet \textsc{WaitUntil}\, t_0,t\rangle \xrightarrow{a}_1 \langle P' \bullet \textsc{WaitUntil}\, t_0,t\rangle}$

r7b : $\dfrac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle,\ \delta<t_0}{\langle P \bullet \textsc{WaitUntil}\, t_0,t\rangle \xrightarrow{\delta}_1 \langle P' \bullet \textsc{WaitUntil}(t_0-\delta),t'\rangle}$

r7c : $\dfrac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle,\ \delta \geqslant t_0}{\langle P \bullet \textsc{WaitUntil}\, t_0,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle}$

r8a : $\dfrac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P;\ Q,t\rangle \xrightarrow{a}_1 \langle P';\ Q,t\rangle}$

r8b : $\dfrac{\langle P,t\rangle \xrightarrow{a}_1 \langle \epsilon,t\rangle}{\langle P;\ Q,t\rangle \xrightarrow{a}_1 \langle Q,t\rangle}$

r8c : $\dfrac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle}{\langle P;\ Q,t\rangle \xrightarrow{\delta}_1 \langle P';\ Q),t'\rangle}$

r9a : $\dfrac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P \Box Q,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}$

r9b : $\frac{\langle Q,t\rangle \xrightarrow{a}_1 \langle Q',t\rangle}{\langle P \Box Q,t\rangle \xrightarrow{a}_1 \langle Q',t\rangle}$

r9c : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle \quad \langle Q,t\rangle \xrightarrow{\delta}_1 \langle Q',t'\rangle}{\langle P \Box Q,t\rangle \xrightarrow{\delta}_1 \langle P' \Box Q'),t'\rangle}$

r10a : $\frac{\langle P,t\rangle \xrightarrow{\tau}_1 \langle P,t\rangle, t=0}{\langle P \sqcap Q,t\rangle \xrightarrow{\tau}_1 \langle P,t\rangle}$

r10b : $\frac{\langle Q,t\rangle \xrightarrow{\tau}_1 \langle Q,t\rangle, t=0}{\langle P \sqcap Q,t\rangle \xrightarrow{\tau}_1 \langle Q,t\rangle}$

r10c : $\frac{}{\langle P \sqcap Q,t\rangle \xrightarrow{\delta=0}_1 \langle P \sqcap Q,t\rangle}$

r11a : $\frac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P \rhd \{t_0\} Q,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}$

r11b : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle, \delta <= t_0}{\langle P \rhd \{t_0\} Q,t\rangle \xrightarrow{a}_1 \langle P' \rhd \{t_0-\delta\} Q,t'\rangle}$

r11c : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle, \delta = t_0}{\langle P \rhd \{t_0\} Q,t\rangle \xrightarrow{\delta}_1 \langle Q,t'\rangle}$

r12a : $\frac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle, a \neq b}{\langle P \triangledown b \rightarrow Q,t\rangle \xrightarrow{a}_1 \langle P' \triangledown b \rightarrow Q,t\rangle}$

r12b : $\frac{\langle Q,t\rangle \xrightarrow{a}_1 \langle Q',t\rangle, a=b}{\langle P \triangledown b \rightarrow Q,t\rangle \xrightarrow{a}_1 \langle Q,t\rangle}$

r12c : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle}{\langle P \triangledown b \rightarrow Q,t\rangle \xrightarrow{\delta}_1 \langle P' \triangledown b \rightarrow Q,t'\rangle}$

r13a : $\frac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle}{\langle P \triangledown \{t_0\} Q,t\rangle \xrightarrow{a}_1 \langle P' \triangledown \{t_0\} Q,t\rangle}$

r13b : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle, \delta <= t_0}{\langle P \triangledown \{t_0\} Q,t\rangle \xrightarrow{\delta}_1 \langle P' \triangledown \{t_0-\delta\} Q,t'\rangle}$

r13c : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle, \delta = t_0}{\langle P \triangledown \{t_0\} Q,t\rangle \xrightarrow{\delta}_1 \langle Q,t'\rangle}$

r14 : $\frac{}{\langle \mu X \bullet P,t\rangle \xrightarrow{\tau}_1 \langle P[(\mu X \bullet P)/X],t\rangle}$

r15a : $\frac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle, a \notin X}{\langle P[X]Q,t\rangle \xrightarrow{a}_1 \langle P'[X]Q,t\rangle}$

r15b : $\frac{\langle Q,t\rangle \xrightarrow{a}_1 \langle Q',t\rangle, a \notin X}{\langle P[X]Q,t\rangle \xrightarrow{a}_1 \langle P[X]Q',t\rangle}$

r15c : $\frac{\langle P,t\rangle \xrightarrow{a}_1 \langle P',t\rangle \quad \langle Q,t\rangle \xrightarrow{a}_1 \langle Q',t\rangle, a\in X}{\langle P[X]Q,t\rangle \xrightarrow{a}_1 \langle P'[X]Q',t\rangle}$

r15d : $\frac{\langle P,t\rangle \xrightarrow{\delta}_1 \langle P',t'\rangle \quad \langle Q,t\rangle \xrightarrow{\delta}_1 \langle Q',t'\rangle}{\langle P[X]Q,t\rangle \xrightarrow{\delta}_1 \langle P'[X]Q',t'\rangle}$

In order to derive observable behaviors of TCOZ processes, internal actions in the above LTS are abstracted away by introducing a new LTS as follows.

$$TS^2_{\text{TCOZ}} \mathrel{\widehat{=}} (\mathcal{C},\ \Sigma \cup \mathbb{T},\ \Longrightarrow_1)$$

Note that the set of configurations remains the same as that in TS^1_{TCOZ}, but the transition relation abstracts away from internal actions. That is, for any states c, c',

$$c \overset{a}{\Longrightarrow}_1 c' \mathrel{\widehat{=}} \exists\, c_1, c_2 \cdot c \xrightarrow{\tau}{}^*_1 c_1 \xrightarrow{a}_1 c_2 \xrightarrow{\tau}{}^*_1 c'$$
$$c \overset{\delta}{\Longrightarrow}_1 c' \mathrel{\widehat{=}} \exists\, c_1, c_2 \cdot c \xrightarrow{\tau}{}^*_1 c_1 \xrightarrow{\delta}_1 c_2 \xrightarrow{\tau}{}^*_1 c'$$

where the relation $\xrightarrow{\tau}{}^*_1$ is the sequential composition of a finite number of $\xrightarrow{\tau}_1$.

Now we construct an abstract transition system for our target formalism, Timed Automata. Note that a "normal" transition system associated with timed automata ([4, 15]) can be

$$TS^1_{\text{TA}} \mathrel{\widehat{=}} (\mathcal{S}, s_0, \Sigma^\tau \cup \mathbb{T}, \longrightarrow_2)$$

where $\mathcal{S} \mathrel{\widehat{=}} S \times V$ denotes all possible states of the transition system. Note that each state is composed of a state of the timed automaton and a clock valuation (interpretation). The initial state $s_0 = \langle i, v_0\rangle$ comprises the initial state i and a zero valuation v_0.

$\longrightarrow_2 \ \subseteq \ \mathcal{S} \times (\Sigma^\tau \cup \mathbb{T}) \times \mathcal{S}$ comprises all possible transitions of the following two kinds:

- a time passing move $\langle s, v \rangle \xrightarrow{\delta}_2 \langle s, v + \delta \rangle$.
- an action execution $\langle s, v \rangle \xrightarrow{a}_2 \langle s', v' \rangle$. It can only be performed when (1) the transition of the timed automaton $s \xrightarrow{a;\, X;\, \varphi} s'$ can be performed. (1) can only be performed when (2) and (3) holds. That includes:

  (2) the clock interpretation meets the guard. That is, $v \models \varphi$.

  (3) the new clock valuation satisfies:

  $v'(x) = 0$ for all $x \in X$;

  $v'(x) = v(x)$, for all $x \notin X$.

Based on the transition system TS^1_{TA}, a new transition system with more abstract transitions is defined as follows.

$$TS^2_{\text{TA}} \mathrel{\widehat{=}} (\mathcal{S}, s_0, \Sigma \cup \mathbb{T}, \Longrightarrow_2)$$

The only difference from TS^1_{TA} lies in the new transition relation $\Longrightarrow_2 \subseteq \mathcal{S} \times (\Sigma \cup \mathbb{T}) \times \mathcal{S}$, which abstracts away from all internal (τ) actions. That is, for states s, s',

$$s \overset{a}{\Longrightarrow}_2 s' \mathrel{\widehat{=}} \exists\, s_1, s_2 \cdot s \xrightarrow{\tau}{}^{*}_{2}\, s_1 \xrightarrow{a}_2 s_2 \xrightarrow{\tau}{}^{*}_{2}\, s'$$

$$s \overset{\delta}{\Longrightarrow}_2 s' \mathrel{\widehat{=}} \exists\, s_1, s_2 \cdot s \xrightarrow{\tau}{}^{*}_{2}\, s_1 \xrightarrow{\delta}_2 s_2 \xrightarrow{\tau}{}^{*}_{2}\, s'$$

Note that the transition relation $\xrightarrow{\tau}{}^{*}_{2}$ is the sequential composition of at most a finite number of τ-transitions $\xrightarrow{\tau}_2$.

With the above preparation, a bisimular (homomorphic) relation between TS^2_{TCOZ} and TS^2_{TA} is readily defined.

Definition 4.3.1 *(Bisimulation)*

The relation $\approx \subseteq \mathcal{C} \times \mathcal{S}$ is defined between states of TS^2_{TCOZ} and states of TS^2_{TA}. That is, for any $c \in \mathcal{C}$ and $s \in \mathcal{S}$, $c \approx s$ if and only if the following conditions hold:

- $c \stackrel{\alpha}{\Longrightarrow}_1 c'$ *implies there exists* s' *such that* $s \stackrel{\alpha}{\Longrightarrow}_2 s'$, *and* $c' \approx s'$;
- $s \stackrel{\alpha}{\Longrightarrow}_2 s'$ *implies there exists* c' *such that* $c \stackrel{\alpha}{\Longrightarrow}_1 c'$, *and* $c' \approx s'$.

The following theorem shows that our mapping rules preserve the bisimulation relation between the source and target transition systems. Since the two transition systems employ the same set of observable actions (events), the theorem thus demonstrates that each source TCOZ process and its corresponding target timed automaton are semantically equivalent.

Theorem 4.3.2 *(Correctness)*

For any TCOZ process P and its corresponding timed automaton $\mathcal{A}(P)$, $\langle P, t \rangle \approx \langle i, v_0 \rangle$ for some t, where i is the initial state of $\mathcal{A}(P)$, v_0 is the zero valuation.

Proof The correctness is proved by structural induction. Given P is a TCOZ process and $\mathcal{A}(P)$ is its translated timed automaton (please refer to chapter 3 for the definitions (D1 - D12) of all the translated timed automata for the proof of $\mathbf{R}0 - \mathbf{R}15$), here we will demonstrate that $P \approx \mathcal{A}(P)$.

- $\text{Skip} \approx \mathcal{A}(\text{Skip})$. Proof is trivial as the only transition allowed by both is a τ transitions.

- $\text{Stop} \approx \mathcal{A}(\text{Stop})$. Proof is trivial as the only transitions allowed by both are δ transitions.

- $P_{op} \approx \mathcal{A}(P_{op})$. Let P_{op} be an operation schema. P_{op} may take a δ transition and the process expression remains unchanged or take an e_{op} transition and then behave as Skip. $\mathcal{A}(P_{op})$ may take a δ transition and remain at the initial state or an e_{op} transition and then remain at the final state where only δ transitions are allowed. Therefore, $P_{op} \approx \mathcal{A}(P_{op})$.

- $\text{Wait}\ t_0 \approx wait(t_0)$. The process $\text{Wait}\ t_0$ can perform a time passing move (δ). By definition [D6], the automaton $wait(t_0)$ can also advance a corresponding δ-step.

  If $\delta < t_0$, $\langle P, t\rangle$ moves to $\langle \text{Wait}(t_0 - \delta), t + \delta\rangle$, while $\langle i, v_0\rangle$ moves to $\langle w_0, v_0 + \delta\rangle$. By hypothesis, we know $\langle \text{Wait}(t_0 - \delta), t + \delta\rangle \approx \langle w_0, v_0 + \delta\rangle$.

  If $\delta = t_0$, both $\langle P, t\rangle$ and $\langle i, v_0\rangle$ move to their terminal states and preserve the bisimulation as well.

- $P = P_1 \ \Box\ P_2$. The target automaton is $extchoice(\mathcal{A}(P_1), \mathcal{A}(P_2))$. Assume $P_j \approx \mathcal{A}(P_j)$ (j=1,2).

  The process P may perform a communication event a or a time passing move δ. By rule [**r9a**, **r9b**], we know that $\langle P, t\rangle \overset{a}{\Longrightarrow}_1 \langle P'_j, t\rangle$. By hypothesis, we know $\langle i(\mathcal{A}(P_j)), v_0\rangle \overset{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P'_j)), v_0\rangle$, and $\langle P'_j, t\rangle \approx \langle i(\mathcal{A}(P'_j)), v_0\rangle$. By definition [D12], we know $\langle i, v_0\rangle = \langle i(\mathcal{A}(P_j)), v_0\rangle$, thus $\langle i, v_0\rangle \overset{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P'_j)), v_0\rangle$.

This yields $\langle P', t\rangle \approx \langle i(\mathcal{A}(P'_j)), v_0\rangle$.

By rule [**r9c**], we know that $\langle P_1, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P'_1, t+\delta\rangle$, $\langle P_2, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P'_2, t+\delta\rangle$, where $P' = P'_1 \Box P'_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$ and $\langle P'_1, t+\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$; $\langle i(\mathcal{A}(P_2)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$ and $\langle P'_2, t+\delta\rangle \approx \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$. By definition [D12], we know $\langle i(\mathcal{A}(P_1)), v_0+\delta\rangle = \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle = \langle i(\mathcal{A}(P)), v_0+\delta\rangle$, thus $\langle i(\mathcal{A}(P)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P)), v_0+\delta\rangle$. Therefore, the automaton *extchoice*$(\mathcal{A}(P_1), \mathcal{A}(P_2))$ simulates $P_1 \Box P_2$.

Similarly, we can prove the process $P_1 \Box P_2$ simulates the translated automaton. The automaton *extchoice*$(\mathcal{A}(P_1), \mathcal{A}(P_2))$ may perform a communication event a or a time passing move δ. If $\langle i(\mathcal{A}(P)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_0\rangle$, by definition [D12], we know $\langle i(\mathcal{A}(P')), v_0\rangle = \langle i(\mathcal{A}(P'_j)), v_0\rangle$. By hypothesis, we know $\langle P_j, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P'_j, t\rangle$ and $\langle P'_j, t\rangle \approx \langle i(\mathcal{A}(P'_j)), v_0\rangle$. By rule [**r9a, r9b**], we know $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$ and $\langle P', t\rangle = \langle P'_j, t\rangle$, thus $\langle P', t\rangle \approx \langle i(\mathcal{A}(P'_j)), v_0\rangle$. This yields $\langle i(\mathcal{A}(P')), v_0\rangle \approx \langle P'_j, t\rangle$.

If $\langle i(\mathcal{A}(P)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P)), v_0+\delta\rangle$, by definition [D12], we know $\langle i(\mathcal{A}(P)), v_0+\delta\rangle = \langle i(\mathcal{A}(P_j)), v_0+\delta\rangle$. By hypothesis, we know $\langle P_j, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P'_j, t+\delta\rangle$ and $\langle P'_j, t+\delta\rangle \approx \langle i(\mathcal{A}(P_j)), v_0+\delta\rangle$. By rule [**r9c**], we know if $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, then $\langle P_j, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P'_j, t+\delta\rangle$, thus $i(\mathcal{A}(P_j)), v_0+\delta\rangle \approx \langle P', t+\delta\rangle$. This yields $i(\mathcal{A}(P)), v_0+\delta\rangle \approx \langle P', t+\delta\rangle$.

Therefore, $P_1 \Box P_2 \approx$ *extchoice*$(\mathcal{A}(P_1), \mathcal{A}(P_2))$

- $P = P_1 \sqcap P_2$. The target automaton is *intchoice*$(\mathcal{A}(P_1), \mathcal{A}(P_2))$.

If $\langle P, t\rangle \stackrel{\tau}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rules of TCOZ processes [**r10a**, **r10b**], we know that $\langle P_j, t\rangle \stackrel{\tau}{\Longrightarrow}_1 \langle P_j, t\rangle$, where $P' = P_j (j = 1, 2)$. By hypothesis, $\langle i(\mathcal{A}(P_j)), v_0\rangle \stackrel{\tau}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_j)), v_0\rangle$, and $\langle P_j, t\rangle \approx \langle i(\mathcal{A}(P_j)), v_0\rangle$. This yields $\langle P', t\rangle \approx \langle i(\mathcal{A}(P_j)), v_0\rangle$, while $\langle i(\mathcal{A}(P)), v_0\rangle \stackrel{\tau}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_j)), v_0\rangle$ is straightforward.

$\langle P', t\rangle \approx \langle i(\mathcal{A}(P')), v_0\rangle$ is straightforward when $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t\rangle$ and $\delta = 0$.

- $P = a \to P_0$. The target automaton is $\mathit{tprefix}(a, \mathcal{A}(P_0))$. The process P performs a communication event a then perform P_0.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, where $P' = P_0$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_0)), v_0\rangle$. By hypothesis, we know $\langle P', t\rangle \approx \langle i(\mathcal{A}(P_0)), v_0\rangle$.

- $P = a@u \to P_0$. The target automaton is $\mathit{tprefix}(a, u, \mathcal{A}(P_0))$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, where $P' = a@u \to P[u+\delta/u]$, while $\langle i, v_0\rangle$ moves to $\langle s_0, v_0+\delta\rangle$. By hypothesis, we know $\langle P', t+\delta\rangle \approx \langle s_0, v_0+\delta\rangle$.

  If $\langle P_0, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_0', t+\delta\rangle$, where $P' = P[0/u]$, while $\langle i, v_0\rangle$ moves to $\langle s_0, v_0+\delta\rangle$ and immediately moves to$\langle i(\mathcal{A}(P_0)), v_0\rangle$. Because $\langle i, v_0\rangle$ moves to $\langle s_0, v_0+\delta\rangle$ with an internal events τ, thus now $\langle s_0, v_0+\delta\rangle$ can be abstracted away in this automaton. By hypothesis, we know $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P_0)), v_0\rangle$.

- $P = P_0 \bullet \textsc{Deadline}\, t_0$. The target automaton is $\mathit{deadline}(\mathcal{A}(P_0), t_0)$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rule of TCOZ processes [**r6a**], we know that $\langle P_0, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_0', t\rangle$, and $P' = P_0' \bullet \textsc{Deadline}\, t$. By hypothesis,

$\langle i(\mathcal{A}(P_0)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_0')), v_0\rangle$, and $\langle P_0', t\rangle \approx \langle i(\mathcal{A}(P_0')), v_0\rangle$. This yields $\langle P', t\rangle \approx \langle i(\mathcal{A}(P')), v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_0\rangle$ is straightforward.

If $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, where $P' = P_0' \bullet \text{DEADLINE}(t_0 - \delta)$, from the operational rule of TCOZ processes [**r6b**], we know that $\langle P_0, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_0', t+\delta\rangle$. By hypothesis, $\langle i(\mathcal{A}(P_0)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_0')), v_1\rangle$ and $\langle P_0', t+\delta\rangle \approx \langle i(\mathcal{A}(P_0')), v_1\rangle$, where $v_1 = v_0 + \delta$. By a simple displacement of the specific clock x with $x + \delta$, we obtain $\langle i, v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_1'\rangle$, where $v_1' = v_1 \oplus \{x \mapsto 0\}$, and $\langle P', t + \delta\rangle \approx \langle i(\mathcal{A}(P')), v_1'\rangle$.

- $P = P_0 \bullet \text{WAITUNTIL}\, t_0$. The target automaton is $\mathit{waituntil}(\mathcal{A}(P_0), t_0)$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rule of TCOZ processes [**r7a**], it is easy to know that $\langle P_0, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_0', t\rangle$, and $P' = P_0' \bullet \text{WAITUNTIL}\, t$. By hypothesis, $\langle i(\mathcal{A}(P_0)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_0')), v_0\rangle$, and $\langle P_0', t\rangle \approx \langle i(\mathcal{A}(P_0')), v_0\rangle$. This yields $\langle P', t\rangle \approx \langle i(\mathcal{A}(P')), v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_0\rangle$ is straightforward.

  If $\delta < t_0$ and $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t + \delta\rangle$, where $P' = P_0' \bullet \text{WAITUNTIL}(t_0 - \delta)$, from the operational rule of TCOZ processes [**r7b**], we know that $\langle P_0, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_0', t+\delta\rangle$. By hypothesis, $\langle i(\mathcal{A}(P_0)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_0')), v_1\rangle$ and $\langle P_0', t+\delta\rangle \approx \langle i(\mathcal{A}(P_0')), v_1\rangle$, where $v_1 = v_0 + \delta$. By a simple displacement of the specific clock x with $x+\delta$, we obtain $\langle i, v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_1'\rangle$, where $v_1' = v_1 \oplus \{x \mapsto 0\}$, and $\langle P', t + \delta\rangle \approx \langle i(\mathcal{A}(P')), v_1'\rangle$.

  If $\delta >= t_0$ and $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, where $P' = P_0'$, from the operational rule

of TCOZ processes [**r7c**], we know that $\langle P_0, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_0', t{+}\delta\rangle$. By hypothesis, $\langle P_0', t+\delta\rangle \approx \langle i(\mathcal{A}(P_0')), v_0+\delta\rangle$, This yields $\langle P', t+\delta\rangle \approx \langle P_0', t+\delta\rangle \approx \langle i(\mathcal{A}(P')), v_0+\delta\rangle$, while $\langle i, v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P')), v_0+\delta\rangle$ is straightforward.

- $P = P_1;\ P_2$. The target automaton is $seq(\mathcal{A}(P_1), \mathcal{A}(P_2))$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rules of TCOZ processes [**r8a**, **r8b**], we know that $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$; or $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle \epsilon, t\rangle$.

  If $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$, where $P' = P_1';\ P_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$, and $\langle P_1', t\rangle \approx \langle s_0, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$ is straightforward.

  If $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle \epsilon, t\rangle$, where $P' = P_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle e(\mathcal{A}(P_1)), v_0\rangle$, and $\langle P_1', t\rangle \approx \langle e(\mathcal{A}(P_1)), v_0\rangle$. According to the sequential composition pattern, $\langle e(\mathcal{A}(P)), v_0\rangle \stackrel{\tau}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0\rangle$. This yields $\langle P', t\rangle \approx \langle e(\mathcal{A}(P)), v_0\rangle \approx \langle i(\mathcal{A}(P_2)), v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0\rangle$ is straightforward.

  If $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, from the operational rule of TCOZ processes [**r8c**], we know that $\langle P_1, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_1', t+\delta\rangle$, where $P' = P_1';\ P_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0{+}\delta\rangle$ and $\langle P_1', t{+}\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0{+}\delta\rangle$. This yields $\langle P', t{+}\delta\rangle \approx \langle i(\mathcal{A}(P)), v_0{+}\delta\rangle$, while $\langle i(\mathcal{A}(P)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P)), v_0{+}\delta\rangle$ is straightforward.

- $P = P_1 \rhd \{t_0\} P_2$. The target automaton is $timeout(\mathcal{A}(P_1), \mathcal{A}(P_2), t_0)$. The process P may perform a communication event a or a time passing move δ.

If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rule of TCOZ processes [**r11a**], we know that $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$, where $P' = P_1'$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$, and $\langle P_1', t\rangle \approx \langle s_0, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$ is straightforward.

If $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, from the operational rules of TCOZ processes [**r11b**, **r11c**], we know that $\langle P_1, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_1', t+\delta\rangle$. If $\delta <= t_0$, we know $P' = P_1' \rhd \{t_0 - \delta\}P_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$ and $\langle P_1', t+\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$. According to the timeout pattern, $\langle i, v_0+\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i, v_0+\delta\rangle$ while $\langle i, v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i, v_0+\delta\rangle$ is straightforward. If $\delta = t_0$, then $P' = P_2$. According to the timeout pattern, we know $\langle i, v_0\rangle \approx \langle i(\mathcal{A}(P_1)), v_0\rangle$ and $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$ while $\langle i, v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$ is straightforward.

- $P = P_1 \nabla b \rightarrow P_2$. The target automaton is *einterrupt*$(\mathcal{A}(P_1), \mathcal{A}(P_2), b)$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rules of TCOZ processes [**r12a**, **r12b**], we know that $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$, If $\{a\} \neq \{b\}$, we know $P' = P_1' \nabla b \rightarrow P_2$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$, and $\langle P_1', t\rangle \approx \langle s_0, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$ is straightforward. If $\{a\} = \{b\}$, then $P' = P_2$. According to the event interrupt pattern, we know $\langle i, v_0\rangle \approx \langle i(\mathcal{A}(P_1)), v_0\rangle$ and $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$, while $\langle i, v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$ is straightforward.

If $\langle P, t\rangle \overset{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, from the operational rule of TCOZ processes [**r12c**], we know that $\langle P_1, t\rangle \overset{\delta}{\Longrightarrow}_1 \langle P_1', t+\delta\rangle$. we know $P' = P_1' \nabla b \rightarrow P_2$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \overset{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$ and according to the event interrupt pattern, $\langle i, v_0\rangle \approx \langle i(\mathcal{A}(P_1)), v_0\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i, v_0+\delta\rangle$ while $\langle i, v_0\rangle \overset{\delta}{\Longrightarrow}_2 \langle i, v_0+\delta\rangle$ is straightforward.

- $P = P_1 \nabla\{t\} P_2$. The target automaton is *tinterrupt*$(\mathcal{A}(P_1), \mathcal{A}(P_2), t_0)$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \overset{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rule of TCOZ processes [**r13a**], we know that $\langle P_1, t\rangle \overset{a}{\Longrightarrow}_1 \langle P_1', t\rangle$, where $P' = P_1' \nabla\{t\} P_2$. By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \overset{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$, and $\langle P_1', t\rangle \approx \langle s_0, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$, while $\langle i, v_0\rangle \overset{a}{\Longrightarrow}_2 \langle s_0, v_0\rangle$ is straightforward.

  If $\langle P, t\rangle \overset{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, from the operational rules of TCOZ processes [**r13b**, **r13c**], we know that $\langle P_1, t\rangle \overset{\delta}{\Longrightarrow}_1 \langle P_1', t+\delta\rangle$. If $\delta <= t_0$, we know $P' = P_1' \nabla\{t_0 - \delta\} P_2$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \overset{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$ and $\langle P_1', t+\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i, v_0+\delta\rangle$ while $\langle i, v_0\rangle \overset{\delta}{\Longrightarrow}_2 \langle s_0, v_0+\delta\rangle$ is straightforward. If $\delta = t_0$, then $P' = P_2$. According to the timeout pattern, we know $\langle i, v_0\rangle \approx \langle i(\mathcal{A}(P_1)), v_0\rangle$ and $\langle i(\mathcal{A}(P_1)), v_0\rangle \overset{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle \approx \langle i, v_0+\delta\rangle$.

- $P = \mu N \bullet P(N)$. The target automaton is *recursion*$(\mathcal{A}(P(N)), N)$.

  If $\langle P, t\rangle \overset{\tau}{\Longrightarrow}_1 \langle P', t\rangle$, where $P' = P[(\mu N \bullet P)/N]$, By hypothesis, we know that $\langle i(\mathcal{A}(P(N))), v_0\rangle \overset{\tau}{\Longrightarrow}_2 \langle s_0, v_0\rangle$, and $\langle P'(N), t\rangle \approx \langle s_0, v_0\rangle$. According to

the definition of recursion, we know $P'(N) = P'$, this yields $\langle P', t\rangle \approx \langle s_0, v_0\rangle$ while $\langle i, v_0\rangle \approx \langle s_0, v_0\rangle$ is straightforward.

Similarly $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P)), v_0+\delta\rangle$ if $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$

- $P = P_1 \mid[X]\mid P_2$. The target automaton is $\mathcal{A}(P_1) \parallel \mathcal{A}(P_2)$. The process P may perform a communication event a or a time passing move δ.

  If $\langle P, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P', t\rangle$, from the operational rules of TCOZ processes [**r15a**, **r15b**, **r15c**], we know that:

  $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$ and $a \notin X$, where $P' = P_1' \mid[X]\mid P_2$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_1, v_0\rangle$ and $\langle P_1', t\rangle \approx \langle s_1, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_1, v_0\rangle$.

