Timed Patterns: TCOZ to Timed Automata

J.S. Dong[1] P. Hao[*1] S.C. Qin[1] J. Sun[1] W. Yi[2]

[1] National University of Singapore
{dongjs,haoping,qinsc,sunj}@comp.nus.edu.sg
[2] Uppsala University, Sweden
yi@docs.uu.se

Abstract. The integrated logic-based modeling language, Timed Communicating Object Z (TCOZ), is well suited for presenting complete and coherent requirement models for complex real-time systems. However, the challenge is how to verify the TCOZ models with tool support, especially for analyzing timing properties. Specialized graph-based modeling technique, Timed Automata (TA), has powerful mechanisms for designing real-time models using multiple clocks and has well developed automatic tool support. One weakness of TA is the lack of