  Or $\langle P_2, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_2', t\rangle$ and $a \notin X$, where $P' = P_1 \mid[X]\mid P_2'$, By hypothesis, $\langle i(\mathcal{A}(P_2)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_1, v_0\rangle$ and $\langle P_2', t\rangle \approx \langle s_1, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle s_1, v_0\rangle$.

  Or $\langle P_1, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_1', t\rangle$, $\langle P_2, t\rangle \stackrel{a}{\Longrightarrow}_1 \langle P_2', t\rangle$ and $a \in X$, where $P' = P_1' \mid[X]\mid P_2'$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{a}{\Longrightarrow}_2 \langle s_1, v_0\rangle$ and $\langle P_1', t\rangle \approx \langle s_1, v_0\rangle$, similarly $\langle P_2', t\rangle \approx \langle s_2, v_0\rangle$. This yields $\langle P', t\rangle \approx \langle i(\mathcal{A}(P')), v_0\rangle$ while $\langle i(\mathcal{A}(P')), v_0\rangle \approx \langle (s_1, s_2), v_0\rangle$ is straightforward.

  If $\langle P, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P', t+\delta\rangle$, from the operational rule of TCOZ processes [**r15d**], we know that $\langle P_1, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_1', t+\delta\rangle$ and $\langle P_2, t\rangle \stackrel{\delta}{\Longrightarrow}_1 \langle P_2', t+\delta\rangle$, where $P' = P_1' \mid[X]\mid P_2'$, By hypothesis, $\langle i(\mathcal{A}(P_1)), v_0\rangle \stackrel{\delta}{\Longrightarrow}_2 \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$ and $\langle P_1', t+\delta\rangle \approx \langle i(\mathcal{A}(P_1)), v_0+\delta\rangle$, similarly $\langle P_2', t+\delta\rangle \approx \langle i(\mathcal{A}(P_2)), v_0+\delta\rangle$. This yields $\langle P', t+\delta\rangle \approx \langle i(\mathcal{A}(P')), v_0+\delta\rangle$ while $\langle i(\mathcal{A}(P')), v_0+\delta\rangle \approx$

$\langle (i(\mathcal{A}(P_1)), i(\mathcal{A}(P_2))), v_0 + \delta \rangle$ is straightforward.

□

4.4 An Example: Railroad Crossing System

In this section, we will use a Railroad Crossing System (RCS) specified in TCOZ as a driving example to illustrate our approach to model-checking TCOZ models of real-time systems. The concept of the Railroad Crossing Problem was primarily evolved by Heitmeyer [36]. It is a system which has a controller to operate a gate at a railroad crossing safely. Based on the above features, we define some assumptions and constraints as follows:

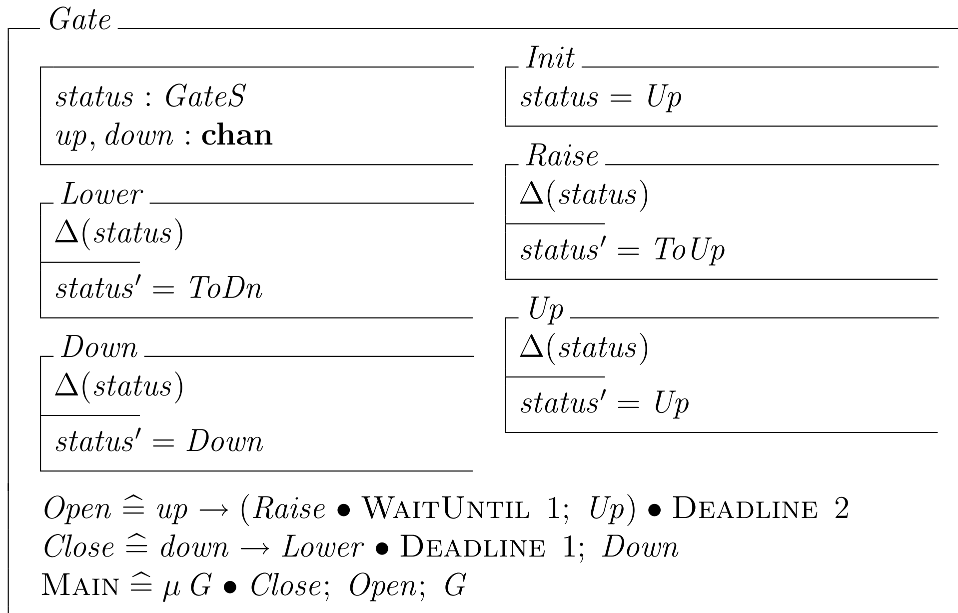
1. The train sends a signal to the controller at least 3 time units before it enters the crossing, stays there no more than 2 time units and sends another signal to the controller upon exiting the crossing.
2. The controller commands the gate to lower exactly 1 time unit after it has received the approaching signal from the train and commands the gate to rise again no more than 1 time unit after receiving the exiting signal.
3. The gate takes less than 1 time unit to come down and between 1 and 2 time units to come up.

TCOZ Model of RCS

According to the requirement description, an RCS consists of three components: a central controller, a train, and a gate to control the traffic. The following provides the formal specification of *Gate*, *Train* and *Controller* class in TCOZ.

Gate: The essential behaviors of this railroad crossing gate are to open and close itself according to its external commands (events) *up* and *down*.

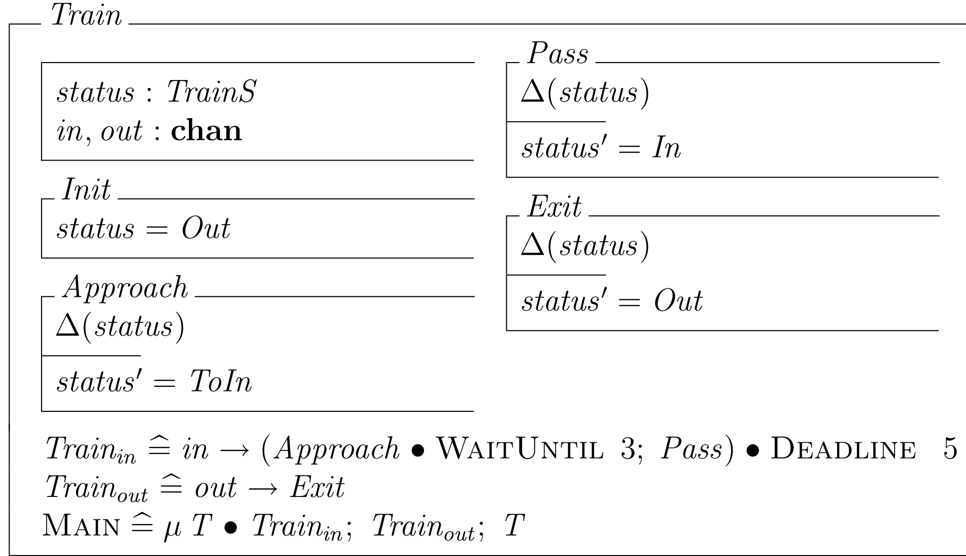
$GateS ::= ToUp \mid Up \mid ToDn \mid Down$



The interface of the *Gate* class is defined through channels *up* and *down*. The DEADLINE and WAITUNTIL expressions are used here to capture its timing properties, which constrain that the gate takes less than 1 time unit to come down and between 1 and 2 time units to come up.

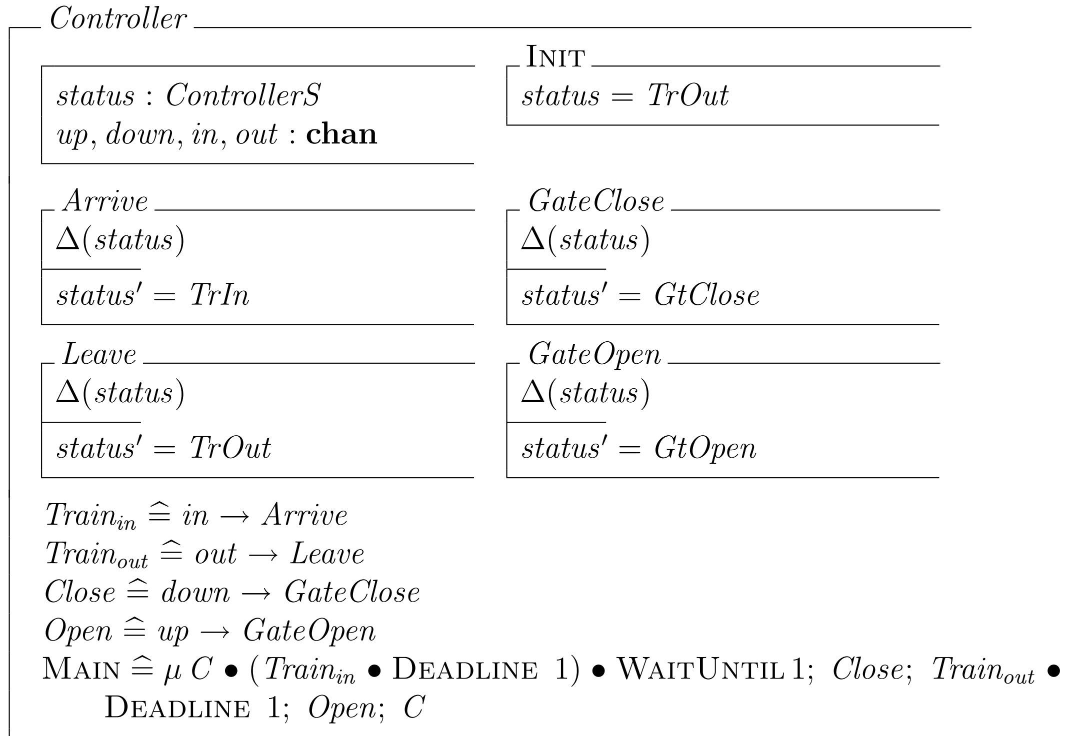
Train: The basic behavior of the train component is to communicate with the controller with its passing information.

TrainS ::= *ToIn* | *In* | *Out*



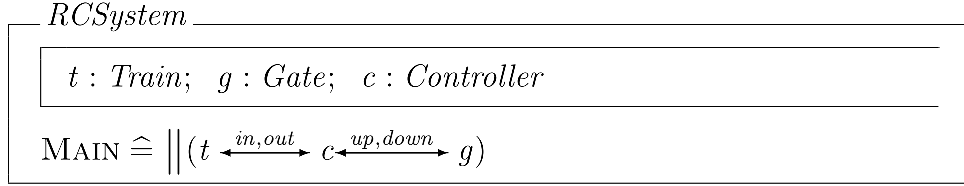
Central Controller: The central controller is the crucial part of the system, which actively communicates with the train, and gate. The *Controller* class is modelled as follows:

ControllerS ::= *TrIn* | *TrOut* | *GtClose* | *GtOpen*



The attribute *status* keeps the records of the train's passing information in the system. When the train sends an *in* signal, the *status* of the controller changes from *TrOut* to *TrIn*. When the train has passed the crossing and sent an *out* signal to the controller, the *status* of the controller changes from *TrIn* to *TrOut*. The main processes of the controller are receiving the train passing information and manipulating the gate operations at the same time. If the gate is open then instructions on closing the gate will be sent to the *Gate*. On the other hand, when the train has passed the gate, the controller will open the gate.

RCS Configuration: After specifying individual components, the next step is to compose them into a whole system. The overall system is a composition of all the communicating components.



Two essential properties of RCS are: first, the gate is never closed at a stretch for more than a stipulated time range (suppose 10 time units); second, the gate should be down whenever a train is crossing. These properties can be formally expressed as:

$RCSystem \bullet \Box(g.status = ToDn \rightarrow \Diamond_{\leq 10}\ g.status = Up)$

$RCSystem \bullet t.status = In \Rightarrow g.status = Down$

Translation

In this section, we show how the given translation rules can be applied to map TCOZ specification into a timed automata.

First of all, for the whole RCS system, three automata can be identified in the Timed Automata model, i.e., gate, train and controller. We use the gate class as an example to show the identification of the states, transitions, guards and synchronization mentioned above. According to the translation rules for TCOZ classes/objects, four states can be identified through the static view of the *Gate* class. It has four operation schemas. Each one is mapped into a state, namely, *Up*, *ToDown*, *Down*, and *ToUp*, among which *Up* is the initial state as indicated by the *INIT* schema in the *Gate* class. Synchronization and clock conditions on

the transitions are constructed by transforming the *Open* and *Close* processes of the *Gate* class according to the translation rules on DEADLINE and WAITUNTIL primitives. A clock is generated to guard the atomic process *Lower* to finish no later than 1 time unit, then it is reused to guard *Raise* and *Up* process to meet their timing constraints by resetting its value to 0. The initial and terminal states generated for every non-atomic process due to those translation rules, if they are linked by a transition with a τ event, are incorporated into one state to simplify the resultant automaton.

This gate automaton can be automatically generated by our translation tool and visualized in UPPAAL as "process gate" in Figure 4.1. In the same way, we can get the train and controller automata as "process train" and "process controller".

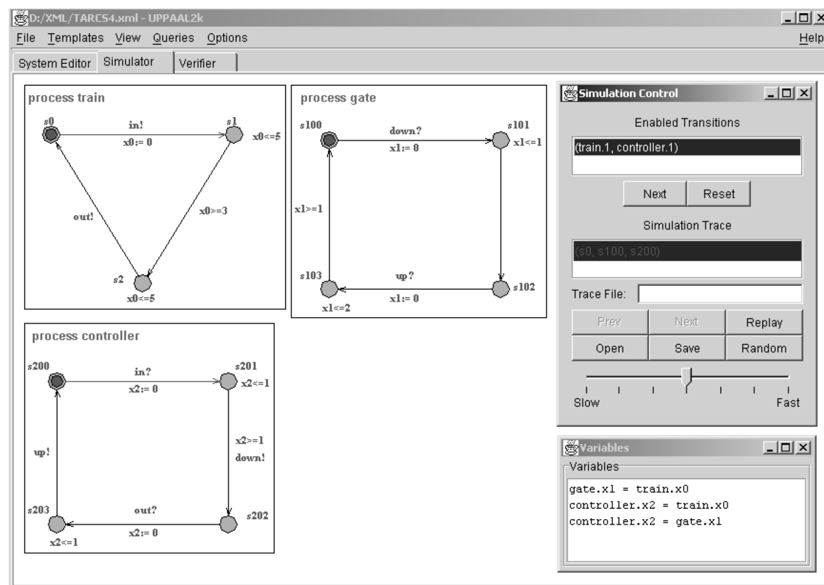


Figure 4.1: Simulation

Model-checking RCS

Now we can use the UPPAAL tool to simulate the system as well as to model-check some invariants and real-time properties. In UPPAAL correctness criteria can be specified as formulas of the timed temporal logic TCTL [37], for which UPPAAL implements model-checking algorithms.

From a safety critical perspective, the key point of the RCS is to provide guaranteed safe and efficient services. Examples of these properties can be formally described in our model as:

- safety property - whenever the train is in, the gate is down. It can be translated into a TCTL formula in UPPAAL as follows:

```
A[] train.s2 imply gate.s102
```

- efficient service property - the gate is never closed at a stretch for more than 10 time units. To verify this property, we add a clock x to record the time the gate takes to reopen itself:

```
gate.s101 --> (gate.s100 and gate.x<=10)
```

UPPAAL verified that these properties actually hold for this given model.

4.5 Tool Support

The translation process is automated by employing XML/XSL [73] technology. In our previous work, the syntax of Z family languages, i.e., Z/Object-Z/TCOZ, has been defined using XML Schema and supported by a Z family Mark-up Language (ZML) [71] tool. As the UPPAAL tool can read an XML representation of Timed Automata, we developed a tool to automatically project the TCOZ model (in ZML) to TA model (in UPPAAL XML). Our prototype is programmed in Java.

The main process and techniques for the transformation tool we developed are depicted in Figure 4.2. The tool takes in a TCOZ specification represented in XML, and outputs an XML representation of a Timed Automata specification which has its own defined style file DTD by UPPAAL. The formal Z definitions of timed patterns and the projection rules are used as a design document and applied recursively during the implementation of the transformation. Building on the strength of ZML, the automatic transformation can make use of the XML parser Xerces [20] to easily abstract information from the specification.

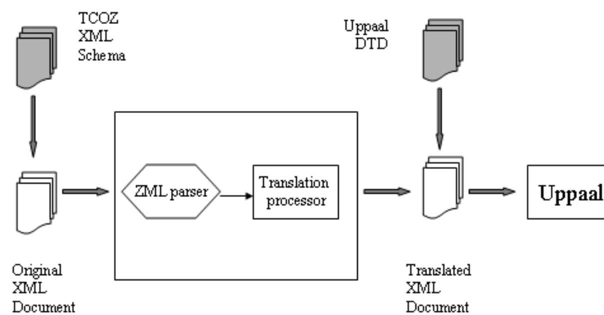


Figure 4.2: TCOZ to UPPAAL diagram

The transformation is achieved firstly by implementing a ZML parser, which will take in a ZML specification and build a virtual model of the system in the memory. A TA interface is then built according to the UPPAAL document structure, e.g., each TA document contains multiple templates and each template contains a set of states, a set of transitions between states and transition conditions. A transformation module is built to get information from the ZML parser, apply the right transformation rule and feed the outcome of the transformation to the TA interface. Note that TCOZ process expressions can be defined recursively, i.e., a process expression may contain one or more other process expressions. Our transformation modules are built to take care of all valid TCOZ specifications and the transformation rules are applied recursively.

In the translation process, two kinds of abstraction can be applied to alleviate the problem of state explosion. One is to reduce the number of translated TA states by incorporating two or more consecutive intermediate states. These intermediate states can be identified with an outgoing transition labelled by an internal τ event. In UPPAAL, these states are mapped as urgent states [48], marked with 'U' representing that no delay is allowed. This kind of simplification is automated by our translation tool. Another kind of simplification is to reuse clocks instead of defining a fresh clock for each time constraint. It is currently done by manually examining whether a clock can be reused to specify more than one timing constraint.

Note that UPPAAL also adopts channels as its synchronization mechanism for the interaction between automata. This is equivalent to the CSP channels in TCOZ

for pairwise communications, the translation of which thus can be accomplished directly by mapping the channel events from a TCOZ model to its TA model. Hence it is not necessary to define a specific parallel composition pattern. Worthy of mention is that for the translation of multiple synchronization of CSP in TCOZ model, committed states[2] need to be used to achieve broadcasting communication in the target UPPAAL model (Detailed information can be found in [48]). Value passing in UPPAAL is done by global variable assignments on the TA transitions, which corresponds to TCOZ channel input/output. One may use integer variables and arrays of integers, each with a bounded domain and an initial value. Predicates over the integer variables can be used as guards on the edges of an automaton process. Thus, attributes of the similar types in a TCOZ model can also be transformed into their corresponding TA model. Variables are updated according to the values of the post states of the corresponding TCOZ operations. However,currently the mapping of data variables from a TCOZ model to its UPPAAL model is done by manual work due to the mismatch of data types between the two languages.

The outcome of our transformation tool is UPPAAL's XML representation of TA, which is ready to be taken as input for verification and simulation. For example, the following is part of the TCOZ XML representation of the *Gate* class :

```
<! The following is the Gate Class in TCOZ. > <classDef>
```

[2] An outgoing transition has to be taken instantaneously in committed states and an automaton in a committed location blocks both time progress as well as enabled transitions in all other automata.

```
<name>Gate</name>
 ...
<state> ... </state>
 <operation><name>MAIN</name>
   <processExpr>
      <mu>G</mu>
      <processExpr>
      <processExpr><simpleProExp>Close</simpleProExp>
      </processExpr>
      <proConnSym>composition</proConnSym>
      <processExpr><simpleProExp>Open</simpleProExp>
      </processExpr>
      </processExpr>
   </processExpr>
 </operation>
</classDef>
```

The transformation tool takes in this *gate* class and generates the following *gate* automaton.

```
- <template>
  <name>Gate</name>
  <declaration>clock y;</declaration>
- <location id="id10" >
```

```
  <name>ToUp</name>
  <label kind="invariant"> y<=2 </label>
  </location>
  ...
- <transition>
  <source  ref="id5" />
  <target  ref="id7" />
  <label  kind="assignment" >y:=0</label>
  <label  kind="synchronisation">down?</label>
  </transition>
  ...
  </template>
```

Following the style file [1] defined by UPPAAL, this XML code can be directly visualized in UPPAAL and is ready for simulation and model-checking.

4.6 Conclusion

Formal Methods have been demonstrated to be effectively applicable in the industrial development of complex real-time systems. Nevertheless, formal methods such as TCOZ (Timed Communicating Object-Z), are not widely used in industry. If the full potential of such languages is to be realized, they need to be supplemented with software tools to assist with the more taxing mathematical aspects, such as

the detection of specification errors and the verification of system properties.

In this section, we have proposed an approach to model-check TCOZ via projecting TCOZ models to TA models so that tools associated with TA can be used to do the verification. The transformation rules are provided in detail to define the relation between a TCOZ class, i.e., its static and dynamic components, and a timed automaton, i.e., its states and transitions. And a translation tool has been implemented to automate the projection process.

We also investigated the semantic equivalence issue between TCOZ processes and timed automata and provided a full proof. Little theoretical work has been done in this area except that Joel and James in their recent work [55] have demonstrated that Timed CSP has equal expressiveness with closed timed automata.

Since TCOZ is a superset of Timed CSP, one consequence of this work is that a semantic link and a practical translation tool from Timed CSP to TA has been achieved so that TA tools, i.e., UPPAAL, can also be used to check Timed CSP timing properties. In this context, this work complements the recent pure theoretical investigation [55] on the expressiveness of Timed CSP and closed timed automata.

Chapter 5

Case Study: Multi-terminal Railcar System

In this chapter, we will use a much more complex real-time system, i.e., a Multi-terminal Railcar System (MRS) to further demonstrate the applicability of our approach to modelling and checking real-time complex systems. The system was primarily evolved from the railcar system [33] by D. Harel and E. Grey with additional timing constraints.

In the MRS system, there are four terminals which are located in a cyclic path. Each terminal contains a push button for passengers to place their requests for car service. A railcar travels clockwise on the track to transport passengers between the terminals; it is equipped with a cruising controller for maintaining speed and a destination board to record internal passenger requests and to help calculation of the destination terminal. There is also a control center which receives, processes and sends data to the terminals and the railcar so that external requests from any terminal can be fulfilled.

Possible scenarios including the timing issues, stated as customer requirements, are:

Railcar approaching terminal: When the railcar is approaching a terminal, it sends an approach request 20 seconds in advance to the terminal to prepare for its parking. After it receives the approach acknowledgment from the terminal, it will check both its internal requests and external requests to decide whether it should stop at the terminal or pass through directly.

Railcar departing terminal: When the railcar decides to leave a terminal, it sends a depart signal to the terminal, the terminal then prepares for the railcar to depart and responds to it within 5 seconds, the railcar then leaves and starts cruising to its next destination.

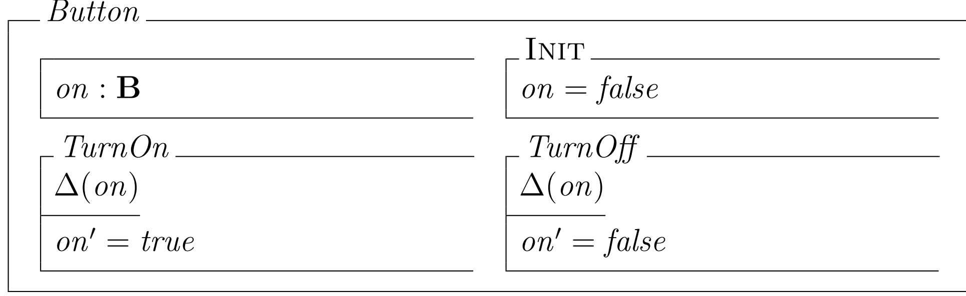
Railcar loading passenger: When the railcar comes to a stop at a certain terminal, it will open its door for 10 seconds, after that it will close the door, and begins to wait for either internal requests or an external request dispatched from the controller. Passengers inside the railcar are given 5 seconds to make an internal request before the railcar accepts any external requests.

Passenger in terminal: When a passenger in a terminal wishes to travel to some destination terminal, and there is no available railcar, the passenger pushes the button in the terminal and waits until the railcar arrives.

5.1 TCOZ Model of MRS

According to the requirements, the MRS has the following components: a controller, terminals, and a railcar, which is also composed of three basic components: a car destination panel, a car door and a car handler.

Buttons: A basic component of the MRS system is the button panel inside the railcar and on the terminals. The behavior of buttons is modeled as follows:



Button

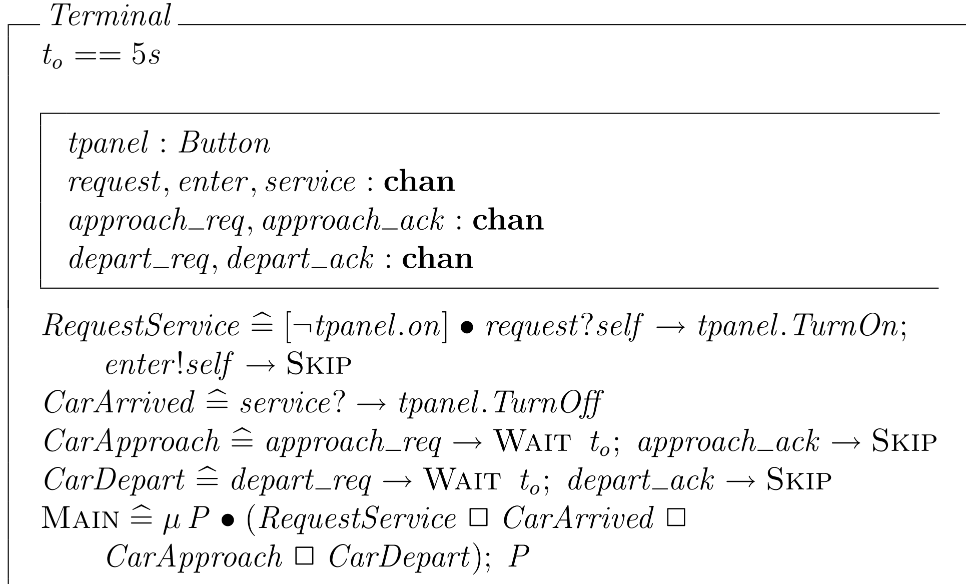
$on : \mathbf{B}$

INIT: $on = false$

TurnOn: $\Delta(on)$; $on' = true$

TurnOff: $\Delta(on)$; $on' = false$

Terminal: The class *Terminal* has the following behaviors: it records external requests from passengers through channel *request* and passes the request information to the controller through channels *enter* and *service*; and it communicates with the railcars through *approach_req*, *approach_ack*, *depart_req*, and *depart_ack* channels.



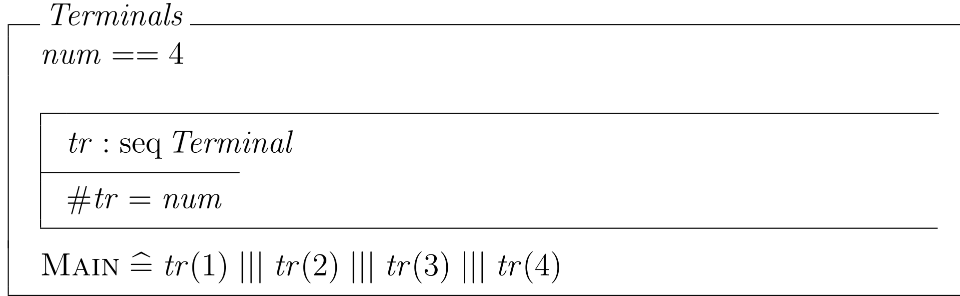
Terminal

$t_o == 5s$

$tpanel : Button$
$request, enter, service : \mathbf{chan}$
$approach_req, approach_ack : \mathbf{chan}$
$depart_req, depart_ack : \mathbf{chan}$

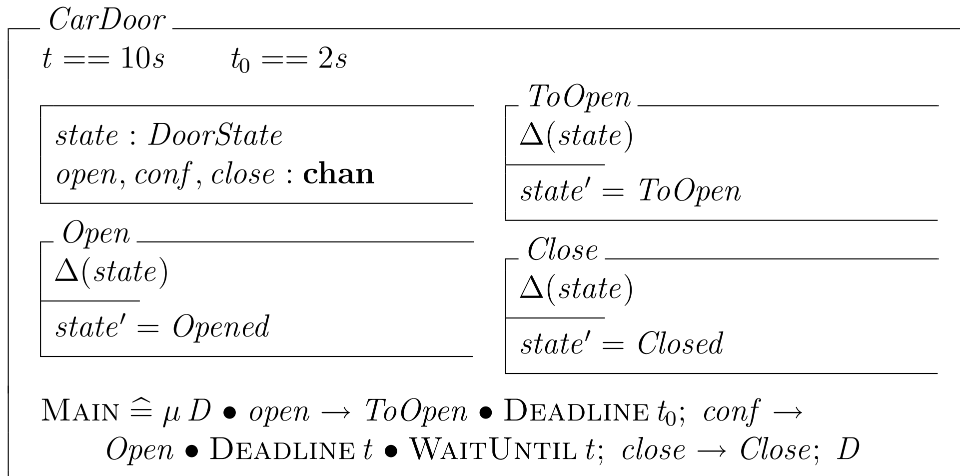
$RequestService \mathrel{\widehat=} [\neg tpanel.on] \bullet request?self \rightarrow tpanel.TurnOn;$
$\quad enter!self \rightarrow \text{SKIP}$
$CarArrived \mathrel{\widehat=} service? \rightarrow tpanel.TurnOff$
$CarApproach \mathrel{\widehat=} approach_req \rightarrow \text{WAIT}\ t_o;\ approach_ack \rightarrow \text{SKIP}$
$CarDepart \mathrel{\widehat=} depart_req \rightarrow \text{WAIT}\ t_o;\ depart_ack \rightarrow \text{SKIP}$
$\text{MAIN} \mathrel{\widehat=} \mu P \bullet (RequestService \Box CarArrived \Box$
$\quad CarApproach \Box CarDepart);\ P$

self is an object identity attribute which can be included in the communication to tell the environment which object it is communicating with.



Car Door: The car door is treated as a separate class so as to ensure a clear description of its timing and safety property. In our model, it takes t_0 time units for the door to open when it receives the *open* command from the car handler. Once the door is open, it will inform the handler by sending a *conf* message immediately and remain open for t time units before closing.

$DoorState ::= ToOpen \mid Opened \mid Closed.$



Car Handler: The car handler maintains the state of the railcar, keeps track of its current location, checks and decides whether to stop when passing a terminal and provides the interface between the railcar environment and the other railcar components.

$CarState ::= Idle \mid Ready \mid Departing \mid Approaching \mid Cruising$

CarHandler

$t_1 == 5s \quad t_2 == 15s \quad t_3 == 20s \quad t_4 == 5s \quad num == 4$

$tm, destination : \mathbb{N}$
$ccur, tcur : \mathbb{B}$
$state : CarState$
$open, conf, close : \mathbf{chan}$
$int_sched, int_serv : \mathbf{chan}$
$select, service : \mathbf{chan}$
$cstop, tstop : \mathbf{chan}$
$approach_req : \mathbf{chan}$
$approach_ack : \mathbf{chan}$
$depart_req, depart_ack : \mathbf{chan}$

INIT
$state = Idle \wedge tm = 1$
$tcur = false \wedge ccur = false$

Approaching
$\Delta(state)$
$state' = Approaching$

Departing
$\Delta(state)$
$state' = Departing$

Cruising
$\Delta(state)$
$state' = Cruising$

ToStop
$\Delta(tcur, ccur)$
$tsuc?, csuc? : \mathbb{B}$
$tcur' = tsuc \wedge ccur' = csuc$

Ready
$\Delta(state, destination)$
$dest? : N$
$state' = Ready$
$destination' = dest$

SetTm
$\Delta(tm)$
$tm' = tm \bmod num + 1$

$Depart \mathrel{\widehat=} depart_req \rightarrow Departing \bullet \text{WAITUNTIL}\ t_1 \bullet \text{DEADLINE}\ t_1;$
$\quad depart_ack? \rightarrow \text{SKIP}$
$Approach \mathrel{\widehat=} approach_req \rightarrow$
$\quad (approach_ack? \rightarrow \text{SKIP}) \bullet \text{WAITUNTIL}\ t_1 \bullet \text{DEADLINE}\ t_1;$
$\quad Approaching \bullet \text{WAITUNTIL}\ t_2 \bullet \text{DEADLINE}\ t_3$
$Check \mathrel{\widehat=} tstop!tm \rightarrow [tsuc : \mathbb{B}] \bullet tstop?tsuc \rightarrow cstop!tm \rightarrow$
$\quad [csuc : \mathbb{B}] \bullet cstop?csuc \rightarrow ToStop;$
$Move \mathrel{\widehat=} \mu\, M \bullet [destination \neq tm \wedge \neg ccur \wedge \neg tcur] \bullet Depart;$
$\quad Cruising;\ SetTm;\ Approach;\ Check;\ M$
$Handle \mathrel{\widehat=} Move \Box [destination = tm \vee ccur \vee tcur] \bullet open \rightarrow$
$\quad conf \rightarrow service!tm \rightarrow int_serv! \rightarrow close \rightarrow \text{SKIP}$
$\text{MAIN} \mathrel{\widehat=} \mu\, CH \bullet (int_sched!tm \rightarrow int_sched?dest \rightarrow \text{SKIP}) \rhd \{t_4\}$
$\quad ((int_sched!tm \rightarrow int_sched?dest \rightarrow \text{SKIP}) \Box$
$\quad (select!tm \rightarrow select?dest \rightarrow \text{SKIP}));\ Ready;\ Handle;\ CH$

The attribute *CarState* is a free type variable which defines the five basic states of the railcar; *tm* is an attribute which records the current terminal where the railcar

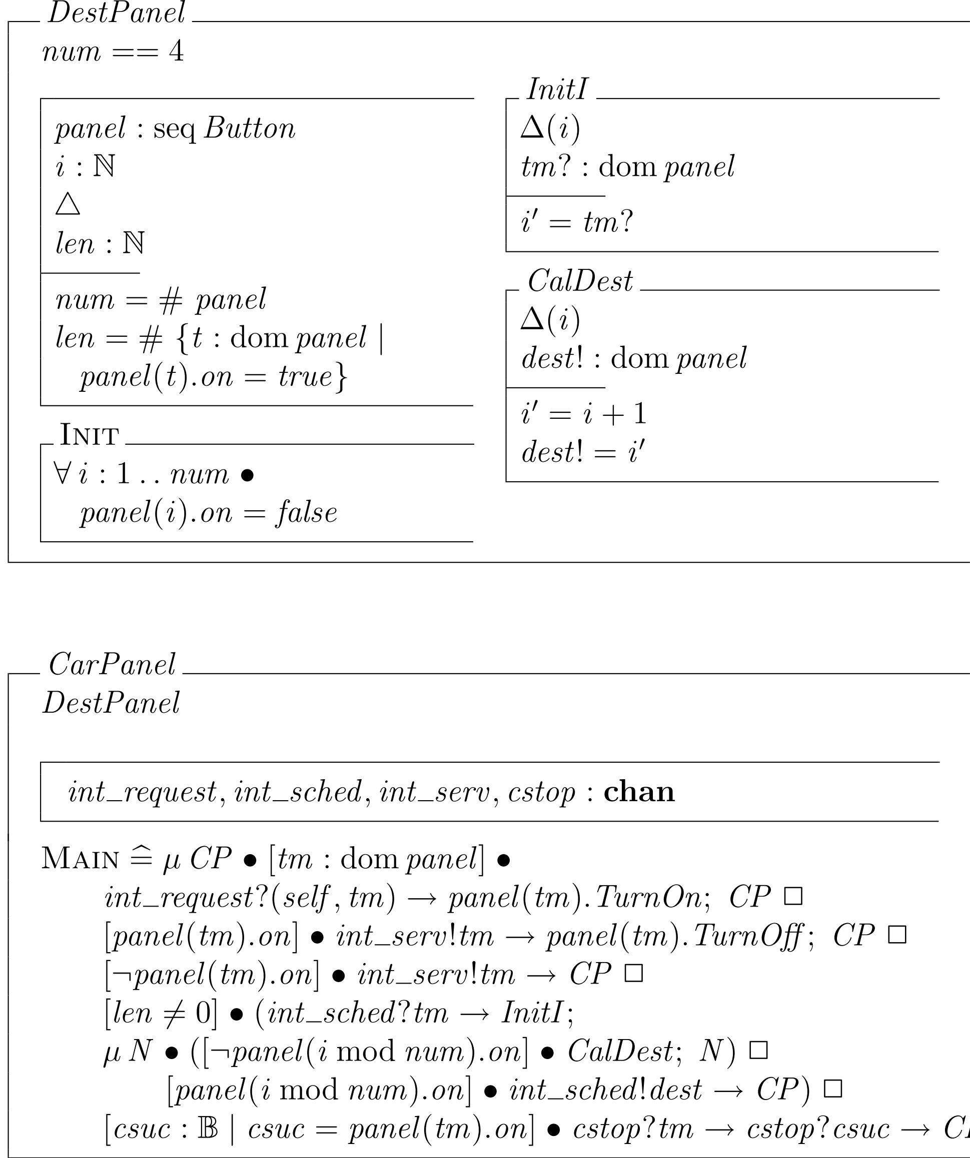
is. It will be updated right before the railcar sends the request *approach_req* to enter the terminal; *destination* records the next destination indicated by the car destination panel or the controller; *ccur* and *tcur* are two Boolean variables used to decide whether the railcar should stop when passing a terminal.

Initially, the railcar idles at a terminal. *CarHandler* will quest for a destination from the *CarPanel* first through channel *int_sched* for 5 seconds before the it quests for any external requests from the controller through channel *select*. Once the car destination panel or the controller has indicated the next destination, the state of the car will be set as *Ready* and it will start to move to its new destination. During this period, when coming to each intermediate terminal, the car handler will check the car destination panel and the controller through channel *cstop* and *tstop* to see whether there are any new requests from passengers, then update the value of *ccur* and *tcur* accordingly through the operation *ToStop*. If the car comes to stop at a terminal, its door will be opened, the car destination panel and the central controller will be notified and then once the door is closed, control will be returned to the initial mode.

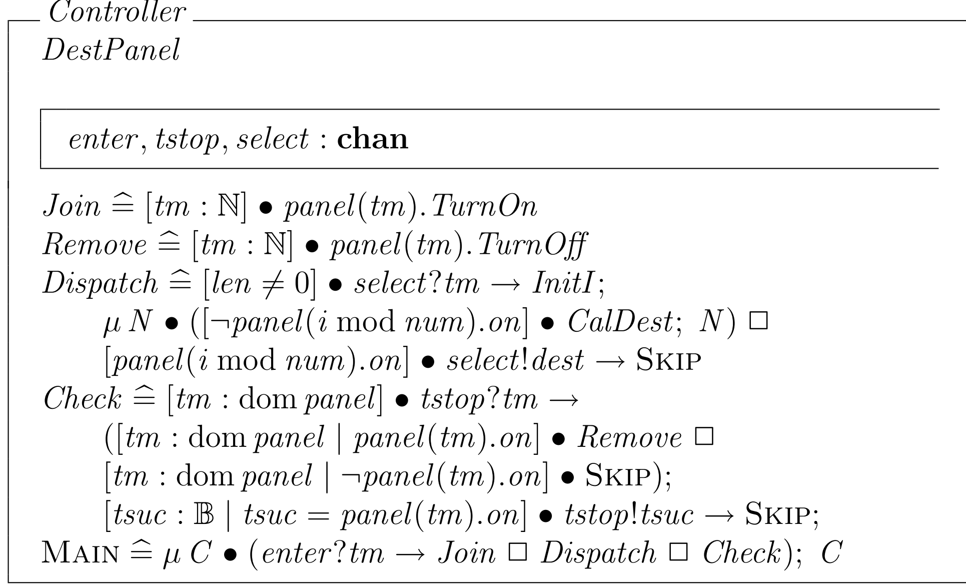
Car Panel: The car destination panel receives internal requests from passengers inside the railcar through the *int_request* channel and maintains a record of the requests to provide scheduling services for the car handler through channel *int_sched*. It also communicates with the car handler through channel *int_serv*, and *cstop*.

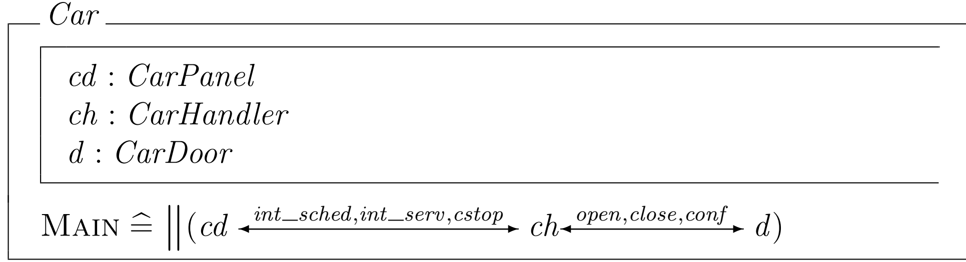
Due to the common behaviors of handling requests and calculating destinations between the car destination panel class and the controller class, a destination panel

class *DestPanel* is firstly defined to capture these operations. The car destination panel class *CarPanel* thus inherits from *DestPanel*.

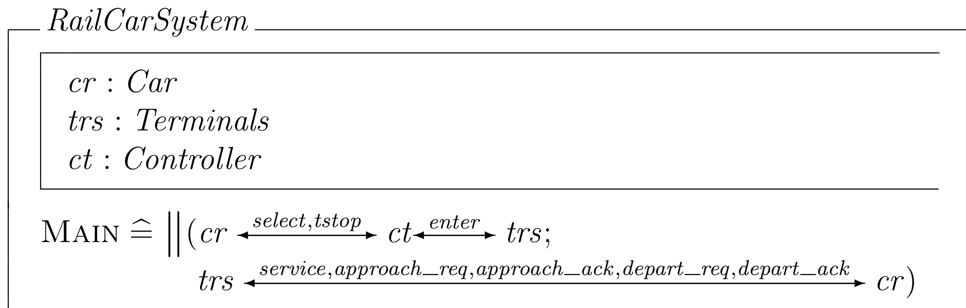


Central Controller: The responsibility of the central controller is to dispatch external requests to the railcar. It consists of a request queue with channels that connect the railcar and terminals. The *Controller* class is modeled as follows:





MRS Configuration: After specifying individual components, the next step is to compose them into a whole system. The overall system is a composition of all the communicating components.



5.2 Translation

In this section, we show how the given translation rules can be applied to map TCOZ specifications into timed automata.

For this system, first of all, each active class in the TCOZ model is projected to a TA model as a timed automaton template. Namely, a terminal template for the class *Terminal*; a car door, a car destination panel and a car handler template, respectively, for the classes *CarDoor*, *C_DestPanel*, and *CarHandler*; and a controller template for the class *Controller*. The terminal template has four instances which represent four different terminals according to the TCOZ specifications. The other templates will have only one instance.

We use the terminal class as an example to show the identification of the states, transitions, guards and synchronization mentioned above. Its processes mainly have an external choice pattern and a recursion pattern as shown in its process definition of the TCOZ model. According to the translation rules for the external choice pattern, four transition branches can be identified, respectively, representing the processes *RequestService*, *CarArrived*, *CarApproach*, and *CarDepart*. The first two processes *RequestService* and *CarArrived* match the *event prefix* pattern and are then projected as a returning switch. The latter two processes *CarApproach* and *CarDepart* are a little more complex. They both have an atomic operation in them, which is mapped into a state, i.e., $s2$ and $s3$ as shown in Figure 5.1. Synchronization and clock conditions on the transitions are constructed by trans-

forming the *CarApproach* and *CarDepart* process of *Terminal* class according to the translation rules for the WAIT and *event prefix* primitives.

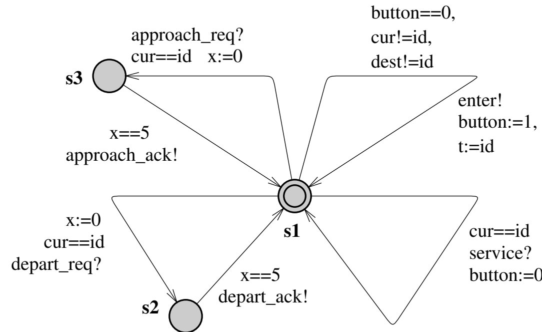


Figure 5.1: Terminal Panel

These main behaviors of this terminal automaton can be automatically generated by our translation tool. Then by adding the object reference information manually, such as the identification of each different terminal, the whole automaton can be visualized in UPPAAL.

We can get the other automata in the same way as follows,

carhandler : In this automaton, x is a clock variable which is reused several times by resetting its value to 0. *cur* records the position where the railcar currently is located. *tcur*/*ccur* determines whether the railcar should stop to serve the external/internal requests at a certain terminal or not when the railcar arrives at that terminal.

carpanel : The car panel automaton records internal requests in an array *clist*[4]. There are seven branches coming out from its initial state, they correspond to the

branches of the external choice processes in the TCOZ model of Class *CarPanel*; among which four of them corresponds to all the possible internal requests.

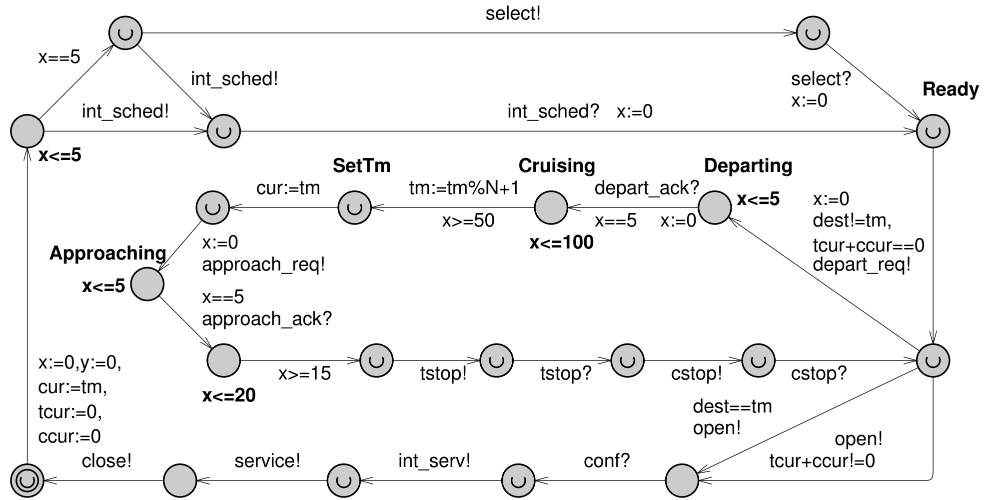


Figure 5.2: Car Handler

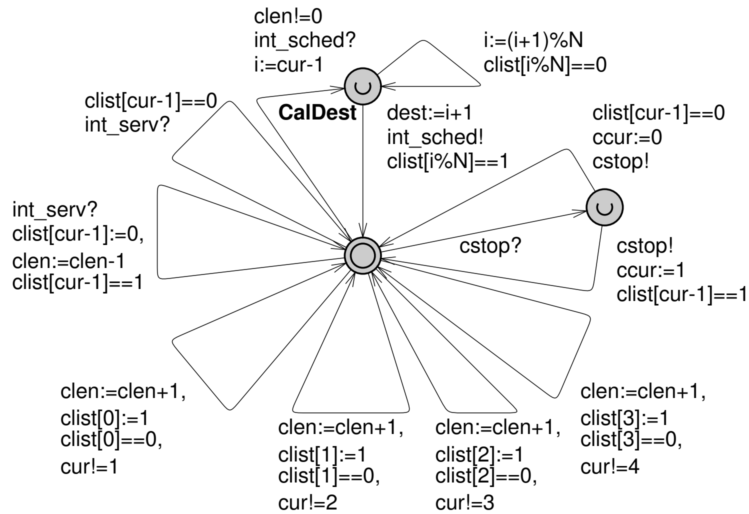


Figure 5.3: Car Panel

controller : The controller receives external requests from channel *enter* and records them in an array *tlist*[4]. After calculating the next destination, it dispatches one of the requests from the array to the railcar through channel *select*. When the

railcar has served that request, the controller will remove the request immediately.

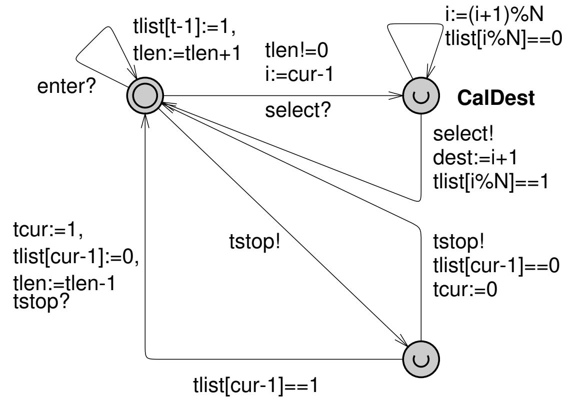


Figure 5.4: Controller

5.3 Model-checking MRS

Now we can use the simulator and verifier of UPPAAL to simulate the system as well as to model-check invariants and real-time properties. In UPPAAL correctness criteria can be specified as formulas of the timed temporal logic TCTL.

The key point of the MRS is to provide efficient services. These properties can be formally interpreted as follows.

- Efficient service properties - Whenever the car destination board receives a request to a terminal, say terminal 1, the railcar will eventually get to that terminal within 600 seconds. It can be translated into the TCTL as liveness properties:

```
cp1.clist[0]==1-->ch1.tm==1 and ch1. y<=600 //The railcar will
eventually reach terminal 1 within 600 seconds if a passenger
pressed terminal 1 button.
```

- Some other properties - Timing constraints, deadlock-freeness, and safety properties can also be checked, as shown in the following:

```
A[] cd1.open imply cd1.x<=10 and cd1.close imply cd1.x>=10 //Door
must be open for and only for 10 seconds. A[] not deadlock //The
system is deadlock-free. A[] ch1.cruising imply cd1.Close //
Whenever the railcar is running, the car door is always close.
```

UPPAAL verified that these properties actually hold for this given model.

Properties	**Time (N=3)**	**Time (N=4)**
cp1.clist[0]==1- ->ch1.tm==1 and ch1.y<=600	8.3s	250.0s
A[] cd1.Open imply cd1.x<=10	1.0s	11.2s
A[] not deadlock	2.6s	29.1s
A[] ch1.Cruising imply cd1.Close	1.1s	10.3s

Note that N is the number of terminals. The experiment was done under XP Windows system with RAM 2G, CPU Intel 3.0GHz.

Figure 5.5 and Figure 5.6 show the simulation and model-checking in UPPAAL.

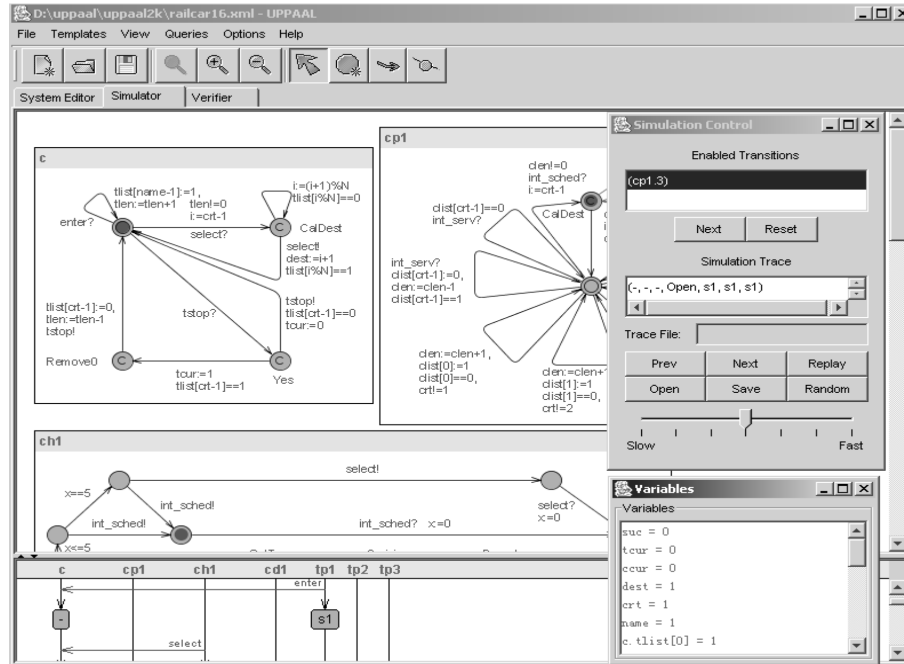


Figure 5.5: Simulation

5.4 Conclusion

In this chapter, we demonstrated the modelling and verification of a Multi-terminal Railcar system. Firstly, we found that TCOZ can be a good candidate for the high-level abstracted specifications of complex real-time systems. The class constructs in TCOZ are well suited for component declaration. The communication interfaces, i.e., channels, act as implicit connectors for modelling the communications between components. The network topology is used for defining the configuration of the system. All these features may provide a consistent and flexible way of specifying complex real-time models. Secondly, we found that after projecting the TCOZ

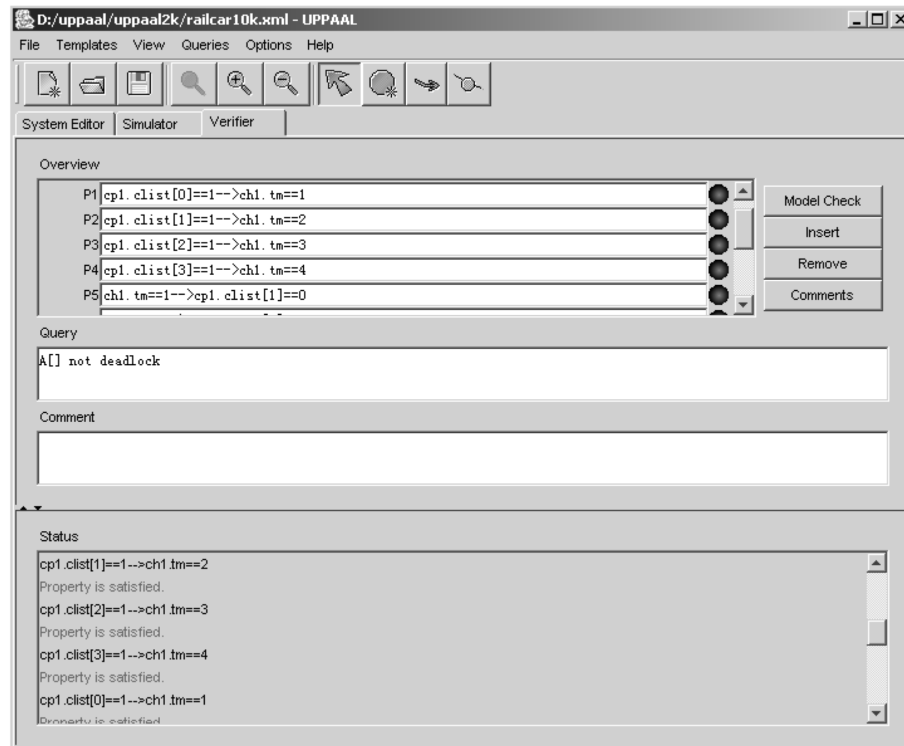


Figure 5.6: Verification

model to a TA model, the system behavior can be simulated and system properties, especially, real-time properties can be proved by reusing TA's tool support UPPAAL.

In summary, this chapter demonstrates a unified framework that supports one possible engineering process for modelling and verification of complex real-time systems. Namely, integrated formal modelling techniques (i.e., TCOZ) can be adopted for modelling complex systems; and low-level modelling techniques with direct tool support (TA) can be used for design and verification by projecting the high-level abstracted models (i.e., TCOZ models) to low-level models or designs (i.e., TA).

Chapter 6

Integrating Object-Z with Timed Automata

This chapter presents a new integrated formal method by combining Object-Z and Timed Automata.

6.1 Introduction

In our previous chapters 3, 4 and 5, we investigated the projection techniques from the TCOZ [53] (extension to Object-Z) to TA and discussed the notion of timed patterns. One interesting question arised from that part of work is that: can we integrate Object-Z and Timed Automata directly? In this way, not only the tool support of TA can be reused straightforward, but also the timed composable patterns now can be directly utilized for systematic TA designs. This motivate us to further our research on an integration approach. In the following chapters, rather than taking the transformation point of view, we propose a novel integrated formal language which combines Object-Z with TA. An effective combination of Object-Z and TA can not only help Object-Z with real-time modelling capability, but also help TA with enhanced structure and state modelling features. The result of such a combination can be a powerful unified method for designing complex computer systems. The challenge of achieving an effective combination of Object-Z and TA is to

- semantically and syntactically link the key language constructs so that the two notations can be used in a cohesive way;
- clearly separate system functionality aspects from time control behavior pat-

terns, so that separate tools can later be applied to check the related system properties;

- consistently unify the composition techniques from both Object-Z (class instantiation) and TA (automaton product) so that subsystem models can be easily and meaningfully composed;
- systematically develop the communication mechanisms so that various concurrent interactions between system components can be precisely captured.

In the remaining sections of this chapter we will demonstrate how Object-Z and TA can be effectively combined using motivating examples.

A Store

Consider a simple stock-control system for a store. The store stocks items which each have a fixed use-by date. An item can be added to the store's stock, but only if the use-by date of the added item is today's date or later. Any item can be sold by the store. At the beginning of each day, those items whose use-by date is less than the current date are removed (i.e., purged) from the store.

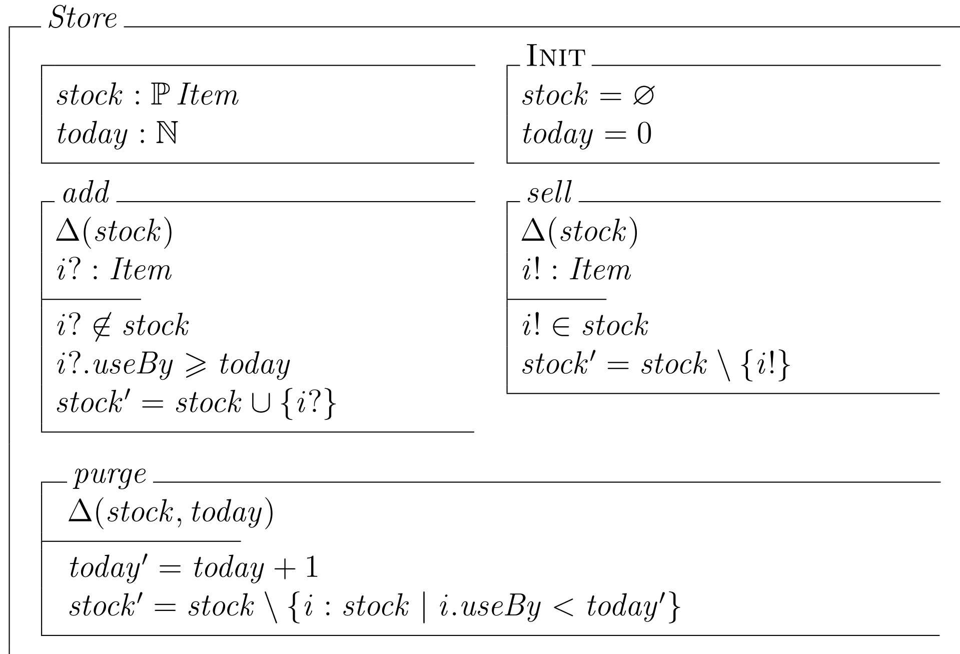
To specify this system in Object-Z, first we specify an item as an object of the class *Item*:

Item

$useBy : \mathbb{N}$

Effectively, at this level of abstraction the only important thing about an item is its (fixed) use-by date.

The stock control system is specified by the class *Store*:



The semantics of Object-Z can be seen as a state transition system. For example, given a particular *Store* object state

$$\sigma = \{(stock, \{item_a, item_b\}), (today, 20)\},$$

if operation *add* is then performed with a new input item $item_c$, the new object state would be

$$\sigma = \{(stock, \{item_a, item_b, item_c\}), (today, 20)\}.$$

Notice that although there is an attribute *today* in this class and this attribute is incremented whenever the *purge* operation takes place, no notion of the progressive

passing of time is captured by this specification. Conceptually, we think of the purge operation as taking place once a day, but this is not captured explicitly. Furthermore, in standard Object-Z the operations are assumed to be atomic, so there is no direct way of capturing the idea that an operation may take a specific time to complete.

6.2 Overview on Combining Object-Z and TA

In this section, the semantic and syntactic issues on integrating Object-Z and TA are discussed and a combined notation is proposed.

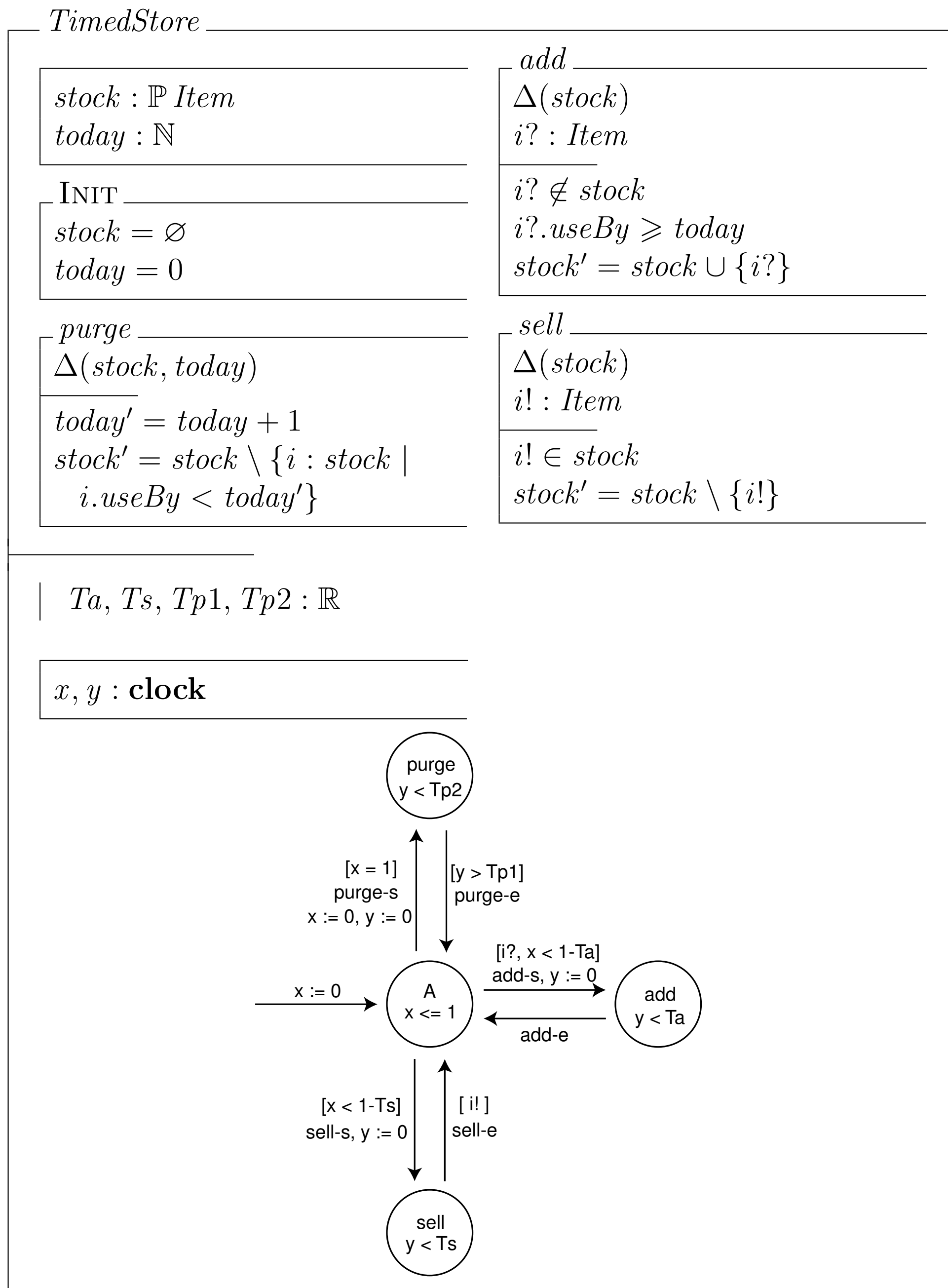
To illustrate how Object-Z and TA can be effectively integrated, consider the simple stock-control system we met in the last section. What we want is to integrate into this specification a notion of the sequential passing of time.

Suppose time is a positive real number measured in days starting at 0 (so, for example, 1.5 is halfway through the second day). The use-by date associated with an item is a positive integer denoting the day by which the item must be sold or else purged from the store (e.g., a use-by date of 3 means that if the item is not sold on or before day 3, at the start of day 4 it is purged).

We shall suppose it takes at most Ta time units to add an item to the stock, at most Ts time units to sell an item in stock, and more than $Tp1$ but less than $Tp2$ time units ($Tp1 < Tp2$) to purge the stock at the beginning of the day, where each of Ta, Ts and $Tp2$ is much less than 1. Furthermore, the addition of any item to

the stock or the selling of any item in stock must be started and completed within the same day. In our model, operations of the store will be disjoint, i.e., time-wise they do not overlap.

The store with this timing information incorporated is specified by adding a Timed Automaton to the class *Store* to get the class *TimedStore*:



Consider the *TimedStore* class in detail. The top part of the class box is the standard Object-Z specification we met in the last section and contains no timing information. The bottom part of the class box contains a declaration of the timing constants and the names of the clocks (in this case there are two clocks, x and y) as well as the associated automaton. The declaration x : **clock** means that x plays a dual role: it identifies (i.e., names) a clock and also records the time showing on the clock, i.e., it is a variable that takes positive real number values. In fact, as we shall see, the value of the clock x in this specification always lies between 0 and 1 inclusive and denotes the time that has passed in the current day. The clock y is used to ensure that the operations are completed within the specified time. It is assumed that both clocks progress at the same rate, i.e., the passage of time is universally uniform.

The locations of the automata represent the various situations in which the store can find itself. A location, together with the switches to and from that location, specifies the timing limits (if any) for the corresponding situation. For each of the three operations specified in the Object-Z part, there is a similarly-labelled location to capture the situation when the store is undergoing this operation; the store can undergo this operation only when in the corresponding location. The other location, A, represents the situation when the store is idle and no operation is being performed.

To illustrate the switches, consider those between locations A and *add*. The switch from A to *add* is labelled *add-s* (i.e., add start), while the switch from *add* to A

is labelled *add-e* (i.e., add end). The expression in square brackets, i.e., $[i?, x < 1 - Ta]$ in the case of the switch labelled *add-s*, captures the requirements that must be met if the switch is to take place, i.e., the input item $i?$ (as defined in the Object-Z operation *add*) must be supplied, and in addition the time as recorded by the clock x must be less than $1 - Ta$ (so that the operation, which can take up to Ta time units to occur, can be completed within the same day). In addition, the precondition of the Object-Z operation *add* must hold for the *add-s* switch to occur. As the precondition of an operation must always hold before the switch to the place labelled by that operation's name can occur, this precondition is always implicitly conjoined with any specific additional requirements within the square brackets. When the switch from A to *add* occurs, the clock y is reset to 0; annotating the location *add* with the condition $y \leqslant Ta$ ensures that this location is exited within time duration Ta, as required.

For the operation *sell*, the supply of the output item $i!$ is a requirement that must be met for the switch *sell-e* to occur after the completion of the operation. Compare this with the *add* operation where the input $i?$ was required for the switch starting the operation to occur.

Looking now at the switch *purge-s*, this switch can occur only when x is 1. Furthermore, it must occur at this time because of the time restriction placed on location A. This ensures that the purge operation occurs precisely once a day (starting at the end of each day and the beginning of the next). When the switch does occur, the clock x is reset to 0 (ensuring that x always lies between 0 and 1 and hence

denotes the time that has passed in the current day).

Location A is the automaton's initial location. The understanding is that the initial conditions as specified by the INIT schema must hold when the automaton is started in location A, and at the same time the clock x is set to 0 (the initial value of the clock y can be arbitrary and so is not specified).

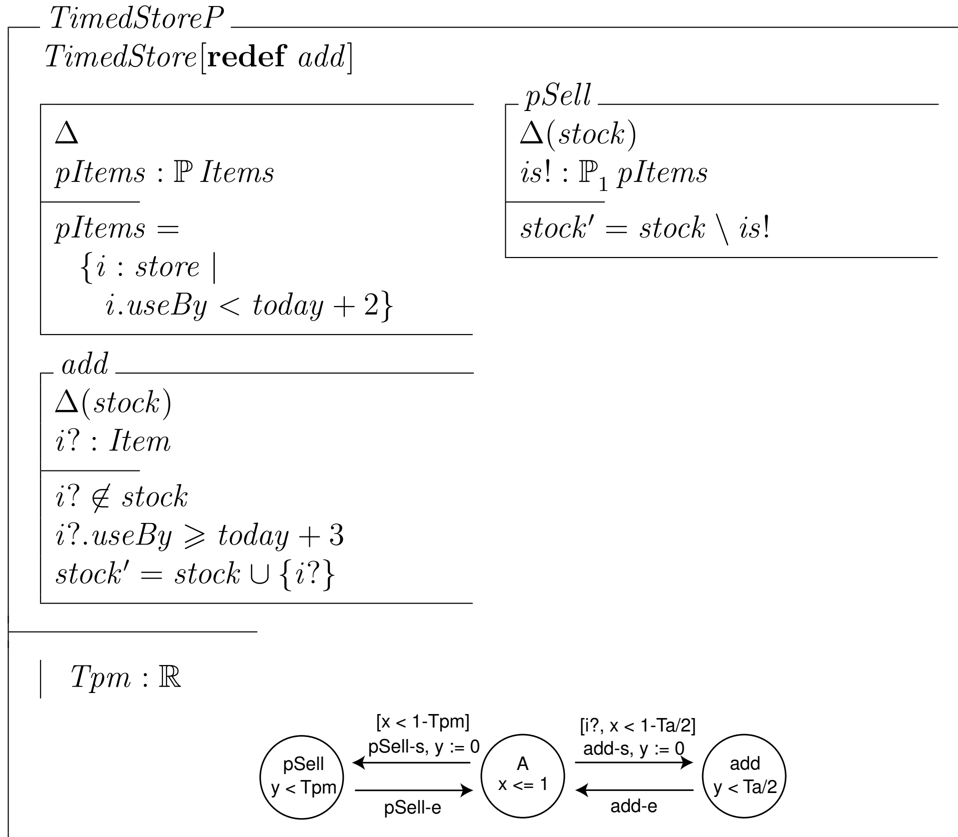
The fact that the 'start' switches associated with each operation emanate from location A, and the 'end' switches each return to A, ensures that the operations *add*, *sell* and *purge* do not overlap time-wise.

Note that the naming of switches can be systematic, e.g., a switch pointing to an operation state can be labeled with the operation name follow by 's' (for start). If a switch is pointing from an operation state to an idle (control) state, then it can be labeled with the operation name follow by 'e' (for end).

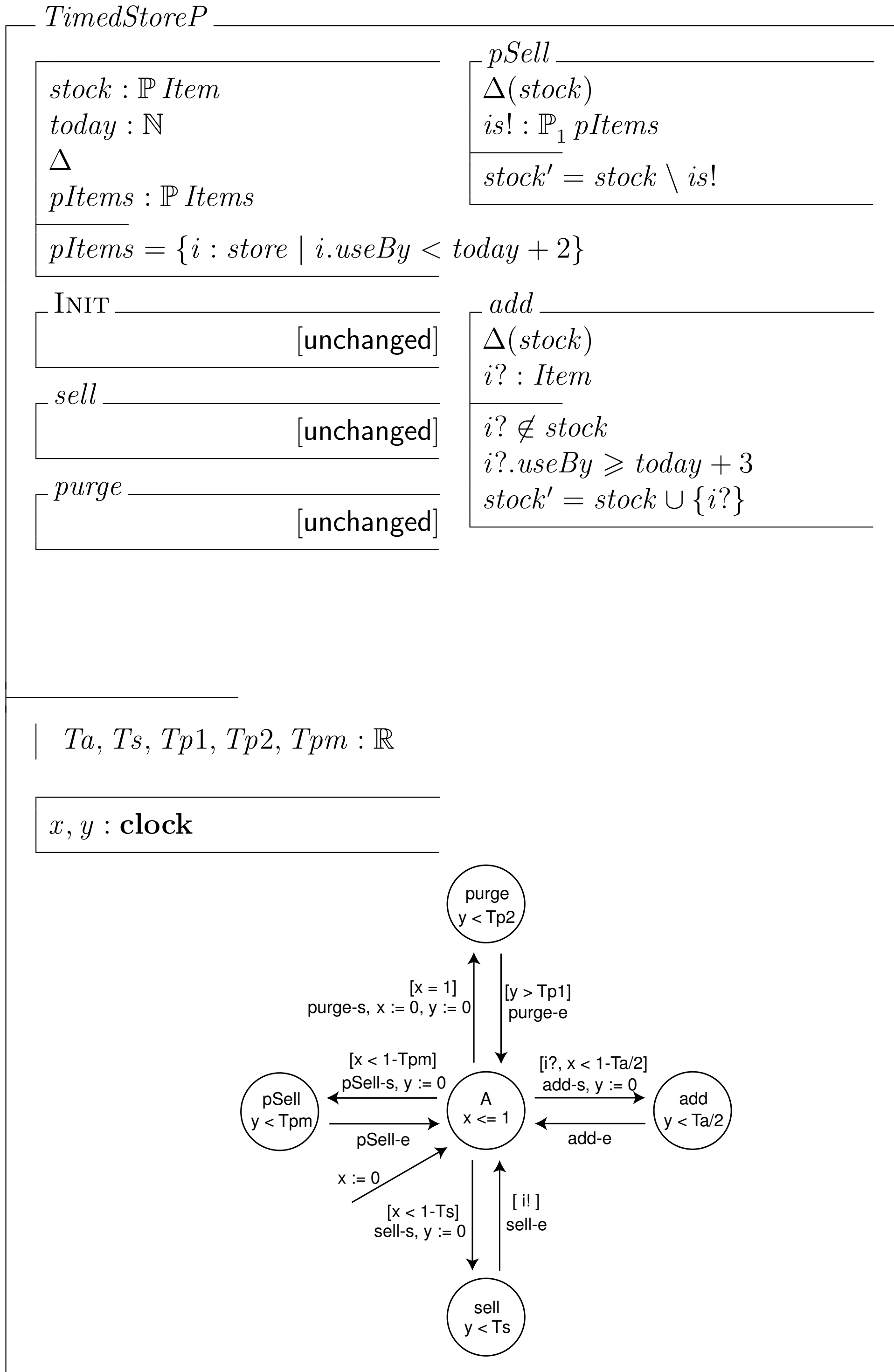
Inheritance

Inheritance is a mechanism for incremental specification and reuse, whereby new classes may be derived from an existing class. Object-Z inheritance has a similar style as the Z schema inclusion. We propose that the control behaviour (expressed by the TA) can also be inherited and extended in a simple way. Consider a system *TimedStoreP* which has the same *sell* and *purge* functionalities as *TimedStore*, except for the *add* operation: only items with an expire date at least 3 days ahead of the current day can be added into the store, and the *add* operation takes less

than a half of the *Ta* time units to finish. In addition, the system is able to identify the set *pItems* (promotion items) of items which have only two days left before their expiry. An extra operation *pSell* (promotion sell), which takes at most *Tpm* time units to execute, can sell a subset of these promotion items. The class *TimedStoreP* can be defined by inheriting the class *TimedStore*. For the behaviour (automaton) part, the *add* location refers to the redefined *add* operation, and its local invariant changes to $y < Ta/2$ and its enabling condition changes to $x < 1 - Ta/2$. And a new location *pSell* is introduced and connected (by new switches) with the control (idle) location *A* from *TimedStore*. The other locations and their connections remain unchanged as follows:



If we expand the inheritance, then *TimedStoreP* becomes:



Note that *pItems* is modelled as a secondary attribute whose value is subject to change with each operation (implicitly it is included in every operation's Δ list).

For multiple inheritance cases, the rules are that all similarly named locations (and switches) are merged, with all corresponding invariants and conditions conjoined.

6.3 Design Decisions

In the previous section, we had an overview on how Object-Z and Timed Automata have been integrated. The approach taken in the OZTA notation is to identify Object-Z operations as states in Timed Automata. Accordingly, pre/post-conditions of an Object-Z operation are identified as transition conditions. Questions may arise on our integration approach, i.e., why Object-Z operation schemas are identified as states in the associated timed automaton in an OZTA class? Can we semantically link Object-Z operations with TA transitions in the associated timed automaton? In the following, the design decisions related to this new formalism will be discussed and some examples will be used to show the reason we choose to link Object-Z operations with TA states instead of TA transitions.

Consider the previous timed store example. Suppose we choose to identify the Object-Z operations as transitions in the associated timed automaton. Let us focus on the *add* operation part only, this branch can be designed as shown in Figure 6.1.

The interpretation of this model is now different: there will be no state operations any more; all the states are control states; instead, there will be two kinds of transitions: operation transitions which represent the Object-Z operation schemas

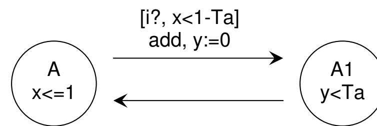


Figure 6.1: Model 1

and control transitions which coordinate the control flow together with control states. On an operation transition, both the pre-condition and post-condition of the corresponding operation schema should hold, which indicates that the update of data variables is done on the corresponding operation transition. For example, the data variable *stock* will be updated when the *add* operation transition occurs.

This approach fits nicely with the Object-Z interpretation of operations being atomic, but is not well suited to multi-thread and real-time modelling. Restricting operations to be atomic events collapses the spatial and temporal aspects of operations. Everything happens at a single point and instantaneously. This would result in a need for a number of control states to artificially reconstruct the temporal aspects of the operations in the associated TA. Sometimes the design of these control states can be unnecessarily tricky and may take extra effort to create or understand. For example, in the timed store system, suppose that during the *add* operation, a store inspector may interrupt the process to check the quality of the item to be added for no more than T_c time units. For this requirement, the model can be easily extended in our adopted approach (where operation schemas are identified as states), as shown in Figure 6.2, in which the transition labelled 'add-s' implicitly indicates that the precondition of the *add* operation schema holds and

the transition labelled with 'add-e' implicitly indicates the post-condition of the *add* operation schema holds. This design allows the *add* operation to take time and the data variable *stock* to be updated after the operation .

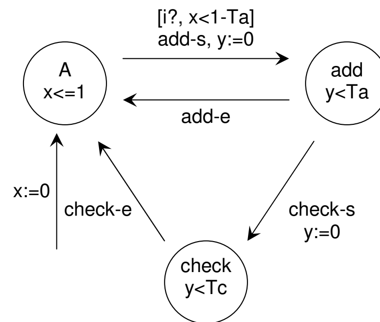


Figure 6.2: Model 2

However, in a model where operation schemas are identified as transitions, a problem happens as shown in Figure 6.3.

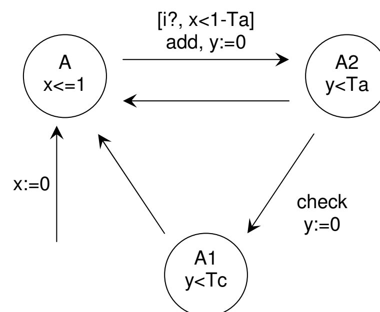


Figure 6.3: Model 3

As the *add* transition happens, both the pre-condition and post-condition of *add* operation would hold, namely, the value of *stock* would be updated simultaneously. As a result, the *check* event can never actually interrupt the *add* operation because it is atomic. The same thing may happen to any processes involved with timed

interrupt or time-out behaviors. To address this situation, we have to add some control state(s) before the *add* operation transition to make sure that the data variable *stock* is only updated when the add operation has not been interrupted as shown in Figure 6.4,

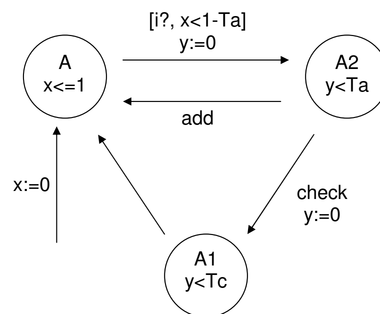


Figure 6.4: Model 4

where the *add* operation is now identified as the outgoing transition of control state $A2$, meanwhile, the pre-condition of *add* also has to hold on the incoming transition of $A2$. Compared to this model, the model in Figure 6.2 is more apt for modelling timing properties in a natural way and carries no redundant pre/post-condition from the Object-Z operation schemas.

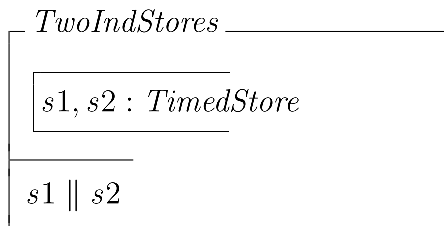
Another problem of linking operation schemas with transitions is that, for a complex real-time system with communications, identifying synchronization event names with operation names would create unnecessary tensions between the data and process views of objects, and considerably reduce the potential for reuse of operation definitions since both of them are treated as events on transitions in Timed Automata.

6.4 Composition and Communication

In this section, various composition and communication aspects of the combined language are discussed, and synchronized communication links are systematically introduced.

Independent stores

Consider now a system consisting of two stores operating independently. This system is specified by the class *TwoIndStores*:



The timed automaton of this class is simply the product [4] of the automata for the two stores $s1$ and $s2$. The timed automaton for $s1$ is just the automaton of the *TimedStore* class, but with the label of each location, the label of each switch, and the names of the clocks distinguished by an '$s1$.' prefix, as illustrated in Figure 6.5. Notice that the input/output variables are not prefixed.

The timed automaton for $s2$ is labeled similarly. The $s1 \parallel s2$ notation in the class *TwoIndStores* denotes the product of the associated automata. The implication here is that the two stores are not only completely independent, but operations in different stores can be executed concurrently. Indeed, when an object of the

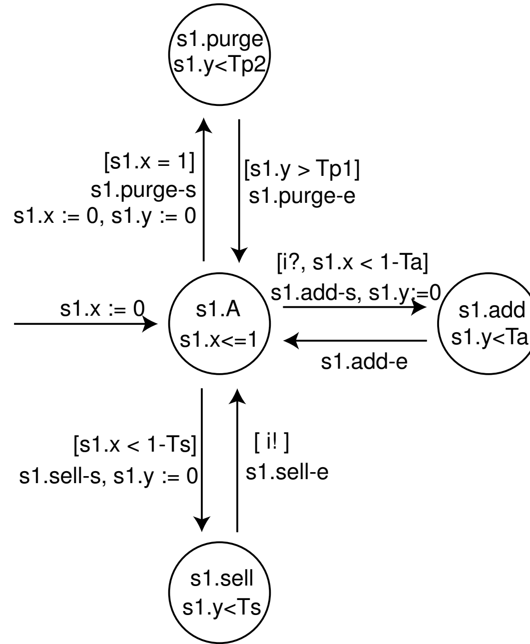
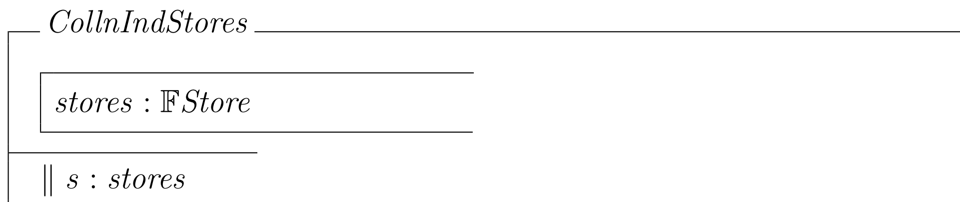


Figure 6.5: The Automaton $s1$

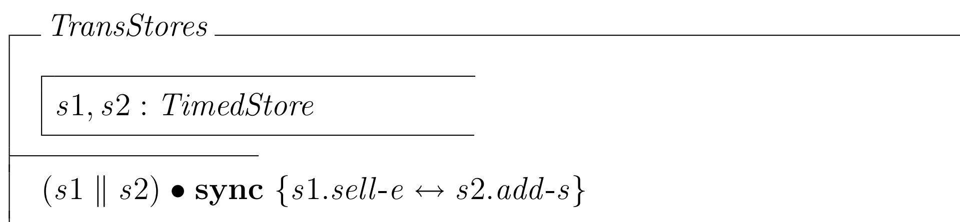
class *TwoIndStores* is instantiated, the two store objects start at the same time in their A position with $s1.x$ and $s2.x$ set to 0 synchronously. As time passes at the same rate for all clocks, both stores will always synchronise on the start of their respective *purge* operations, namely, at the start of the next day, but apart from that they run completely independently.

The two-stores example can be generalised to a collection of independent stores, as specified by the class *CollnIndStores*. In this class the expression $\parallel s : stores$ denotes the timed automata product $(s_1 \parallel s_2 \parallel \cdots)$ where the set *stores* is $\{s_1, s_2, \cdots\}$.



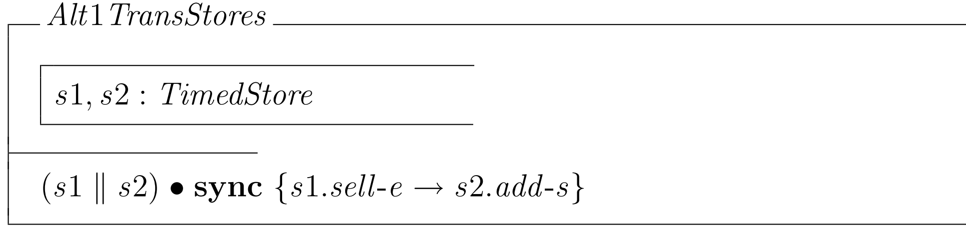
Transferring between stores

Consider now a system consisting of two stores, where each item sold by the first store is added (i.e. transferred) to the second. Effectively, the first store sells items only to the second store. A specification of this system is given by the class *TransStores*:



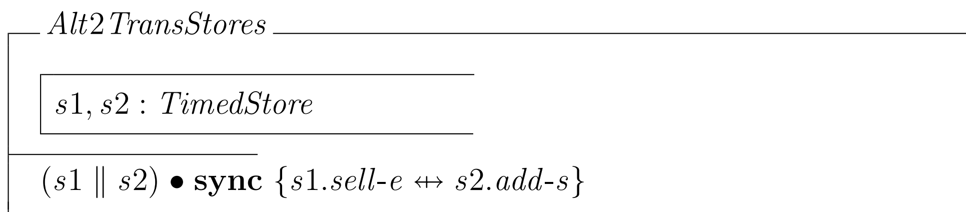
The **sync** clause indicates that the two switches labeled $s1.sell\text{-}e$ and $s2.add\text{-}s$ are to be treated as if these labels were identical, i.e. the automata must synchronize on these switches. As part of this synchronization, as the output $i!$ and the input $i?$ have the same base-name they are identified and hidden (just as is the case for the Object-Z parallel operator, i.e. they specify internal communication rather than communication with the environment). Apart from this synchronization, the product of the two timed automata effectively ensures that the two automata operate independently and concurrently.

Now consider a system like *TransStores* where again each item sold by the first store is added (i.e. transferred) to the second. However, an item from the environment may also be added to the second store, i.e. not all items added to the second store are necessarily transferring from the first. This system is specified in the class *Alt*1 *TransStores*:



The implication here is that whenever the switch $s1.\mathit{sell}\text{-}e$ is taken then there must be synchronization with the switch $s2.\mathit{add}\text{-}s$. However, the switch $s2.\mathit{add}\text{-}s$ can occur independent of (i.e. without synchronizing with) the switch $s1.\mathit{sell}\text{-}e$. With this notation, notice that the synchronization $\textbf{sync}\ \{s1.\mathit{sell}\text{-}e \leftrightarrow s2.\mathit{add}\text{-}s\}$ in the *TransStores* class could have been alternatively (but less elegantly) expressed as $\textbf{sync}\ \{s1.\mathit{sell}\text{-}e \rightarrow s2.\mathit{add}\text{-}s, s2.\mathit{add}\text{-}s \rightarrow s1.\mathit{sell}\text{-}e\}$.

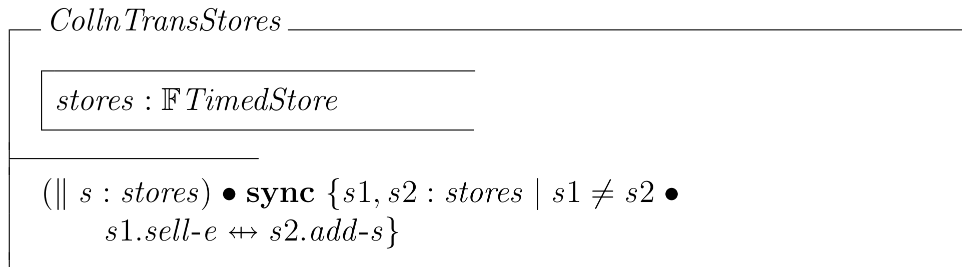
Now consider the situation as before where an item sold by the first store can be transferred to the second, but in addition not only can an item from the environment be directly added to the second store, (i.e. not all items added to the second store are necessarily transferring from the first) but also an item sold by the first store can be passed to the environment (i.e. not all items sold by the first store are necessarily transferred to the second). This system is specified in the class *Alt2 TransStores*:



The implication here is that when any of the switches $s1.\mathit{sell}\text{-}e$ or $s2.\mathit{add}\text{-}s$ is taken there may or may not (the choice is non-deterministic) be synchronization with the

switch $s2.add\text{-}s$ or $s1.sell\text{-}e$ respectively.

The examples involving two stores given so far in this section can be generalised to a collection of stores. Consider a system consisting of a collection of stores where an item from the environment can be added to any store, an item sold by any store can be passed back to the environment, and given any two stores in the collection, an item sold by the first store can be added (transferred) to the second. Such a system is specified in the class *CollnTransStores*:



CollnTransStores

$stores : \mathbb{F}\, TimedStore$

$(\| \; s : stores) \bullet \mathbf{sync}\ \{s1, s2 : stores \mid s1 \neq s2 \bullet$
$\quad s1.sell\text{-}e \leftrightarrow s2.add\text{-}s\}$

More on synchronization

To further illustrate synchronization in TA, consider the three timed automata U, V and W illustrated in Figure 6.6.

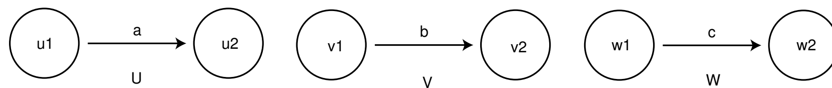


Figure 6.6: Timed Automata U, V and W

The timed automaton $(U \parallel V \parallel W) \bullet \mathbf{sync}\ \{a \leftrightarrow b\}$ is behaviorally equivalent to the product $U1 \parallel V1 \parallel W1$ of the timed automata $U1$, $V1$ and $W1$ illustrated in Figure 6.7. In this case the switches labeled a and b have been re-named to a

common label d. As these labels are the same, the product automaton will synchronize on these switches. Consequently, the switch from location $u1$ to location $u2$ in $U1$ is always synchronized with the switch from location $v1$ to location $v2$ in $V1$, and conversely.

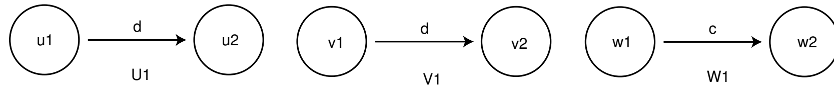


Figure 6.7: Timed Automata $U1$, $V1$ and $W1$

The timed automaton

$$(U \parallel V \parallel W) \bullet \textbf{sync}\ \{a \leftrightarrow b, a \leftrightarrow c, b \leftrightarrow c\}$$

is behaviorally equivalent to the product $U2 \parallel V2 \parallel W2$ of the 3 automata $U2$, $V2$ and $W2$ illustrated in Figure 6.8. In this case the switch from location $u1$ to $u2$ in $U2$ must synchronize with either the switch from location $v1$ to $v2$ in $V2$ or from $w1$ to $w2$ in $W2$; the switch from location $v1$ to $v2$ in $V2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $w1$ to $w2$ in $W2$; and the switch from location $w1$ to $w2$ in $W2$ must synchronize with either the switch from location $u1$ to $u2$ in $U2$ or from $v1$ to $v2$ in $V2$.

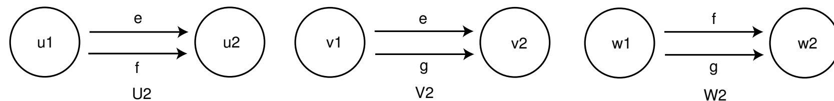


Figure 6.8: Timed Automata $U2$, $V2$ and $W2$

Compare this to the automaton

$$(U \parallel V \parallel W) \bullet \textbf{sync}\ \{a \leftrightarrow b \leftrightarrow c\}.$$

This automaton is behaviorally equivalent to the product $U3 \parallel V3 \parallel W3$ of the three automata $U3$, $V3$ and $W3$ illustrated in Figure 6.9. In this case the three switches from location $u1$ to $u2$ in $U3$, from location $v1$ to $v2$ in $V3$ and from location $w1$ to $w2$ in $W3$ must synchronize.

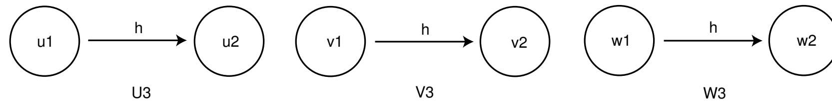


Figure 6.9: Timed Automata $U3$, $V3$ and $W3$

The timed automaton $(U \parallel V) \bullet \textbf{sync}\ \{a \rightarrow b\}$ is behaviorally equivalent to the product $U4 \parallel V4$ of the timed automata $U4$ and $V4$ illustrated in Figure 6.10. In $V4$ a switch labeled a is added to duplicate the switch labeled b. As the switch in automaton $U4$ is also labeled a, this ensures that the product automaton will synchronize on these two switches. Consequently, the switch from location $u1$ to location $u2$ in $U4$ is always synchronized with a switch from location $v1$ to location $v2$ in $V4$, but not conversely. The translation from location $v1$ to location $v2$ can use the switch labeled b in which case no synchronization takes place.

The timed automaton $(U \parallel V) \bullet \textbf{sync}\ \{a \nleftrightarrow b\}$ is behaviorally equivalent to the product $U5 \parallel V5$ of the timed automata $U5$ and $V5$ illustrated in Figure 6.11. In this case the switches labeled a and b are both duplicated and a common name, d, is assigned to these new switches. This ensures that the product automaton will synchronize on these two switches. Consequently, a translation from location $u1$ to location $u2$ in $U5$ can synchronize with a translation from location $v1$ to location $v2$ in $V5$ if the switch labeled d is used. However, a translation from location $u1$ to

Figure 6.10: Timed Automata $U4$ and $V4$

Figure 6.11: Timed Automata $U5$ and $V5$

location $u2$ in $U5$ could use switch a, or a translation from location $v1$ to location $v2$ in $V5$ could use switch b; in either case no synchronization takes place.

6.5 Operation Semantics

we present a formal description of the operational behavior of this integrated language. The fundamental semantic links between Object-Z and TA are:

- Object-Z operations are identified with states in Timed Automata.
- Pre/Post-condition of an Object-Z operation are identified with TA transition conditions.

The key novel idea of integrating the Object-Z semantics and TA semantics is to embed object state updates (of Object-Z) into the action transition semantics of TA. To facilitate the description of dynamic behaviors of a system, we introduce a set of locations A, called control locations, to coordinate the location switches from

one Object-Z operation to another. Each location of a timed automaton specified in a class must be either a control location or an Object-Z operation location. A class has an Object-Z part *OZDefinition* which obeys the conventional definition [66] and a TA part *TADefinition* which is a a timed automaton. We write *OZop* to denote the set of Object-Z operations defined in the class. The original Object-Z operation operators: parallel composition, nondeterministic choice, and sequential composition are replaced by *TADefinition* defined as follows.

$\mathbb{S}_{OZTA}$ is a tuple $(S, S_0, \Sigma, X, I, E)$, where

- S is a union of A and Op, in which A is a finite set of control (idle) states and Op is a finite set of operation states corresponding to the Object-Z operations;
- S_0, a subset of S, is a set of initial locations;
- Σ is a set of labels;
- X is a finite set of clocks;
- I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$; and
- E, a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, r, \varphi \rangle$ represents a transition from location s to location s' on input symbol a. The set r gives the clocks to be reset with this transition, and φ is a clock constraint over X that specifies when the switch is enabled.

In the following, we present a timed transition system $\mathbb{S}_{OZTA}$ to represent operational semantic models for this integrated language. Before we start to define the

operational semantics, we need some definitions for the validity of Object-Z and TA expressions.

The fact that a state guard G is valid under the semantic function $\sigma : \mathit{Var} \nrightarrow \mathit{Value}$ is denoted by the following notation:

$$\sigma \models G$$

The fact that an operation Op is valid under the semantic functions σ_1, σ_2 is denoted by

$$\sigma_1, \sigma_2 \models \mathit{Op}$$

For example, in the context of the Store system,

$$\{(\mathit{stock}, \{\mathit{item}_a, \mathit{item}_b\}), (\mathit{today}, 20)\}, \{(\mathit{stock}, \{\mathit{item}_a, \mathit{item}_b, \mathit{item}_c\}), (\mathit{today}, 20)\}$$

$$\models \mathit{add}[i? \mapsto \mathit{item}_c]$$

To keep track of the changes of clock values, we use functions known as clock assignments mapping X to the non-negative reals R_+. Let u, v denote such functions, and use $u \models \varphi$ to mean that the clock values denoted by u satisfy the guard φ. For $d \in R_+$, let $u + d$ denote the clock assignment that maps all $x \in X$ to $u(x) + d$, and for $r \subseteq X$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and agree with u for the other clocks in $X \backslash r$.

To facilitate the description of operational semantics, let

$$OP : Location \twoheadrightarrow OZop$$

denote the association between TA locations to Object-Z operations.

The operational semantics of this integrated language is an extension of TA transition semantics coupled with object states. The timed state transition system $\mathbb{S}_{OZTA}$ consists of states which are tuples $\langle l, u, \sigma, \sigma_1 \rangle$ and state transitions are defined by the rules:

$$R_1 : \frac{l \xrightarrow{a,\varphi,r}_1 l' \ \sigma, \sigma_1 \models OP(l) \ \sigma_1, \sigma_2 \models OP(l') \ u \models \varphi \ u' = [r \mapsto 0]u \ u' \models I(l') \ l, l' \in Op}{\langle l, u, \sigma, \sigma_1 \rangle \xrightarrow{a}_1 \langle l', u', \sigma_1, \sigma_2 \rangle}$$

R_1 is an action transition from one operation (location) state l to another operation state l' where the post object state of l must be the same as the pre object state of l', the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_2 : \frac{\sigma_1, \sigma_2 \models OP(l) \ u \models I(l) \ u + d \models I(l') \ d \in R_+ \ l \in Op}{\langle l, u, \sigma_1, \sigma_2 \rangle \xrightarrow{d}_1 \langle l, u + d, \sigma_1, \sigma_2 \rangle}$$

R_2 is a delay transition in a certain operation state where only time is progressed.

$$R_3 : \frac{l \xrightarrow{a,\varphi,r}_1 l' \ u \models \varphi \ u' = [r \mapsto 0]u \ u' \models I(l') \ l \in A \ l' \in A}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{a}_1 \langle l', u', \sigma, \sigma \rangle}$$

R_3 is an transition from one control (location) state l to another control state l' where object states remain the same.

$$R_4 : \frac{u \models I(l) \;\; u + d \models I(l') \;\; d \in R_+ \;\; l \in A}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{d}_1 \langle l, u + d, \sigma, \sigma \rangle}$$

R_4 is a delay transition in a control state where time is progressed.

$$R_5 : \frac{l \xrightarrow{a,\varphi,r}_1 l' \;\; \sigma_1, \sigma_2 \models OP(l) \;\; u \models \varphi \;\; u' = [r \mapsto 0]u \;\; u' \models I(l') \;\; l \in Op \;\; l' \in A}{\langle l, u, \sigma_1, \sigma_2 \rangle \xrightarrow{a}_1 \langle l', u', \sigma_2, \sigma_2 \rangle}$$

R_5 is an action transition from one operation (location) state l to a control state l' where the post object state of l much be the same as the object state of l', the timing constraints on the transition must be satisfied and the location invariants of l and l' must be true.

$$R_6 : \frac{l \xrightarrow{a,\varphi,r}_1 l' \;\; \sigma, \sigma_1 \models OP(l') \;\; u \models \varphi \;\; u' = [r \mapsto 0]u \;\; u' \models I(l') \;\; l \in A \;\; l' \in Op}{\langle l, u, \sigma, \sigma \rangle \xrightarrow{a}_1 \langle l', u', \sigma, \sigma_1 \rangle}$$

R_6 is the inverse of R_5.

These rules define six types of transitions in $\mathbb{S}_{OZTA}$. These rules are applied to a single timed transition system. A complex system can be described as a product of interacting timed transition systems. The communications between two transition systems are obtained by synchronizing the transition with identical labels.

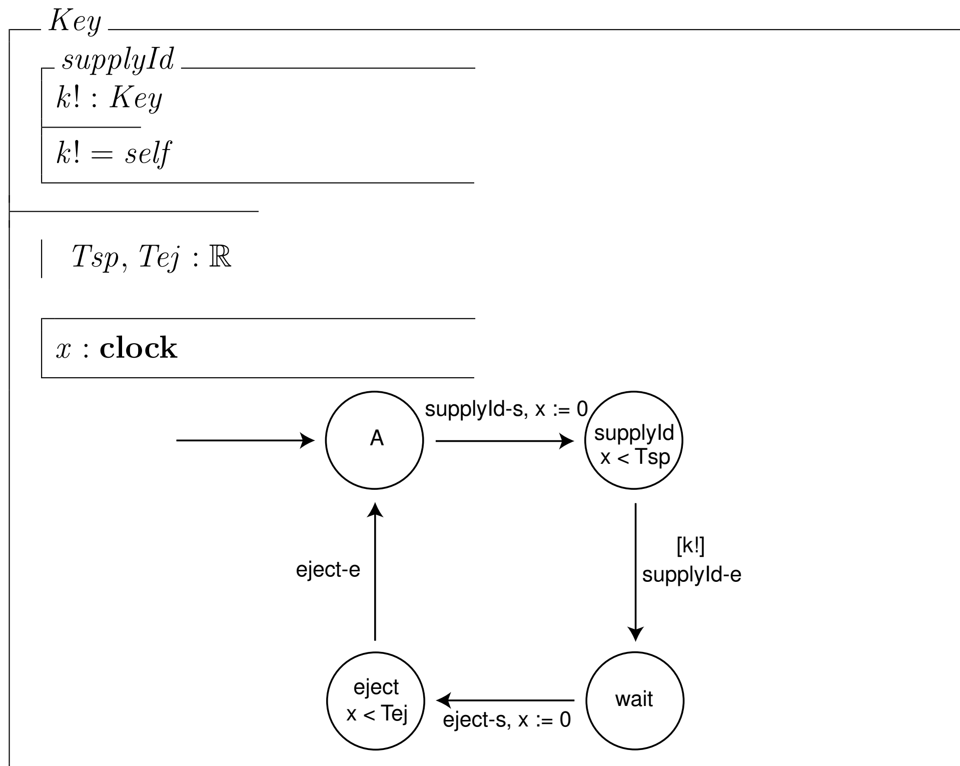
6.6 An Example: Electronic Key System

As an illustration of how Object-Z and TA can be successfully integrated in practice, we present here an electronic key system as an example.

A room can be accessed through a sliding door. To open the sliding door, an electronic key is inserted into the door's electronic lock. The identity of the key (as encoded as part of the key) is passed to the lock that then checks to see if the key has permission to access the room. When access permission has been checked the key is ejected from the lock. If the key has access permission, the door is opened (or remains open); otherwise the door is closed (or remains closed).

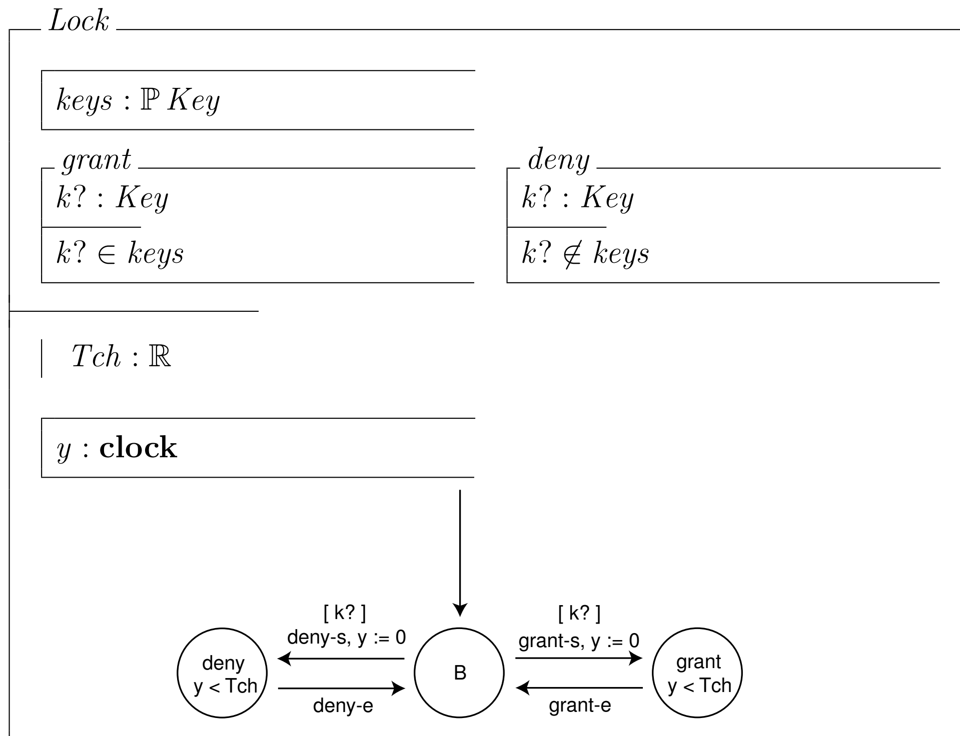
We shall suppose that it takes less than Tsp time units, from the time the key is inserted in the door's lock, for the key to supply its identity to the lock, less than Tch time units for the lock to check if the key has permission to access the room, less than Tej time units for the key to be ejected from the lock, and less than Top time units for the door to satisfy an 'open' request. Also, if the door has been open Tto time units since the last 'open' request, a time-out occurs and the door is closed. It takes less than Tcl time units for the door to satisfy a 'close' request.

A key is specified by the class Key:



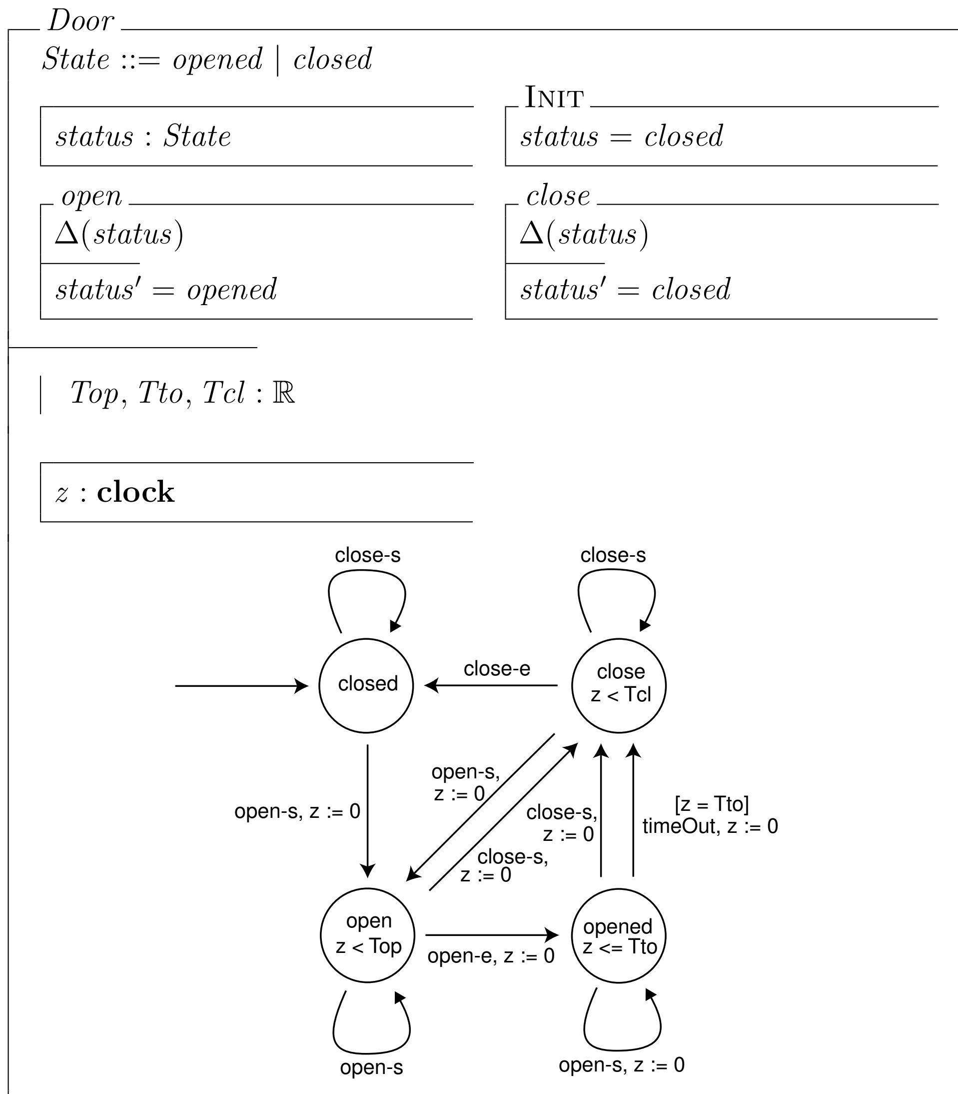
The only operation in the Object-Z section of this class is *supplyId* specifying the situation in which a key supplies its identity (to the lock). When considering time aspects, however, other situations arise. A key will be in location *wait* after it has supplied its identity and is waiting to see whether or not access is granted. A key will be in location *eject* when it is being ejected from the lock once access permission has been decided.

The lock is specified by the class *Lock*:



The attribute *keys* in this class denotes the set of keys that have permission to access the room. The operations *grant* and *deny* capture whether or not any supplied key is in this set, and hence whether or not access to the room is granted or denied.

The door is specified by the class *Door*:

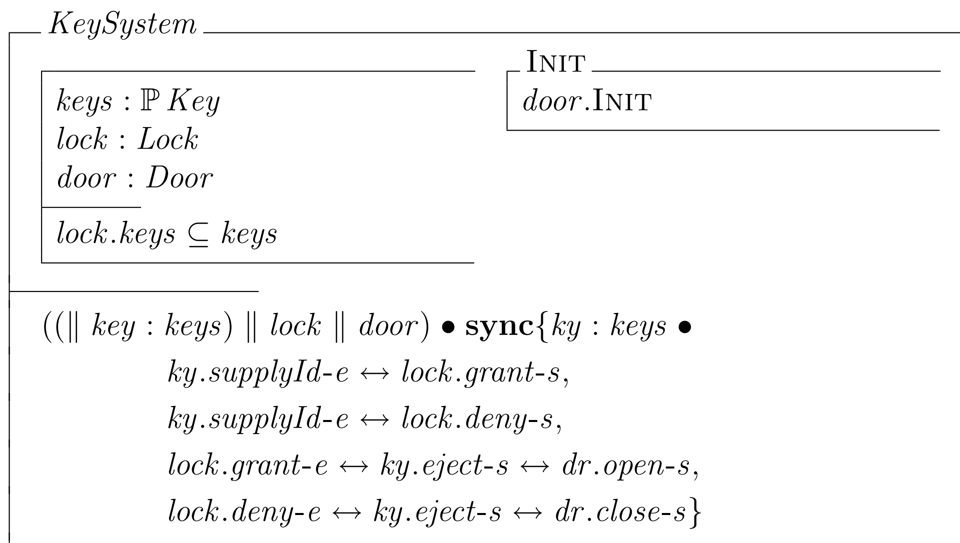


A door can be in any of four situations: closed (location *closed*), opening (location *open* where the operation *open* occurs), opened (location *opened*), and closing (location *close* where the operation *close* occurs). In each situation, the door can receive an instruction to open or close the door. In all cases, when an instruction to open the door is received, the switch *open-s* is taken, while if an instruction to close the door is received, the switch *close-s* is taken.

In locations *closed* or *close*, if the instruction to open is received, the operation

open is invoked, while if the instruction to close is received effectively the door continues as if nothing had happened. In location *open*, if the instruction to open is received effectively the door continues as if nothing had happened, while if the instruction to close is received the operation *close* is invoked. In location *opened*, if the instruction to open is received the door remains open, but the timing is reset to 0, while if the instruction to close is received the operation *close* is invoked.

The complete electronic key system can now be specified by the class *KeySystem*. In this class, the attribute *keys* denotes the set of all keys in the system; the set of keys that have permission to access the room will be a subset of *keys*. The synchronization conditions ensure that the key identity output by a key is passed to the lock and used to determine whether or not that key has permission to access the room, and that once the access permission has been decided the key is ejected and the door requested to open or close, depending on whether access was granted or denied.



KeySystem

$keys : \mathbb{P}\, Key$
$lock : Lock$
$door : Door$
$lock.keys \subseteq keys$

INIT
$door.\mathrm{INIT}$

$((\| \; key : keys) \parallel lock \parallel door) \bullet \mathbf{sync}\{ky : keys \bullet$
$ky.supplyId\text{-}e \leftrightarrow lock.grant\text{-}s,$
$ky.supplyId\text{-}e \leftrightarrow lock.deny\text{-}s,$
$lock.grant\text{-}e \leftrightarrow ky.eject\text{-}s \leftrightarrow dr.open\text{-}s,$
$lock.deny\text{-}e \leftrightarrow ky.eject\text{-}s \leftrightarrow dr.close\text{-}s\}$

6.7 Conclusion

Software system specification is an important activity in software engineering. The specification of complex real-time systems requires powerful mechanisms for modeling state, concurrency and real-time behavior as well as tool support for verifying the established system model.

In this chapter, we proposed a new integrated specification language, OZTA. Building on the strength of Object-Z and Timed Automata, which are state-of-the-art modelling techniques, respectively prevailing in Europe and North America, OZTA provides

- not only powerful mechanisms to capture various aspects of a complex real-time system, namely, system functionalities can be best captured in terms of operations and constraints — the ideal application for Object-Z, system control behaviors can be best captured in terms of visual flows between system functionalities— the ideal application for Timed Automata;
- but also easy access to TA's tool support, for the verification.

Chapter 7

OZTA Semantics

7.1 Introduction

OZTA is a novel integrated formal language which builds on the strengths of Object-Z and Timed Automata in order to provide a single notation for modelling the static, dynamic and timing aspects of complex systems as well as for verifying system properties by reusing Timed Automata's tool support.

In the previous chapter, we have introduced the basics of OZTA notation and made an preliminary exploration of its operational semantics. In this chapter, we will further enhance the OZTA notation by extending its automaton part with time pattern structures. And based the enhanced OZTA syntax, we will formalize the semantics of OZTA in the Unifying Theories of Programming(UTP) [40] to provide the foundation for language understanding, reasoning and especially, tool construction. Note that the previous operational semantics we provided is not totally compatible with the denotation semantics as it doesn't support pattern concepts.

7.2 The Syntax of OZTA

OZTA specifications are combination of Object-Z schemas with timed automata. TA has powerful mechanisms for designing real-time models using multiple clocks and has well developed automatic tool support. However, if TA is used to capture real-time requirements, then one often needs to manually cast common timing behaviors, such as *deadline*, *timeout* etc., into a set of clock variables with carefully

calculated clock constraints, which is a process that is very much towards design rather than specification. In chapter 3, we studied TA patterns and found that a set of common timed patterns, such as *deadline*, *timeout*, *waituntil*, can be used to facilitate TA design in a systematic way. In this chapter, before presenting the semantics of OZTA, we will firstly give a full version of the OZTA syntax, in which the OZTA notation is enhanced by implementing its automaton part with timed pattern structures. The specification of the syntax of OZTA enhanced with the notion of timed patterns can be presented as follows:

$$
\begin{array}{l}
\mathit{Specification} ::= \mathit{CDecl};\ ...;\ \mathit{CDecl} \\
\mathit{CDecl} ::= \upharpoonright \mathit{Visiblist};\ \mathit{InheritC};\ \mathit{StateSch};\ \mathit{INIT};\ \mathit{StaOp};\ [\mathit{TADecl}] \\
\mathit{Visiblist} ::= \mathit{VisibAttr};\ \mathit{VisibOp} \\
\mathit{InheritC} ::= \mathit{InheritCName} \\
\mathit{StateSch} ::= \mathit{CVarDecl} \\
\mathit{CVarDecl} ::= v : T \\
\mathit{StaOp} ::= \Delta(\mathit{AttrName} \mid \mathit{ActName}), \mathit{CVarDecl} \bullet \mathit{Pred}(u, v') \\
\mathit{TADecl} ::= \mathit{ClockDecl};\ \mathit{TA} \\
\mathit{ClockDecl} ::= x : \mathit{Clock} \\
\mathit{TA} ::= \mathit{State} \\
\mid \mathit{State} \bullet \mathit{Invar}(x, n) \\
\mid [\mathit{Event}][\mathit{Reset}(x)][\mathit{Guard}(x, n)] \bullet \mathit{TA} \\
\mid \mathit{Wait}\ (x, n) \\
\mid \mathit{TA} \bullet \mathit{Deadline}\ (x, n) \\
\mid \mathit{TA} \bullet \mathit{WaitUntil}\ (x, n) \\
\mid \mathit{TA} \bullet \mathit{Timeout}(x, n) \bullet \mathit{TA} \\
\mid \mathit{TA};\ \mathit{TA} \\
\mid \mathit{TA} \Box \mathit{TA} \\
\mid \mathit{TA} \sqcap \mathit{TA} \\
\mid \mu X \bullet \mathit{TA}(X) \\
\mid \mathit{TA}_1 \parallel \mathit{TA}_2 \bullet S \\
\mathit{State} ::= \mathit{StaOp}(\mathit{operation\ state}) \mid \mathit{StaCtr}(\mathit{control\ state}) \mid \mathit{StaU} \\
\mathit{Event} ::= \mathit{Event} \mid \mathit{Event}! \mid \mathit{Event}? \\
\mathit{Reset} ::= (_ := _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \\
S ::= \{_ \nleftrightarrow _\}\langle\!\langle \mathit{Event} \times \mathit{Event} \rangle\!\rangle \mid \{_ \leftrightarrow _\}\langle\!\langle \mathit{Event} \times \mathit{Event} \rangle\!\rangle \mid \\
\{_ \rightarrow _\}\langle\!\langle \mathit{Event} \times \mathit{Event} \rangle\!\rangle \\
\mathit{Guard} ::= (_ <= _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \mid (_ >= _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \\
\mid (_ < _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \mid (_ > _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \\
\mid (_ \wedge _)\langle\!\langle \Phi \times \Phi \rangle\!\rangle \mid \mathit{true} \\
\mathit{Invar} ::= (_ <= _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \mid (_ < _)\langle\!\langle \mathit{Clock} \times N \rangle\!\rangle \mid \mathit{true}
\end{array}
$$

in which,

- *StaCtr* represents a control (idle) state which coordinates the state switches from one Object-Z operation to another and *StaOp* is an operation state corresponding to the Object-Z operation. Each state of a timed automaton specified in an OZTA class must be either a control state or an operation state.

- Object-Z operations are identified with states in TA part of an OZTA class. *INIT* operation scheme defines the initial values of the data variables declared in a state schema, It implicitly corresponds to the first state defined in the expression of the TA part of an OZTA model, which is either a control state or an operation state.

- *Event*, *Reset*(x), *Guard*(x, n) are transition labels for an automaton *TA*, which respectively specifies synchronization (*Event*! is an output event, *Event*? is an input event), clock reset and clock constraint; *State* $\bullet$ *Invar*(x, n) specifies a state with a local invariant. Meanwhile, pre/post-condition of an Object-Z operation are implicitly identified as TA transition conditions.

- The rest of the expressions are the Timed Automata patterns which now can be directly utilized to construct Timed Automata.

Among the specification, the argument x represents a certain clock, and n is a natural number. The key novel idea of integrating the Object-Z semantics and TA semantics is to embed object state updates (of Object-Z) into the action transition semantics of TA.

7.2.1 An example : Shunting Game

The rules of the shunting game is that: given a board, a starting position and four marked positions, a move consists of the black piece (the shunter) moving one position either vertically or horizontally provided either

- the position moved to is empty, or
- the position moved to is occupied by a white piece (a box) but the position beyond the box is empty, in which case the box is pushed into empty position

The shunter can not push two or more boxes at a time. At each stage a score is kept of the number of moves made so far. The game ends when the boxes occupy the four marked positions.

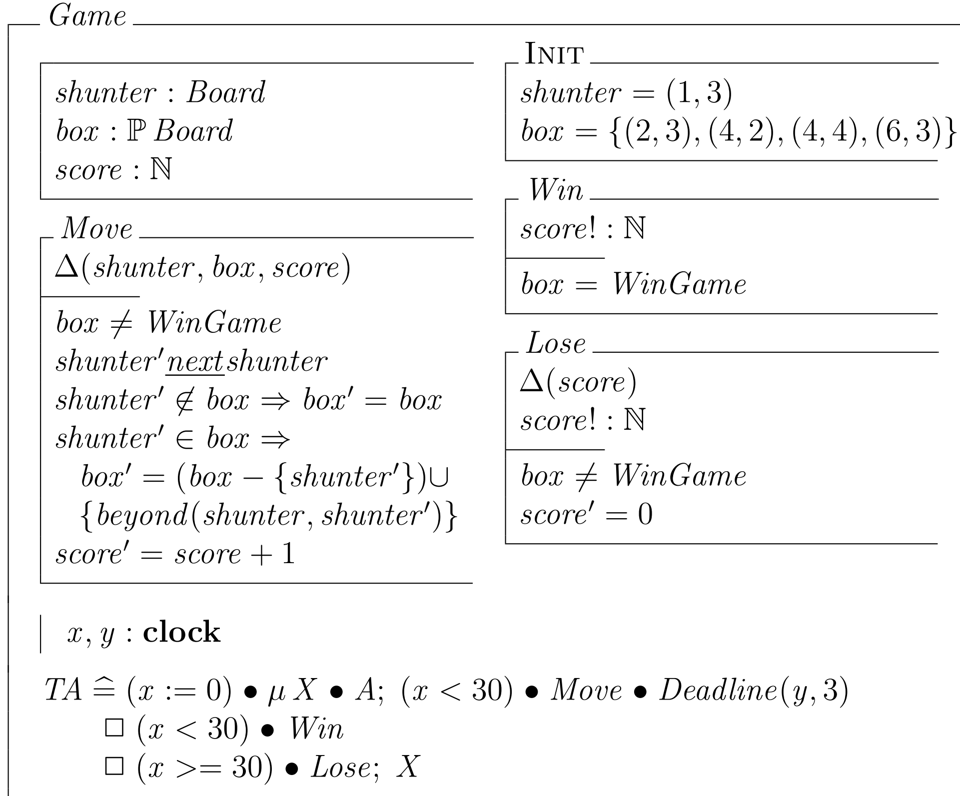
The OZTA model of this shunting game is given as follow,

$Board == (1..7 \times 3..4) \cup (3..4 \times 1..6)$

$$
\begin{array}{|l}
next : Board \times Board \\
\hline
\forall (i,j),(k,l) : Board \bullet \\
(i,j)\underline{next}(k,l) \Leftrightarrow \\
i = k \land (j = l+1 \lor j = l-1) \lor \\
j = l \land (i = k+1 \lor i = k-1)
\end{array}
$$

$$
\begin{array}{|l}
WinGame : \mathbb{P}\, Board \\
\hline
over = \{(3,3),(4,3),(3,4),(4,4)\}
\end{array}
$$

$$
\begin{array}{|l}
beyond : \mathbb{P}\, Board \times \mathbb{P}\, Board \twoheadrightarrow \mathbb{N} \times \mathbb{N} \\
\hline
\mathrm{dom}\, beyond = \{b, w : Board \mid b \underline{next} w\} \\
\forall\, b, w : \mathrm{dom}\, beyond \bullet \\
beyond(b,w) = 2w - b
\end{array}
$$



The Init schema defines the starting positions of the shunter and the four boxes. *WinGame* represents the set of the marked positions.

The shunter takes at most 3 time units to push an item to its next position. The shunter loses the game if he can not finish his task in 30 time units. According to the *composition*, *external choice*, *deadline*, *recursion* timed patterns, the corresponding graphical TA specification can be derived as in Figure 7.1.

7.3 The Semantics of OZTA

Before building the semantics model for OZTA, we need to choose an appropriate model of time. There are two typical time models: a discrete model and a

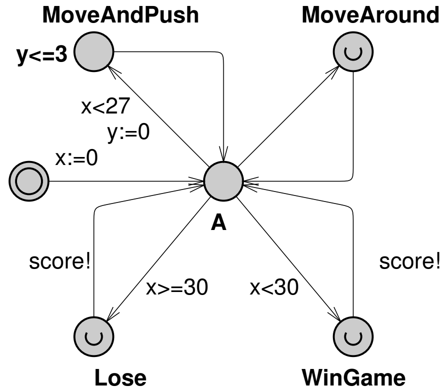


Figure 7.1: The Shunting Game

continuous model. The current semantics model for OZTA [22] is a primitive operational semantics based on continuous time without pattern features. To make our model with the extension of timed patterns more apt for exploration of algebraic refinement laws, we choose the discrete model. The discrete time model has also been adopted by the Sherif and He's work [64] on the semantics for time Circus [75, 76, 77] and Qin, Dong and Chin's work [59] on the semantics for TCOZ.

7.3.1 The Automata Model

The following meta variables are introduced in the alphabet of the observations of the OZTA automata behavior, some of which are similar to those in the previous UTP semantic frameworks [59]. The key difference is that we now take into consideration clock variable updates.

- ok, ok': *Boolean*. These two variables are introduced to denote the observations of automaton initiation and termination. ok records the observation

that the automaton has started. When ok is false, the automaton has not started, so no observation can be made. ok' records the observation that the automaton has successfully terminated. The automaton is divergent when ok' is false.

- $wait$, $wait'$: $Boolean$. Because of the requirement for synchronization, an active process will usually engage in alternate periods of internal activity (computation) and periods of quiescence or stability, while it is waiting for a reaction or an acknowledgement from its environment. We therefore introduce a variable $wait'$, which is true just when a process is waiting in such quiescent periods. Its main purpose is to distinguish intermediate observations from the observations made on termination. $wait$ is used in the initial observation, which is true when the process starts in an intermediate state.

- $state$, $state'$: $Var \rightarrow Value$. In order to record the state of data variables(class attributes and local variables) that occur in an automaton, these two variables are introduced to map each variable to a value in the corresponding observations.

- tr, tr': $\mathrm{seq}(\mathrm{seq}\, Event \times \mathbb{P} Event)$. The two variables are introduced to record the sequence of observations on the interactions between an automaton and its environment. tr records the observations that occurred before the automa-

ton starts and tr' records the final observation. Each element of the sequence represents an observation over one time unit. Each observation element is composed of a pair, where the first element of the tuple is the sequence of events that occurred during the time unit, and the second is the associated set of refusals at the end of the same time unit. The set *Event* includes all possible communicating events.

- *trace*: $\mathrm{seq}\,Event$. This variable is used to record a sequence of events that take place so far since the last observation. It can be derived from tr, tr' as the following:

  $flat(tr) ^\frown trace = flat(tr')$ where $^\frown$ is a concatenation operator flat is defined as:

  $flat : \mathrm{seq}(\mathrm{seq}\,Event \times \mathbb{P}Event) \rightarrow \mathrm{seq}\,Event$

  $flat(\langle\rangle) \mathrel{\widehat{=}} \langle\rangle \qquad flat(\langle(es, ref)\rangle ^\frown tr) \mathrel{\widehat{=}} es ^\frown flat(tr)$

  An auxiliary function $cs(trace)$ is adopted to extract the subsequences of communication events from the sequence *trace*. The function cs is defined as:

  $cs((\langle\rangle)) \mathrel{\widehat{=}} (\langle\rangle)$

  $cs((\langle e\rangle) ^\frown tail) \mathrel{\widehat{=}} \langle e\rangle ^\frown cs(tail), e \in Event.$

- $cval$, $cval'$: $Clock \rightarrow N \cup \{NULL\}$. *Clock* denotes all clock variables; N is the set of natural numbers; $NULL$ denotes the situation that the clock has not been enabled yet.

Some other definitions are given to facilitate the description of OZTA semantics.

- The predicate $no_interact(trace)$ denotes that there are no communication events recorded in $trace$.

  $no_interact(s) \mathrel{\widehat{=}} cs(s) = \langle\rangle$

- The operator $\circ$ is the composition of two sequentially made observations. For two observation predicates $P(v, v')$ and $Q(v, v')$, where v, v' represents respectively the initial and final versions of all observation variables, the composition of them is:

  $P(v, v') \circ Q(v, v') \mathrel{\widehat{=}} \exists\, v_0 \bullet P(v, v_0) \wedge Q(v_0, v')$

- A binary relation $\preceq$ is the ordinary subsequence relation between sequences of the same type.

7.3.2 The Semantics of OZTA Automata with Patterns

In this section, the observation model for OZTA automata is developed. We use TA to stand for the semantics predicate of an automaton TA instead of the term $[\![TA]\!]$ in UTP. Before we go into to the details of the semantics for each OZTA automata expression, three healthiness conditions **R1** – **R3** [59, 40] must be satisfied by the semantics predicate for any automaton,

R1 $TA = TA \wedge (tr \overset{t}{\preceq} tr')$

$tr \overset{t}{\preceq} tr'$ states that, given two timed traces, tr and tr', tr' is an expansion of tr.

R2 $TA(tr, tr') = TA(<>, tr' - tr)$

It states that the initial value of tr may be replaced by '<>' and the events in which the process TA itself engages remains the same.

R3 $TA = \amalg \lhd wait \rhd TA$

Where the predicate $\amalg = \neg ok \wedge (tr \overset{t}{\preceq} tr') \vee ok' \wedge (tr' = tr) \wedge ... \wedge (wait' = wait)$. It means that if the process is asked to start in a waiting state of its predecessor, it leaves the state unchanged.

State and Control Operation

- Operation State

  $StaOp \mathrel{\widehat{=}} \Delta(b), a : T \bullet Pred(u, v') \mathrel{\widehat{=}} ok' \wedge \neg\ wait' \wedge no_interact(trace) \wedge (\forall x : \mathrm{dom}\, cval \mid cval(x) \neq NULL \bullet cval' = cval \oplus \{x \mapsto (cval(x) + \#tr' - \#tr)\}) \wedge ((\exists val_1 \bullet state' = state \oplus \{a \mapsto val_1\}) \circ (\exists val \bullet state' = state \oplus \{a \mapsto val\} \wedge Pred(state(u), state'(v'))))$

  In an operation state, time may progress, and no event or state will be updated. $NULL$ means the clock has no value, it has not been initialized yet.

- Control state

  $StaCtr \mathrel{\widehat{=}} ok' \wedge \neg\ wait' \wedge no_interact(trace) \wedge (\forall x : \mathrm{dom}\, cval \mid cval(x) \neq NULL \bullet cval' = cval \oplus \{x \mapsto (cval(x) + \#tr' - \#tr)\})$

  In a control state, time may progress, and no event or state update.

- Urgent state

  $StaU \mathrel{\widehat=} (StatOP \vee StaCtr) \wedge \#tr' = \#tr$

  The semantics of an urgent state is that the automaton will pass the control from the urgent state to a next state without delay.

- Init State

  $StaI \mathrel{\widehat=} ok' \wedge \neg wait' \wedge tr = \langle\rangle \wedge no_interact(trace) \wedge (\forall x : \mathrm{dom}\, cval \bullet cval(x) = NULL) \vee ok' \wedge \neg wait' \wedge tr \neq \langle\rangle \wedge no_interact(trace) \wedge (\forall x : \mathrm{dom}\, cval \mid cval(x) \neq NULL \bullet cval' = cval \oplus \{x \mapsto (cval(x) + \#tr' - \#tr)\})$

  The sequence of observations of an OZTA model starts from an initial state. The value of each clock variable is initially set to $NULL$. The initial state may also act as a normal control state after the automaton starts working.

Local Invariant

In verification tools e.g. UPPAAL, local invariants are often restricted to constraints that are downwards closed, i.e., in the form: $x < n$ or $x \leq n$ where n is natural number.

$State \bullet Invar(x, n) \mathrel{\widehat=} x \in \mathrm{dom}\, cval \wedge (State \wedge (cval(x) + \#tr' - \#tr) < n \wedge (\forall c : \mathrm{dom}\, cval \mid cval(c) \neq NULL \bullet cval' = cval \oplus \{c \mapsto (cval(c) + \#tr' - \#tr)\}) \vee Stop)$

Clock Reset

$Reset(x) \mathrel{\widehat=} ok' \land \neg wait' \land \#tr' = \#tr \land state' = state \land (\exists x : Clock \mid x \in \operatorname{dom} cval \bullet cval' = cval \oplus \{x \mapsto 0\})$

Consecutive clock reset operations are combined into one atomic reset operation.

$Reset(x) \bullet TA \mathrel{\widehat=} Reset(x);\; TA$

Event

$Event \mathrel{\widehat=} ok' \land \neg wait' \land trace = \langle Event \rangle \land state' = state \land \#tr' = \#tr$

$Event \bullet TA \mathrel{\widehat=} Event;\; TA$

Clock Constraint

An automaton can be guarded by a clock constraint. The clock-guarded automaton $Guard(x, n) \bullet TA$ behaves as TA if the condition $Guard(x, n)$ is initially satisfied.

$Guard(x, n) \bullet TA \mathrel{\widehat=} (\exists x : Clock \bullet x \in \operatorname{dom} cval) \land (Guard(x, n) \land TA \lor \neg\; Guard(x, n) \land Stop)$

It enjoys the following properties:

- G1. $false \bullet TA = Stop$
- G2. $true \bullet TA = TA$
- G3. $Guard(x, n) \bullet Stop = Stop$

- G4. $Guard_1(x_1, n_1) \bullet (Guard_2(x_2, n_2) \bullet TA) = (Guard_1(x_1, n_1) \wedge Guard_2(x_2, n_2)) \bullet TA$

- G5. $Guard(x, n) \bullet (TA_1;\ TA_2) = (Guard(x, n) \bullet TA_1);\ TA_2$

These algebraic laws can be derived from our semantic definition according to propositional and predicate calculus. For example, G1-G3 can be proved as follows,

Proof :

- G1. $false \bullet TA = false \wedge TA \vee \neg\ false \wedge Stop = false \vee true \wedge Stop = Stop$

- G2. $true \bullet TA = true \wedge TA \vee \neg\ true \wedge Stop = TA \vee false = TA$

- G3. $Guard(x, n) \bullet Stop = (\exists\, x : Clock \bullet x \in \mathrm{dom}\, cval) \wedge (Guard(x, n) \wedge Stop \vee \neg\ Guard(x, n) \wedge Stop) = (\exists\, x : Clock \bullet x \in \mathrm{dom}\, cval) \wedge Stop = Stop$

Wait

The Wait construct specifies an automaton which idles for n time units and then terminates.

$Wait(x, n) \mathrel{\widehat=} ok' \wedge \neg\ wait' \wedge \#tr' - \#tr = n \wedge (\forall\, i : \#tr' < i < \#tr \bullet no_interact(\pi_1(tr'(i))))$

It is subject to the following laws.

- $\textsc{Wait}\, n_1;\ \textsc{Wait}\, n_2 = \textsc{Wait}(n_1 + n_2)$

- $\textsc{Stop} \bullet Timeout(x, n) \bullet TA = \textsc{Wait}\, n;\ TA$

Deadline

The Deadline construct $TA \bullet \textsc{Deadline}(x, n)$ imposes a timing constraint on the automaton TA, which requires that TA should terminate no later than n time units.

$TA \bullet \textsc{Deadline}(x, n) \mathrel{\widehat=} (ok \land (x \in \operatorname{dom} cval \land cval' = cval \oplus \{x \mapsto 0\})) \circ TA \bullet Invar(x, n)$

It can also be described in this way,

$TA \bullet \textsc{Deadline}(x, n) \mathrel{\widehat=} ok \land x \in \operatorname{dom} cval \land Reset(x) \bullet TA \bullet Invar(x, n)$

WaitUntil

The WaitUntil construct $TA \bullet \textsc{WaitUntil}(x, n)$ constrains automation TA to finish in no less than n time units.

$TA \bullet \textsc{WaitUntil}(x, n) \mathrel{\widehat=} TA \land (\#tr' - \#tr \geq n) \lor ((\exists\, tr_o \bullet tr \preceq tr_o \preceq tr' \land \#tr_o - \#tr < n) \land (ok \land x \in \operatorname{dom} cval \land cval' = cval \oplus \{x \mapsto 0\})) \circ (TA[tr_o/tr', true/ok', false/wait']) \circ (Wait(x, n - (\#tr_o - \#tr))[tr_o/tr]))$

Timeout

The timeout construct $TA_1 \bullet Timeout(x, n) \bullet TA_2$ specifies that if no transition has been triggered for n time units in the timed automaton TA_1, then TA_1 will timeout and the control will be passed to TA_2.

$TA_1 \bullet Timeout(x, n) \bullet TA_2 \mathrel{\widehat=} (ok \wedge (x \in \operatorname{dom} cval \wedge cval' = cval \oplus \{x \mapsto 0\})) \circ ((TA_1 \wedge no_interact(trace) \wedge \#tr' - \#tr \leq n) \vee (\exists k : \#tr < k \leq tr + n, \exists tr_o \bullet \pi_1(tr'(k)) \neq \langle\ \rangle \wedge tr \preceq tr_o \wedge \#tr_o - \#tr = k \wedge (\forall i : \#tr < i < \#tr + k \bullet no_interact(\pi_1(tr'(i))) \wedge tr_o(i) = tr'(i)) \wedge TA_1[tr_o/tr]) \vee (\exists tr_o \bullet tr \preceq tr_o \wedge \#tr_o - \#tr = n \wedge (\forall i : \#tr < i < \#tr + n \bullet no_interact(\pi_1(tr'(i))) \wedge tr_o(i) = tr'(i)) \wedge TA_2[tr_o/tr]))$

Recursion

We define the semantics of recursion same as [64, 59]. We say that a process A is as good as process B if it will meet all the operations and satisfy all the specifications satisfied by B. This relation is denoted by $A \sqsupseteq B$. A process A is equal to a process B if

$A = B \mathrel{\widehat=} A \sqsupseteq B \wedge B \sqsupseteq A.$

Notice that the set of observations in our model from a complete lattice with respect to the relation $\sqsupseteq$, having *Chaos* (which is a process with its predicate as *true*) as its bottom element, $\sqcap$ as the greatest lower bound. So we can define the semantics of recursion as the weakest fixed point [40].

$\mu X \bullet TA(X) \mathrel{\widehat=} \sqcap\{X \mid X \sqsupseteq TA(X)\}$

X is the fixed point.

Parallel Composition

The parallel composition of two automata represents all the possible behaviors of both automata which are synchronized on a specific set of events and on the time when the events occur.

In addition to the handshake synchronization, OZTA also supports other two synchronization mechanisms, namely, partial synchronization and sometime synchronization.

Given a parallel composition $TA_1 \mid[E]\mid TA_2 \bullet S$, where E denotes the set of events on which TA_1 and TA_2 will synchronize, and S contains elements of the form $a \rightarrow b$, $a \leftrightarrow b$ ($E \cap event(S) = \varnothing$).

The notation $a \rightarrow b \in S$ simply indicates that event a from TA_1 must be synchronized with event b from TA_2, but event b can occur independently of a. Given $a \leftrightarrow b \in S$, it indicates that event a from TA_1 and b from TA_2 may synchronize with each other, or occur independently.

This parallel composition is defined in terms of the general parallel merge operator $\|_M$ in the UTP [40]:

$$A_1 \mid[E]\mid A_2 \bullet S \mathrel{\widehat=} (((A_1;\ idle) \|_M A_2) \vee (A_1 \|_M (A_2;\ idle))); ((ok \Rightarrow \textsc{Skip}) \wedge (\neg ok \Rightarrow tr \overset{t}{\preceq} tr'))$$

Take note that $\textsc{Skip}$ is a semantic predicate which preserves the observations, that is, $\textsc{Skip} \mathrel{\widehat=} (obs' = obs)$, where obs denotes all observables.

An *idle* process, which may either wait or terminate, follows after each of the two automata. This is to allow each of the automata to wait for its partner to terminate.

$$idle \mathrel{\widehat{=}} ok' \wedge no_interact(trace) \wedge state' = state$$

The merge predicate M is defined as,

$M \mathrel{\widehat{=}} ok' = (0.ok \wedge 1.ok) \wedge wait' = (0.wait \vee 1.wait) \wedge state' = (0.state \oplus 1.state) = (1.state \oplus 0.state) \wedge tr' \in syn(0.tr, 1.tr, E, S) \wedge \#tr' = \#0.tr = \#1.tr \wedge cval' = (0.cval \oplus 1.cval) = (1.cval \oplus 0.cval)$

Given two timed traces tr_1, tr_2, and a set of events E, and a set of pairs of partial/sometime synchronizations S, the set $syn(tr_1, tr_2, E, S)$ is defined inductively as follows.

$$
\begin{aligned}
& syn(tr_1, tr_2, E, \varnothing) \mathrel{\widehat{=}} syn(tr_2, tr_1, E, \varnothing) \\
& syn(\langle\rangle, \langle\rangle, E, S) \mathrel{\widehat{=}} \{\langle\rangle\} \\
& syn(\langle(t, r)\rangle, \langle\rangle, E, S) \mathrel{\widehat{=}} \{\langle(t', r)\rangle \mid t' \in (t \underset{E\ S}{\parallel} \langle\rangle)\} \\
& syn(\langle\rangle, \langle(t, r)\rangle, E, S) \mathrel{\widehat{=}} \{\langle(t', r)\rangle \mid t' \in (\langle\rangle \underset{E\ S}{\parallel} t)\} \\
& syn(\langle(t_1, r_1)\rangle \frown tr_1, \langle(t_2, r_2)\rangle \frown tr_2, E, S) \mathrel{\widehat{=}} \\
& \qquad \{\langle(t', r')\rangle \frown u \mid t' \in (t_1 \underset{E\ S}{\parallel} t_2) \wedge r' = r_1 \cup r_2 \wedge \\
& \qquad u \in syn(tr_1, tr_2, E, S)\}
\end{aligned}
$$

$s \underset{E\ S}{\parallel} t$ is used to merge untimed traces s and t into one untimed trace, where E is the set of events to be synchronized, S is the set of partial/sometime synchronization

pairs.

In the following clauses, e, e_1 are representative elements of E (events), x, x_1 represent communication events not residing in E or S, $a \rightarrow b$, $a_1 \rightarrow b_1$ are representative partial synchronization pairs from S, while $c \leftrightarrow d$, $c_1 \leftrightarrow d_1$ are representative sometime synchronization pairs from S. Let $y, y_1, y_2 \in \{x, x_1, b, b_1, c, d, c_1, d_1\}$.

Let $z, z_1, z_2 \in \{e, a, e_1, a_1\}$. Moreover, we use $k(a, b)$ to denote the synchronization of a and b.

$$s \underset{E\ \varnothing}{\parallel} t \mathrel{\widehat{=}} t \underset{E\ \varnothing}{\parallel} s \qquad \langle\rangle \underset{E\ S}{\parallel} \langle\rangle \mathrel{\widehat{=}} \{\langle\rangle\}$$

$$\langle z\rangle \underset{E\ S}{\parallel} \langle\rangle \mathrel{\widehat{=}} \langle\rangle \underset{E\ S}{\parallel} \langle z\rangle \mathrel{\widehat{=}} \{\}$$

$$\langle y\rangle \underset{E\ S}{\parallel} \langle\rangle \mathrel{\widehat{=}} \langle\rangle \underset{E\ S}{\parallel} \langle y\rangle \mathrel{\widehat{=}} \{\langle y\rangle\}$$

$$\langle y\rangle ^\frown s \underset{E\ S}{\parallel} \langle z\rangle ^\frown t \mathrel{\widehat{=}} \{\langle y\rangle ^\frown l \mid l \in (s \underset{E\ S}{\parallel} \langle z\rangle ^\frown t)\},\ z \rightarrow y \notin S$$

$$\langle z\rangle ^\frown s \underset{E\ S}{\parallel} \langle y\rangle ^\frown t \mathrel{\widehat{=}} \{\langle y\rangle ^\frown l \mid l \in (\langle z\rangle ^\frown s \underset{E\ S}{\parallel} t)\},\ z \rightarrow y \notin S$$

$$\langle e\rangle ^\frown s \underset{E\ S}{\parallel} \langle e\rangle ^\frown t \mathrel{\widehat{=}} \{\langle e\rangle ^\frown l \mid l \in (s \underset{E\ S}{\parallel} t)\}$$

$$\langle z_1\rangle ^\frown s \underset{E\ S}{\parallel} \langle z_2\rangle ^\frown t \mathrel{\widehat{=}} \{\},\ \text{where } z_1 \neq z_2$$

$$\langle y_1\rangle ^\frown s \underset{E\ S}{\parallel} \langle y_2\rangle ^\frown t \mathrel{\widehat{=}} \{\langle y_1\rangle ^\frown l \mid l \in (s \underset{E\ S}{\parallel} \langle y_2\rangle ^\frown t)\} \cup$$
$$\{\langle y_2\rangle ^\frown l \mid l \in (\langle y_1\rangle ^\frown s \underset{E\ S}{\parallel} t)\},\ \text{where } y_1 \leftrightarrow y_2 \notin S$$

$$\langle a\rangle ^\frown s \underset{E\ S}{\parallel} \langle b\rangle ^\frown t \mathrel{\widehat{=}} \{\langle k(a,b)\rangle ^\frown l \mid l \in (s \underset{E\ S}{\parallel} t)\} \cup$$
$$\{\langle b\rangle ^\frown l \mid l \in (\langle a\rangle ^\frown s \underset{E\ S}{\parallel} t)\}$$

$$\langle b\rangle^\frown s \underset{E\ S}{\parallel} \langle a\rangle^\frown t \mathrel{\widehat=} \{\langle k(a,b)\rangle^\frown l \mid l \in (s \underset{E\ S}{\parallel} t)\} \cup$$
$$\{\langle b\rangle^\frown l \mid l \in (s \underset{E\ S}{\parallel} \langle a\rangle^\frown t)\}$$
$$\langle c\rangle^\frown s \underset{E\ S}{\parallel} \langle d\rangle^\frown t \mathrel{\widehat=} \{\langle k(c,d)\rangle^\frown l \mid l \in (s \underset{E\ S}{\parallel} t)\} \cup$$
$$\{\langle c\rangle^\frown l \mid l \in (s \underset{E\ S}{\parallel} \langle d\rangle^\frown t)\} \cup \{\langle d\rangle^\frown l \mid l \in (\langle c\rangle^\frown s \underset{E\ S}{\parallel} t)\}$$

A network of timed automata is the parallel composition $A_1 \parallel A_2 \parallel ... \parallel A_n$ of a set of timed automata $A_1, A_2, ..., A_n$.

7.3.3 The Semantics of Class

OZTA has two kinds of classes, active and passive ones. The behavior of (an object of) an active class can be specified by a record of its continuous interactions with its environment via its time automaton specifications, whereby any update on its data state is hidden. A passive class does not have its own thread of control and its state and operations (processes) are available for use by its controlling object.

To address issues like class declarations, their well-formed definitions and their composition, we adopt the same class model for OZTA from TCOZ since TCOZ and OZTA have very similar object-orientation features except that the Timed CSP operations are now replaced with timed automata. More detailed information on the semantics of class model, such as class encapsulation, inheritance, and dynamic binding, can be referred to [59].

7.4 Conclusion

In this chapter, we firstly further enhanced the OZTA notation by introducing a set of timed patterns to its TA part, which can facilitate specifying the dynamic and timing features of complex real-time systems in a systematic way. Secondly, we presented an enhanced semantics in unifying theories of programming. This semantics model of OZTA provides the foundation for language understanding, reasoning and especially, tool construction which will be discussed in the next chapter.

Chapter 8

OZTA Tool Support and Case Study

8.1 Introduction

The specification of complex real-time systems requires powerful mechanisms for modelling data structure, concurrency and real-time behavior as well as tool-support for building up and verifying the established models.

HighSpec is an interactive system for composing and checking OZTA models. **HighSpec** supports all the modelling features of OZTA and provides powerful checking capabilities. The main functionalities are listed below,

- Automated systematic TA design via timed pattern,
- Schema editing and expansion,
- Syntax and type checking,
- Projection to TA model checker, UPPAAL, for verification,
- Generation of LaTeXpresentation of established models.

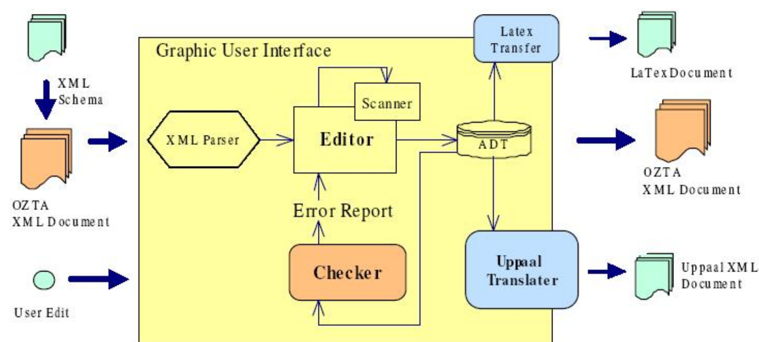


Figure 8.1: Overview of **HighSpec**

Figure 8.1 provides an overview of **HighSpec**: it mainly consists of five components, i.e., a powerful GUI editor to compose Object-Z schemas and the corresponding timed automaton, a syntax and type checker, a LaTeX code generator for read of established OZTA models and model translators to UPPAAL for verification. The input language is based on the syntax and semantics we presented in the previous chapters. The output can either be an XML representation of OZTA models or LaTeX source files of OZTA models; **HighSpec** can also generate projections of OZTA models which are ready to be taken as input for simulation and verification in UPPAAL.

8.2 Modeling

HighSpec provides powerful automated support with user commands for directing the design of an OZTA model. All the information input from the user interface is collected into a special Abstract Data Type (ADT) designed according to the integrated syntax of Object-Z and Timed Automata. Basically, the information can be divided into three parts. The *system configuration part* supports declaration of global information and classes. Each OZTA class contains an *Object-Z part* and a *Timed Automaton part*. The *Object-Z part* contains the information of Object-Z schema such as state variables, operations and pre/post condition of operations, and the *Timed Automaton part* captures the information about the control flow between the Object-Z operations and related timing behaviors according to system requirements.

The Object-Z schema information recorded in the ADT plays an important role for designing the corresponding timed automaton. Once the definitions of the Object-Z operation schemas in an OZTA class are completed, its corresponding timed automaton can be generated in a top-down way by repeatedly applying the timed patterns to fulfil the control and timing requirements. Guidelines for designing the TA part of OZTA models can be found in chapter 3.4.

8.3 Checking

An OZTA syntax and type checker is implemented in **HighSpec** for checking the validity of an OZTA model, as well as model translators for verifying various properties of OZTA models by reusing TA's tool support.

8.3.1 Syntax and Type Checker

The OZTA language has a quite complex syntax since it includes the Z notation. It is easy, especially for an inexperienced OZTA user, to make some syntax or type errors. **HighSpec** is able to detect and report such errors. A full set of type checking rules can be found in our technical report [19]. The class diagram of this checker is shown in Figure 8.2.

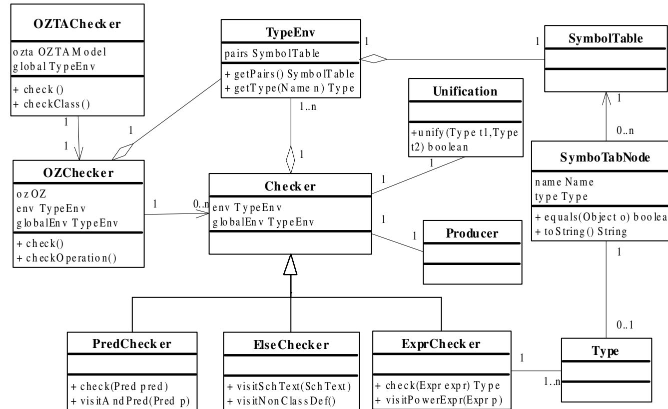


Figure 8.2: Class Diagram of the type checker

8.3.2 OZTA to UPPAAL

HighSpec adheres to light-weight principles: instead of implementing a model checker for OZTA from scratch, we choose to project the integrated requirement models into UPPAAL models so that UPPAAL can be utilized to simulate the dynamic behaviors of the OZTA model and verify various kinds of properties.

The translation process can be automated by employing XML/XSL technology. In our previous work [71], the syntax of Z-family languages, i.e., Z/Object-Z/TCOZ, has been defined using XML Schema and supported by the ZML tool. As the UPPAAL tool can read XML representations of Timed Automata, the automatic projection of the OZTA model (in ZML) to a TA model (in UPPAAL XML) is implemented in our OZTA tool.

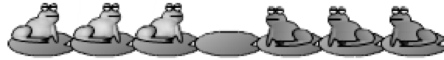
The UPPAAL translator in **HighSpec** takes an OZTA specification represented in XML, and outputs an XML representation of a Timed Automata specification which has its own defined style file DTD by UPPAAL. The automatic transforma-

tion is achieved firstly by making use of our OZTA ADT to easily extract information from the specification. A TA interface is then built according to the UPPAAL document structure, e.g., each TA document contains multiple templates and each template contains some states, their transitions and transition conditions. The outcome of our translator is the UPPAAL's XML representation of TA, which is ready to be taken into UPPAAL as input for future verification and simulation.

Although our projection can handle most of the TA information of an OZTA model, one limitation needed to be pointed out is that: coupled with operation schema predicates and data structures, the semantics of operation states in the TA part of an OZTA model is slightly different from those of states in UPPAAL. However, the main structure of the OZTA automata model is still consistent with that of the UPPAAL model by regarding the OZTA operation states as abstracted automata which need further implementation. This gap between the OZTA's TA model and UPPAAL's TA model can be remedied by some manual work on the operation states, namely, to further embody these abstracted automata by adding the data information.

8.4 Case Study: A Frog Puzzle Game

In this chapter, we use a frog puzzle game model to demonstrate the use of **HighSpec**. The puzzle specifies that, given seven stones, three white frogs on the left facing right and three black frogs on the right facing left. A frog can move



in the direction it is facing to an empty stone, which is adjacent or is reached by jumping over a frog on an adjacent stone. To complicate the puzzle, we add some timing constraints to the moves of frogs, i.e., each frog takes at least 1 time unit, but no more than 2 time units to move to its next position. We define that the puzzle is solved if a sequence of moves can be found that will exchange the positions of the black and white frogs within 30 time units.

In the following, the case study will be carried out starting with the OZTA model design, followed by the syntax and type checking, then projecting to UPPAAL, and lastly generating the LaTeX document.

8.4.1 Design of OZTA Models

Firstly, we build the OZTA model for this frog puzzle. Screen-shots are provided in Appendix A.3 to briefly illustrate the Object-Z and structural TA design.

$Posn == 1..7$

Puzzle

$wf, bf : \mathbb{P}\, Posn$
$nf : Posn$
$win : \mathbb{B}$

$\#wf = 3 \land \#bf = 3$

Win

$\Delta(win)$

$bf = \{1,2,3\} \land wf = \{5,6,7\}$
$nf = 4$
$win' = \text{true}$

INIT

$wf = \{1,2,3\}$
$bf = \{5,6,7\}$
$nf = 4$

WhiteMove

$\Delta(wf, nf)$

$\neg(bf = \{1,2,3\} \land wf = \{5,6,7\} \land nf = 4)$
$nf' \ \underline{leftw}\ nf$
$wf' = wf \cup \{nf\} - \{nf'\}$

BlackMove

$\Delta(bf, nf)$

$\neg(bf = \{1,2,3\} \land wf = \{5,6,7\} \land nf = 4)$
$nf' \ \underline{rightb}\ nf$
$bf' = bf \cup \{nf\} - \{nf'\}$

$Lose_2$

$\Delta(win)$

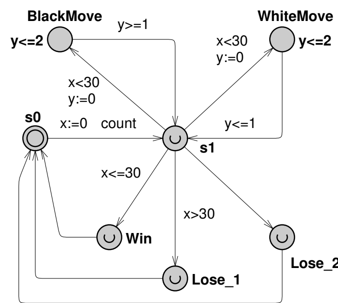
$nf \notin rightb(\!|\ bf\ |\!)$
$nf \notin leftw(\!|\ wf\ |\!)$
$win' = \text{false}$

$Lose_1$

$\Delta(win)$

$win' = \text{false}$

$x, y : \textbf{clock}$



$rightb : Posn \leftrightarrow Posn$

$\forall i, j : Posn \bullet$
$i \ \underline{rightb}\ j \Leftrightarrow i = j + 1 \lor i = j + 2$

$leftw : Posn \leftrightarrow Posn$

$\forall i, j : Posn \bullet$
$i \ \underline{leftw}\ j \Leftrightarrow i = j - 1 \lor i = j - 2$

In this model, we define the empty stone also as a frog object nf. *BlackMove* captures the position exchanges between the black frogs and the empty stone; same for *WhiteMove*; *Win* defines the situation when the puzzle is solved. The

game begins with a *count* event after its initial state; player will lose the game when the time is out as described by $(x > 30) \bullet Lose_1$ or whenever the frogs are all jammed by each other in the middle way as described by $Lose_2$. The graphical TA part of the model can be derived from the following textual specification according to the *sequential composition*, *external choice*, *deadline*, *waituntil*, and *recursion* patterns:

$$
\begin{array}{l}
TA \mathrel{\widehat=} \mu\, Y \bullet (x := 0)(count) \bullet \\
\quad \mu\, X \bullet ((x < 30) \bullet BlackMove \bullet Deadline(y, 2) \bullet WaitUntil(y, 1);\; X) \\
\quad \Box\; ((x < 30) \bullet WhiteMove \bullet WaitUntil(y, 1) \bullet Deadline(y, 2);\; X) \\
\quad \Box\; ((x <= 30) \bullet Win;\; Y) \;\Box\; ((x > 30) \bullet Lose_1;\; Y) \;\Box\; (Lose_2;\; Y)
\end{array}
$$

8.4.2 Syntax and Type Check

After building the model, we will demonstrate several kinds of syntax errors and type errors that can found and reported by **HighSpec**.

$$
\begin{array}{|l}
wf, bf : Posn \\
nf : Posn \\
win : \mathbb{B} \\
\hline
\#wf = 3 \land \#bf = 3
\end{array}
$$

For example, when the state schema shown above is checked, two error messages would be reported as:

```
Error: not set type in size Expr: #wf (Posn)

Error: not set type in size Expr: #bf (Posn)
```

Since *wf* and *bf* are defined as type *Posn*, which is not a set.

8.4.3 Model Checking Using UPPAAL

For this frog puzzle game, **HighSpec** can automatically extract the system configuration and TA part from the ADT of the OZTA model and generate an XML representation of the UPPAAL model. The state variables bf, wf, nf are projected to the UPPAAL model as global *int* variables $bf[3], wf[3], nf$. Due to the limited expressiveness for data manipulation in UPPAAL, we need to respectively expand *BlackMove* and *WhiteMove* into three branches. The predicates in the operation schemas of the OZTA model are projected as guards on the corresponded transitions. The final UPPAAL model can be generated for verification and model-checking in this way as shown in Figure 8.3.

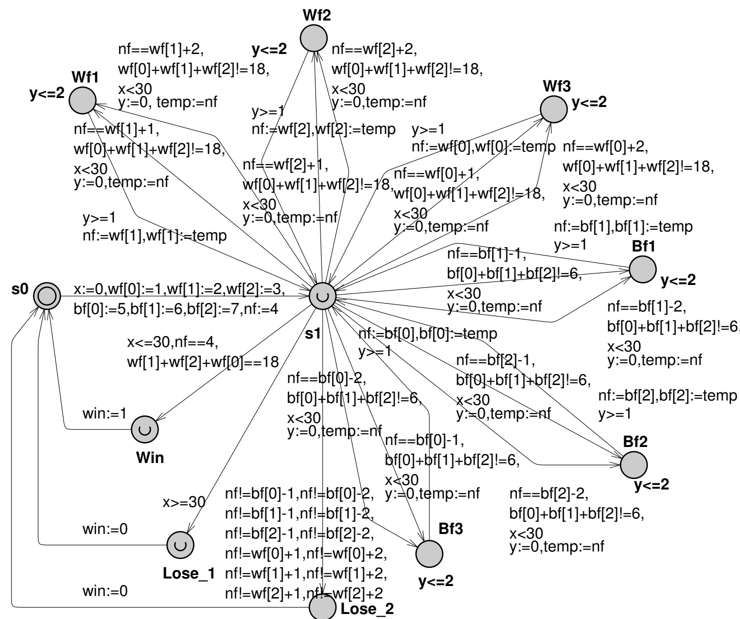


Figure 8.3: Frog Puzzle Model in UPPAAL

To find the solution of this frog puzzle, we can check the following property in UPPAAL.

$$E <> P.Win$$

which means that there exists a sequence of moves that will exchange the positions of the black and white frogs within 30 time units.

UPPAAL verified that this property holds for this given model. Solutions of the puzzle can be visualized in UPPAAL's simulator by running its diagnostics trace.

8.4.4 Generation LaTeX Document

HighSpec supports generation of LaTeX source code. This will be demonstrated using the frog puzzle model once again, its correspondent LaTeX source code can be generated by the tool as follows.

```
\documentclass{llncs}

\usepackage{pt}

...

\usepackage{lncsexample}

\begin{document}

\begin{class}{Puzzle}

\begin{anonschema}

bf : \power Frog \\

wf : \power Frog \\

...

\end{anonschema}

\begin{init}

bf= \{1,2,3\} \\

wf=\{5,6,7\} \\ nf=4 \\

\end{init}

...

\begin{op}{BlackMove}

\Delta (bf,nf) \\
```

```
\where %
!( bf=\{5,6,7\}\land wf=\{1,2,3 \} \\ \land nf=4 ) \\
nf'~ \underline{next1} ~ nf \\
bf'= bf \uni \{nf\}-\{nf'\}
\end{op}
... \where \t1 \mbox{\epsfxsize=80mm
\epsfbox{./Puzzle_Puzzle.eps}}
\end{class}
\end{document}
```

8.5 Conclusion

In this chapter, we presented, **HighSpec**, a distinguished tool support for modelling and verification of complex real-time systems in that:

- it is built for a high-level integrated formal method, i.e., OZTA, which provides powerful mechanisms for capturing various aspects of complex real-time systems such as data structure, concurrency and timing constraints. Many of the integrated formal methods such as TCOZ have no exclusive tool support even just for editing, not to say verification tools. This is mainly because integrated high-level specification languages usually contain various aspects of abstracted system information, which makes the implementation of verification tool from scratch impossible. The problem is circumvented in **HighSpec** by projecting the integrated OZTA model to TA model so that TA's tool support can be reused for checking.
- it implemented a novel mechanism for establishing TA models in a systematic

way by using a set of timed patterns to specify common control and timing behaviors. Comparing to traditional TA tool support such as UPPAAL, **HighSpec** effectively releases users from the trouble to manually cast these common behaviors into clock variables, states, and transition conditions so that they can focus on the specification level of a model rather than the implementation.

Chapter 9

Conclusion and Future Directions

9.1 Summary and Contributions

There are a variety of formal techniques and tools that are reasonably mature and well-understood in the literature. These all have similarities and differences to some degree. It is important for the formal method community to understand and be able to explain how various techniques differ one from another. Such an understanding is necessary before any real evaluation of individual strengths and weakness of the techniques is possible. The techniques under consideration, Object-Z/TCOZ and TA lie at each end of the spectrum of formal modelling techniques. Object-Z/TCOZ is good at structurally specifying high-level requirements for complex systems, while TA is good at designing timed models in simple clock constraints but with highly automatic tool support. It is of great interest to investigate the links of these two kinds of different languages so that they can benefit each other. One important contribution of this thesis is that the investigation on the strengths and links between those two modelling techniques, TCOZ and TA, leads us to an interesting research result, i.e., timed composable patterns (reminiscent of 'design patterns' in object-oriented modelling). These patterns are formally defined in Z and the process algebra-like compositional nature is preserved in the graphical representations. These timed composable patterns:

- not only provide a proficient interchange media for transforming TCOZ specifications into TA designs;
- but also provide a generic reusable framework for systematically developing models of real-time systems in TA alone.

Another main contribution is that we provide two effective ways for building and checking complex real-time models in a unified framework.

- With the projection approach, we demonstrate one possible engineering process for modelling real-time complex systems, i.e., in the life cycle of software development, TCOZ can be adopted for building high-level abstract models and TA's tool support can be reused for model-checking TCOZ model by using the timed patterns as the interchange media for projection from TCOZ models to TA models.

  Based on the timed patterns, a set of transformation rules is defined. We also investigate the semantic equivalence between TCOZ processes and Timed Automata and provide a full proof for the correctness of the transformation. Little theoretical work has been done in this area except that Joel and James in their recent purely theoretical investigation [55] have demonstrated that Timed CSP has equal expressiveness with closed timed automata. In this context, this work complements Joel and James's work.

  Since TCOZ is a superset of Timed CSP, one consequence of this work is that a semantic link and a practical translation tool from Timed CSP to TA has been achieved so that TA tools, i.e., UPPAAL can also be used to check Timed CSP timing properties.

  One closely related area of research to ours is J. Hoenicke and E.-R. Olderog's work on integration of CSP, Object-Z (OZ) and Duration Calculus (DC) [41],

in which CSP-OZ-DC benefits from Timed Automata's tool support, i.e. UPPAAL, for model-checking by transforming the DC part of a system model into a timed automaton. The technical difference between our study and theirs is the development of the generic TA patterns, i.e., we not only focus on lending Timed Automata's tool support to TCOZ for model-checking TCOZ, but also on lending TCOZ's structure to systematic TA designs by providing a set of composable timed patterns. Furthermore, their work mainly focuses on a smooth integration of the underlying semantic models of CSP, OZ, and DC and its use for verifying properties of CSP-OZ-DC specifications. Furthermore, we provide soundness proof for our transformation from TCOZ to TA while their work does not. Another related formalism for modeling real-time systems is TRIO [13]. This notation uses a notion of interface diagram which is quite different from the TCSP-featured TCOZ notation in modeling dynamic behaviors. TRIO has been compared to TCOZ and referenced in the TCOZ paper [50]. Similar to our work, there are also some other real-time formalisms such as Harel's Statecharts [33], Jahanian and Mok's Modechart [47] which have been provided by a translation/projection approach with tool support like model-checking. However, the main contribution of our work is the construction of various generic and reusable timed patterns which are reminiscent of object oriented patterns, even though our work was initially orientated to develop tool support for TCOZ.

This part of our work also inspired an interesting question: can we integrate Object-Z and Timed Automata directly? In this way, not only the wonderful

tool support of TA can be reused straightforward, but also the timed composable patterns can be directly utilized for systematic TA designs. It motivated us to further our study on an integration approach.

- In the integrating approach, rather than taking the transformation point of view, we developed a novel integrated formal language which combines Object-Z with TA. Such an effective combination of Object-Z and TA not only helps Object-Z with a real-time modelling capability, but also helps TA with enhanced structure and state modelling features. The result of such combination is another powerful unified method for designing complex real-time systems.

  The OZTA notation is enhanced by introducing the set of timed patterns as language constructs that can specify the dynamic and timing features of complex real-time systems in a systematic way. We also presented a semantic model of OZTA in Unifying Theories of Programming which provides the semantic foundation for language understanding, reasoning and tool construction. Based on the semantic model, we constructed **HighSpec**, an interactive system which supports editing, type-checking OZTA models as well as transforming OZTA models into TA models so that we can utilize TA model-checkers, e.g., UPPAAL, for simulation and verification.

  One closely related area of research to ours is on integration of Object-Z with various timed calculi. For example, Object-Z is combined with Timed CSP [62] in [53, 17], with the timed refinement calculus [52, 27] in [68]

and with the duration calculus [79] in [41]. Indeed, those combinations have made improvements in comparison to some early conservative framework approaches [18, 57].

The technical difference between our approach to the others has been the way to clearly separate functionalities and timed behavior, and the use of graph based Timed Automata, instead of the calculi to capture behavior. One clear benefit of our approach is that many existing well developed tools [58, 16, 69, 72] for TA can be used to check the timed behavior of the design models. In addition to the benefit of bring graphical appeal in capturing the object behavior of Object-Z classes, our approach also provides a way to structure TA using Object-Z classes, so that the scale problem of TA can be managed. The novel communication mechanism developed in our approach is also more flexible and expressive than CSP channels. For example, arbitrary communication between various objects can be captured at the composite class level with the elegant communication links.

Another related work is the combination of Z with graphical diagrams, i.e., Statecharts [33] and Petri-nets [7]. For example, in [11] a framework is presented to link Z with Statecharts; it treats Z operation schemas as state transition links in Statecharts. Similarly, the language OZS [30] blends Object-Z with Statecharts and treats Object-Z operations as state transition links. Combinations of Z and Petri-net have also been investigated in [35, 34]. All those approaches suffer a common drawback the states in the graph have

no systematic correspondence in Z or Object-Z parts. The issues of object composition and timing have not been addressed. Our approach is different: we treat Object-Z operations as states (i.e., TA locations) instead of state transitions (i.e., TA switches) and furthermore we have a systematic naming convention for all switches. Object composition and real-time issues are the main focus points in our approach.

9.2 Future Work

In this thesis, we build a unified framework to model and check real-time complex systems using Object-Z/TCOZ and Timed Autoamta. This frame can be further extended or improved by integrating more useful techniques from multiple domains so that various properties of an TCOZ or OZTA model can be analyzed in the projected domains.

One future work can be projecting TCOZ/OZTA to Alloy to analyze the OZ part of a TCOZ/OZTA model. Alloy is a structural modelling language based on first-order logic, for expressing complex structural constraints and behavior. The Alloy Analyzer(AA) supports two kinds of automatic analysis: simulation, in which the consistency of an invariant or operation is demonstrated by generating an instance; and checking, in which a consequence of the specification is tested by attempting to generate a counterexample. The essential constructs of Alloy are signature, fact, function, predicate and assertion. Projection guideline for important primitives of

OZ are presented as follows.

- Types: All the basic types of OZ are defined as,

  sig *TypeRef* {}{*sigFact*} in Alloy.

- State variables: The State variables of an OZ class are projected as fields of signature **sig***State*{}. However, not all state variables are necessarily to be projected into Alloy model. Those which are irrelevant for the properties to be checked in Alloy can be abstracted away.

- The predicates of the state schema are projected as facts in Alloy. The keyword **disj** in Alloy can be used to indicate that the variables declared in the OZ state schema are disjoint.

- Functions: OZ Functions can be defined as predicates or functions in Alloy. When an OZ function returns a value, then it should be projected as a function in Alloy, otherwise it should be projected as a predicate in Alloy. Each parameter of the OZ functions corresponds to a parameter in the Alloy models.

- Operation schema: Each operation schema *Op* with an empty Δ-list is projected as a predicate; each OZ operation schema *Op* with a Δ-list, $\Delta(s)$, is projected as a fact *Op* in Alloy. The predicates of those operation schemas can be projected as a predicate $\mathit{pred}(s, s')\{\}$. Each state variable in the Δ-list of an OZ operation schema corresponds to a pair of parameters of the same

type in the Alloy predicate, which respectively represent the pre-state and pose-state of the OZ state variable. i.e.,

```
fact Op{ all s, s': State{ pred(s,s') } }
```

- Ordering of state transitions: To reason about the OZ part of an OZTA model in Alloy, when the ordering of the state transitions of its OZ part is to be analyzed, we need to import a module *ordering* in Alloy to record the instances of its OZ state schemas. The instance of ordering *ord* records pairs of pre-state and post-state for its OZ state variables.

```
open util/ordering[State] as ord module util/ordering[elem] one
sig Ord {
   first_, last_: elem,
   next_, prev_: elem -> lone elem}
```

  The OZ *Init* operation is now projected as a fact which constrains the first element of *ord* in order to record the initial state of the OZ state variables.

Another future work is to use Constraint Logic Programming (CLP) [45] as the underlying reasoning support for TCOZ or OZTA models to prove traditional safety properties and beyond, such as reachability, deadlock-freeness, timewise refinement relationship, lower or upper bound of variables, etc. CLP is designed for mechanized proving based on constraint solving. CLP has been successfully applied to model programs and transition systems for the purpose of verification [31, 46].

Lastly is that we plan to further enhance our **HighSpec** tool by extending the current set of TA patterns into a dynamic pattern library so that new patterns

can be defined by system designers and added into the pattern library for future reuse.

Bibliography

[1] UPPAAL2K. http://www.docs.uu.se/docs/rtmv/uppaal/xml/, 2003.

[2] J. Abrial. The B tool (abstract). In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM – The Way Ahead*, volume 328 of *Lect. Notes in Comput. Sci.*, pages 86–85. Springer-Verlag, 1988.

[3] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[4] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] Rajeev Alur, Costas Couroubetis, and David L. Dill. Model-checking for Real-time Systems. In *proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[6] K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK.* Springer-Verlag, June 1999.

[7] J.-L. Baer. Modelling Architectural Features with Petri Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986 (part II)*, volume 255 of *Lect. Notes in Comput. Sci.*, pages 258–277. Springer-Verlag, 1987.

[8] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. In Maurice Naftalin, B. Tim Denvir, and Miquel Bertran, editors, *FME*, volume 873 of *Lecture Notes in Computer Science*, pages 105–117. Springer, 1994.

[9] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28:56–63, 1995.

[10] P. Brooke. *A Timed Semantics for a Hierarchical Design Notation.* PhD thesis, University of York, 1999.

[11] R. Bussow and W. Grieskamp. A Modular Framework for the Integration of Heterogeneous Notations and Tools. In Araki et al. [6], pages 211–230.

[12] M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods, Turku, Finland*, Lect. Notes in Comput. Sci. Springer-Verlag, May 2002.

[13] D. Mandrioli C. Ghezzi and A. Morzenti. Trio: A logic language for executable specifications of real-time system. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[14] ORA Canada. Z/EVES. http://www.ora.on.ca/z-eves/, 2002.

[15] Albert M. K. Cheng. *Real-time systems : scheduling, analysis, and verification.* John Wiley and Sons, 2002.

[16] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III: Verification and Control*, pages 208–219. Springer, 1996.

[17] J. Derrick. Timed CSP and Object-Z. In *3nd International Conference of Z and B Users (ZB'03)*, LNCS. Springer, June 2003.

[18] J. S. Dong, J. Colton, and L. Zucconi. A Formal Object Approach to Real-Time Specification. In *the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, Seoul, Korea, December 1996. IEEE Computer Society Press.

[19] J. S. Dong, P. Hao, S. C. Qin, and X. Zhang. OZTA. Technical report TRC6/05, School of Computing, National University of Singapore, 2005.

[20] Jin Song Dong, Yuan Fang Li, Jing Sun, Jun Sun, and Hai Wang. XML-based Static Type Checking and Dynamic Visualization for TCOZ. In *4th International Conference on Formal Engineering Methods*, pages 311–322. Springer-Verlag, October 2002.

[21] Jin Song Dong, Hao Ping, Sun Jun, and Zhang Xian. A Reasoning Method for Timed CSP based on Constraint Solving. In *8th International Conference on Formal Engineering Methods*. Springer-Verlag, October 2006.

[22] J.S. Dong, R. Duke, and P. Hao. Integrating Object-Z with Timed Automata. In *The 10th IEEE International Conference on Engineering of Complex Computer System*, Shanghai, China, 2005.

[23] J.S. Dong, P. Hao, and B. Mahony. Formal Designs for Embedded and Hybrid Systems. *International Journal on Software Engineering and Knowledge Engineering*, 15(2):373–378, 2005.

[24] J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi. Timed Patterns: TCOZ to Timed Automata. In J. Davies, W. Schulte, and M. Barnett, editors, *The 6th IEEE International Conference on Formal Engineering Methods*, Seattle, USA, 2004.

[25] J.S. Dong, P. Hao, S.C. Qin, and X. Zhang. The Semantics and Tool Support of OZTA. In Kung-Kiu Lau and Richard Banach, editors, *The 7th IEEE International Conference on Formal Engineering Methods*, Manchester, UK, 2004.

[26] J.S. Dong, P. Hao, S.C. Qin, and X. Zhang. HighSpec: a Tool for Building and Checking OZTA Models. In *The 28th International Conference on Software Engineering*, Shanghai, China, 2006. ACM Press.

[27] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A Set-theoretic Model for Real-time Specification and Reasoning. In *Mathematics of Program Construction*, 1998.

[28] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [6].

[29] W. Grieskamp, T. Santen, and B. Stoddart, editors. *IFM'00: Integrated Formal Methods, Dagstuhl Castle, Germany*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.

[30] J. P. Gruer, V. Hilaire, A. Koukam, and P. Rovarini. Heterogeneous formal specification based on Object-Z and startechart: semantics and verification. *J. Systems and Software*, 2004.

[31] G.l Gupta and E. Pontelli. A Constraint-based Approach for Specification and Verification of Real-time Systems. In *IEEE Real-Time Systems Symposium*, pages 230–239, 1997.

[32] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 07:11–19, 1990.

[33] David Harel and Eran Grey. Executable Object Modeling with Statecharts. *IEEE computer*, 30(7):31–42, 1997.

[34] X. He. PZ nets a formal method integrating Petri nets with Z. *Information & Software Technology*, 43(1):1–18, 2001.

[35] M. Heiner and M. Heisel. Modeling safety-critical systems with Z and Petri nets. In *International Conference on Computer Safety, Reliability and Security, LNCS, Springer*, pages 361–374, 1999.

[36] C. L. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In *Proceedings of RTSS'94, Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE Computer Society Press.

[37] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 111(2):193–243, 1994.

[38] M.G. Hinchey and S.A. Jarvis. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill International Series in Software Engineering, 1995.

[39] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[40] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[41] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes, Data and Time. In *IFM'02*, pages 245–266, 2002.

[42] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[43] C. Hung. CCS used as a proof-assistant tool. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 387–398. North-Holland, 1986.

[44] S. Qin J. Sun J. S. Dong, P. Hao and Y. Wang. Timed Composable Patterns: From TCOZ to Timed Automata. *submitted to Transactions on Software Engineering.*

[45] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[46] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP Proof Method for Timed Automata. In *RTSS*, pages 175–186, 2004.

[47] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, 36(8):961–975, August 1987.

[48] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[49] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual. http://www.formal.demon.co.uk/FDR2.html, May 2000.

[50] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[51] B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 2002. (accepted).

[52] B. P. Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, University of Queensland, 1991. Available as `ftp://ftp.it.uq.edu.au/pub/Thesis/brendan_mahony.ps.Z`.

[53] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Press.

[54] MIT Lab of Computer Science. The Alloy Analyzer, 2002. http://sdg.lcs.mit.edu/alloy/.

[55] J. Ouaknine and J. Worrell. Timed CSP = Closed Timed Automata. In *Proceedings of EXPRESS 02*, volume 68(2) of *ENTCS*, 2002.

[56] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[57] K. Periyasamy and V.S. Alagar. Adding Real-Time Filters to Object-Oriented Specification of Time Critical Systems. In *the 1998 IEEE Workshop on Industrial-strength Formal specification Techniques*, Boca Raton, Florida, USA, October 1998. IEEE Press.

[58] Paul Pettersson and Kim G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[59] S. C. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation of TCOZ in Unifying Theory of Programming. In *Formal Methods(FM'03)*, LNCS 2805, pages 321–340. Springer-Verlag, 2003.

[60] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[61] S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[62] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.

[63] Steve Schneider. *Concurrent and Real-time Systems.* John Wiley and Sons, 2000.

[64] A. Sherif and J. He. Towards a Timed Model for Circus. In *The 2th IEEE International Conference on Formal Engineering Methods*, Shanghai, China, 2002.

[65] G. Smith. A Fully Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[66] G. Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[67] G. Smith and J. Derrick. Specification, Refinement and Verification of Current Systems — An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[68] G. Smith and I. Hayes. Towards Real-Time Object-Z . In Araki et al. [6].

[69] M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proceedings of Workshop on Real-time Tools*. Aalborg, August 2001.

[70] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[71] J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13:329–356, 2002.

[72] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *Proceedings of the 7th Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 546–562. Springer, 1996.

[73] World Wide Web Consortium (W3C). Extensible Markup Language (XML). http://www.w3.org/XML.

[74] F. Wang. Symbolic verification of complex reai-time systems with clock-restriction diagram. In M.Kim, B.Chin, S.Kang, and D.Lee, editors, *Proceedings of International Conference on Formal Techniques for Networked and Distributed Systems*, volume 197 of IFIP Conference Proceedings, pages 235–250. Kluwer, August 2001.

[75] J. Woodcock and A. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[76] Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[77] Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Operational Semantics for Model Checking Circus. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2005.

[78] Zhang Xian, Sun Jun, and Hao Ping. A Tool for Building and Reasoning Timed CSP Models. In *Formal Methods 2006*.

[79] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

Appendix A

A.1 TCOZ Notation

Notation	Explanation
$c : \mathbf{chan}$	declare c to be a channel
$\textsc{Stop}$	deadlocked process
$\textsc{Skip}$	terminate immediately
$\textsc{Wait}\ t$	delay termination by t
$a \rightarrow P$	communicate a then do P
$a@t \rightarrow P$	communicate a at time t then do P
$c.a$	communicate a on channel c
$c?a$	input a on channel c
$c!a$	output a from channel c
$[b] \bullet P$	enable P only if b
$P;\ Q$	perform P till termination then Q

continued on next page

Notation	Explanation
$P \,\Box\, Q$	perform the first enabled of P and Q
$P \sqcap Q$	perform either of P and Q
$P \mid[A]\mid Q$	synchronize P and Q on events from A
$(\parallel p_1, \ldots, p_n \bullet \ldots;\ p_i \overset{A}{\longleftrightarrow} p_j;\ \ldots)$	network topology abstraction with parameters $p_1, \ldots, p_n$ and network connections including p_i communicating with p_j on private channels from A
$P \mid\mid\mid Q$	P and Q running without synchronization
$P \rhd \{t\}\ Q$	if P does not begin by time t, perform Q instead
$P \bigtriangledown \{t\} Q$	perform P until time t, then transfer control to Q
$P \bigtriangledown e \rightarrow Q$	perform P until exception e, then transfer control to Q
$P \bullet \textsc{Deadline}\, t$	P must terminate before time t
$P \bullet \textsc{WaitUntil}\, t$	after P idle until time t
$\textsc{Main}$	identifier of active class

A.2 Type Inference Rule

Expressions:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash n : \mathbb{Z}} \; [\; \mathit{NumExpr} \;]$$

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathbb{Z}} \; [\; \mathit{NaturalNumExpr} \;]$$

$$\frac{\Gamma \vdash x : T \in \Gamma}{\Gamma \vdash x : T} \; [\; \mathit{RefExpr} \;]$$

$$\frac{\Gamma \vdash E : \mathbb{P}\, T}{\Gamma \vdash \mathbb{P}\, E : \mathbb{P}(\mathbb{P}\, T)} \; [\; \mathit{PowerExpr} \;]$$

$$\frac{\Gamma \vdash E_1 : T_1, ..., \Gamma \vdash E_n : T_n,}{\Gamma \vdash \mathbb{P}\, E : T_1 \times ... \times T_n} \; [\; \mathit{TupleExpr}(n \geqslant 2) \;]$$

$$\frac{\Gamma \vdash E_1 : T, ..., \Gamma \vdash E_n : T,}{\Gamma \vdash \mathbb{P}\, E : \mathbb{P}\, T} \; [\; \mathit{SetExpr}(n \geqslant 0) \;]$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}\, T_1, ..., \Gamma \vdash E_n : \mathbb{P}\, T_n,}{\Gamma \vdash \mathbb{P}\, E : \mathbb{P}(T_1 \times ... \times T_n)} \; [\; \mathit{ProdExpr}(n \geqslant 2) \;]$$

$$\frac{\begin{array}{l}\Gamma \vdash x_1 : T_1;\; ...x_n : T_n \mid P, \\ \Gamma[x_1 \leftarrow T_1]...[x_b \leftarrow T_n] \vdash E : T\end{array}}{\Gamma \vdash \{S \bullet E\} : \mathbb{P}\, T} \; [\; \mathit{SetCompExpr}_1 \;]$$

$$\frac{\begin{array}{l}\Gamma \vdash x_1 : T_1;\; ...x_n : T_n, \\ \Gamma \vdash [x_1 \leftarrow T_1]...[x_n \leftarrow T_n] \vdash E : T\end{array}}{\Gamma \vdash \{S \bullet E\} : T_1 \times ... \times T_n} \; [\; \mathit{SetCompExpr}_2 \;]$$

$$\frac{\Gamma \vdash S : T_1 \times ... \times T_n, \quad \Gamma[x_1 \leftarrow T_1]...[x_b \leftarrow T_n] \vdash E : T}{\Gamma \vdash \lambda\, S \bullet E : \mathbb{P}(T_1 \times ... T_n \times T)} \text{ [} \mathit{LambdaExpr} \text{]}$$

$$\frac{\Gamma \vdash S : T_1 \times ... \times T_n, \quad \Gamma[x_1 \leftarrow T_1]...[x_b \leftarrow T_n] \vdash E : T}{\Gamma \vdash \mu\, S \bullet E : T} \text{ [} \mathit{MuExpr}_1 \text{]}$$

$$\frac{\Gamma \vdash S : T_1 \times ... \times T_n, \quad \Gamma[x_1 \leftarrow T_1]...[x_b \leftarrow T_n] \vdash E : T}{\Gamma \vdash \mu\, S \bullet E : T_1 \times ... T_n} \text{ [} \mathit{MuExpr}_2 \text{]}$$

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : T, ... \Gamma \vdash E_n : T}{\Gamma \vdash \langle E_1, E_2, ... E_n \rangle : \mathbb{P}(\mathbb{Z} \times T)} \text{ [} \mathit{SequenceExpr} \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{Z}, \Gamma \vdash E_2 : \mathbb{Z}}{\Gamma \vdash E_1 \mathit{InFun} E_2 : \mathbb{Z}} \text{ [} \mathit{OperExpr}(\mathit{InFun} : +, -, *, \mathit{div}, \mathit{mod}) \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{Z}, \Gamma \vdash E_2 : \mathbb{Z}}{\Gamma \vdash E_1 \mathit{InFun} E_2 : \mathbb{P}\,\mathbb{Z}} \text{ [} \mathit{OperExpr}(\mathit{InFun} : ..) \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}\, T, \Gamma \vdash E_2 : \mathbb{P}\, T}{\Gamma \vdash E_1 \mathit{InFun} E_2 : \mathbb{P}\, T} \text{ [} \mathit{OperExpr}(\mathit{InFun} : \cup, \int, \setminus) \text{]}$$

$$\frac{\Gamma \vdash E_1 : T_1, \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 \mapsto E_2 : T_1 \times T_2} \text{ [} \mathit{OperExpr}(\mathit{InFun} : \mapsto) \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}(\mathbb{Z} \times T) \quad \Gamma \vdash E_2 : \mathbb{P}(\mathbb{Z} \times T)}{\Gamma \vdash E_1 \frown E_2 : \mathbb{P}(\mathbb{Z} \times T)} \text{ [} \mathit{OperExpr}(\mathit{InFun} : \frown) \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}\, T_1 \quad \Gamma \vdash E_2 : \mathbb{P}(T_1 \times T_2)}{\Gamma \vdash E_1 \mathit{InFun} E_2 : \mathbb{P}(T_1 \times T_2)} \text{ [} \mathit{OperExpr}(\mathit{InFun} : \lhd, \ntriangleleft) \text{]}$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}(T_1 \times T_2) \quad \Gamma \vdash E_2 : \mathbb{P}\, T_2}{\Gamma \vdash E_1 InFun E_2 : \mathbb{P}(T_1 \times T_2)} [\ OperExpr(InFun : \rhd, \unrhd)\ ]$$

Predicate:

$$\frac{\Gamma \vdash P, \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} [\ AndPred\ ]$$

$$\frac{\Gamma \vdash P, \Gamma \vdash Q}{\Gamma \vdash P \vee Q} [\ OrPred\ ]$$

$$\frac{\Gamma \vdash P, \Gamma \vdash Q}{\Gamma \vdash P \Rightarrow Q} [\ ImpliesPred\ ]$$

$$\frac{\Gamma \vdash P, \Gamma \vdash Q}{\Gamma \vdash P \iff Q} [\ IffPred\ ]$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \neg P} [\ NegPred\ ]$$

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : \mathbb{P}\, T}{\Gamma \vdash E_1 \in E_2} [\ MemPred \in\ ]$$

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 = E_2} [\ MemPred =\ ]$$

$$\frac{\Gamma \vdash E_1 : \mathbb{Z}, \Gamma \vdash E_2 : \mathbb{Z}}{\Gamma \vdash E_1 InRel E_2} [\ RelationPred(InRel :<, \leqslant, >, \geqslant)\ ]$$

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \neq E_2} [\ RelationPred(InRel :\neq)\ ]$$

$$\frac{\Gamma \vdash E_1 : \mathbb{P}\, T, \Gamma \vdash E_2 : \mathbb{P}\, T}{\Gamma \vdash E_1 InRel E_2} [\ RelationPred(InRel :\subset, \subseteq)\ ]$$

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : \mathbb{P}\, T}{\Gamma \vdash E_1 \notin E_2} \; [\; RelationPred(InRel : \notin) \;]$$

$$\frac{\Gamma \vdash E : ObjectT}{\Gamma \vdash E.INIT} \; [\; PromotedInitPred \;]$$

$$\frac{\begin{array}{l}\Gamma \vdash S \\ \Gamma[x_1 \leftarrow T_1]...[x_n \leftarrow T_n] \vdash P\end{array}}{\Gamma \vdash \exists\, S \bullet P} \; [\; ExistsPred \;]$$

$$\frac{\begin{array}{l}\Gamma \vdash S \\ \Gamma[x_1 \leftarrow T_1]...[x_n \leftarrow T_n] \vdash P\end{array}}{\Gamma \vdash \forall\, S \bullet P} \; [\; ForallPred \;]$$

else:

$$\frac{\begin{array}{l}\Gamma \vdash E_1 : T_1;\; ...E_n : T_n \\ \Gamma[x_1 \leftarrow T_1]...[x_n \leftarrow T_n] \vdash P\end{array}}{\Gamma \vdash x_1 : E_1;\; ...x_n : E_n \mid P} \; [\; SchText_1 \;]$$

$$\frac{\Gamma \vdash E_1 : T_1;\; ...E_n : T_n}{\Gamma \vdash S} \; [\; SchText_2 \;]$$

A.3 Screenshots of HighSpec

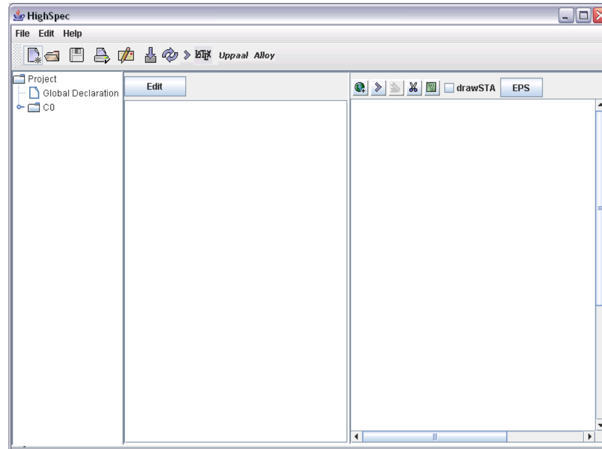


Figure A.1: The Main Window of **HighSpec**

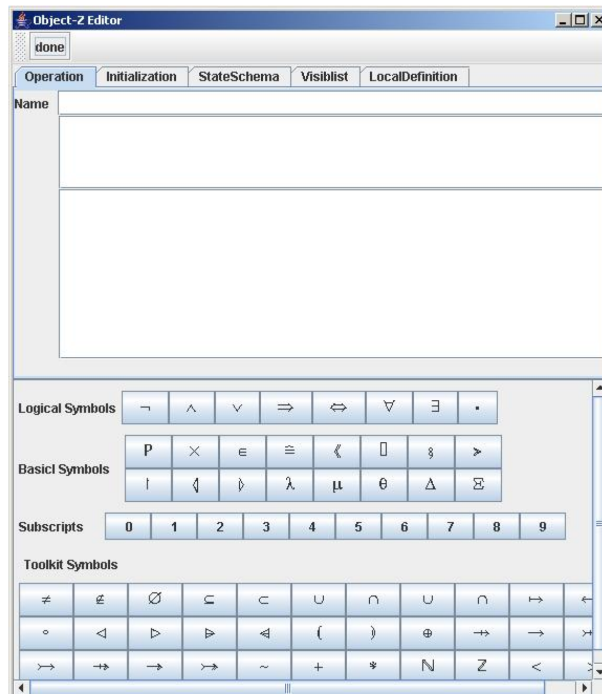


Figure A.2: The Object-Z Editing part

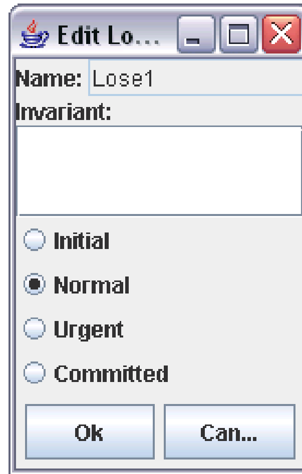


Figure A.3: The Timed Automaton Editing Part 1: state definition

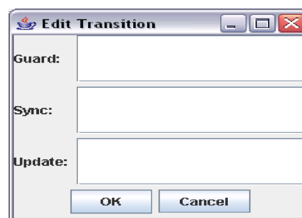


Figure A.4: The Timed Automaton Editing Part 2: transition definition

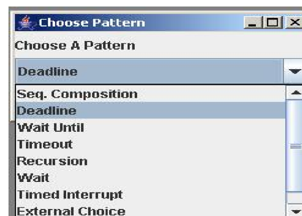


Figure A.5: The Timed Automaton Editing Part 3: pattern library

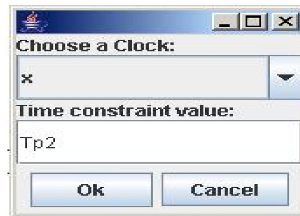


Figure A.6: The Timed Automaton Editing Part 4: timing parameter of patterns

Figure A.7: The Timed Automaton Editing Part 5: relating Object-Z operation with atomic states in the timed automaton

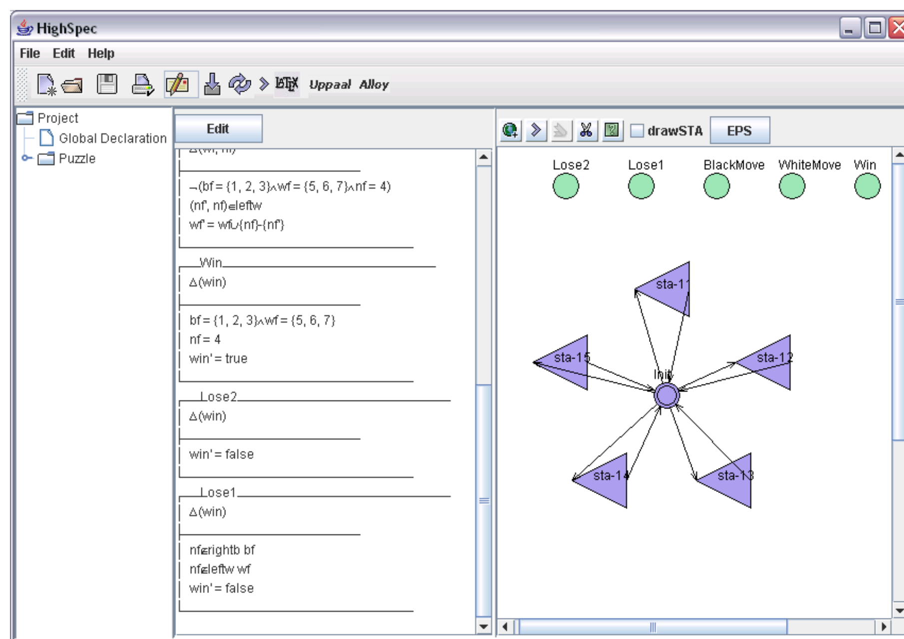


Figure A.8: The Model of Frog Puzzle Game Example: the default abstracted automaton with recursive pattern as the outmost layer and external choice as its inside layer

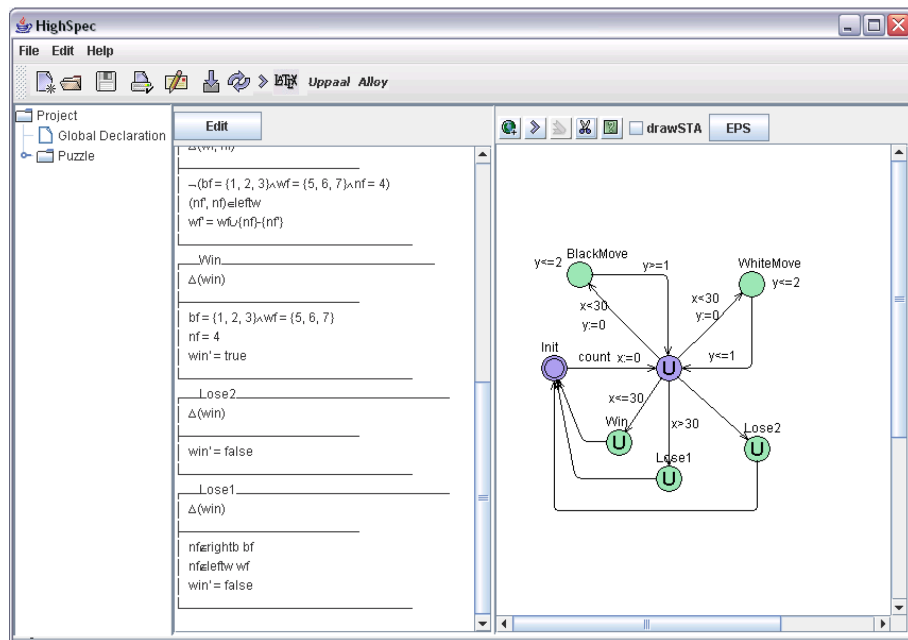


Figure A.9: The Model of Frog Puzzle Game Example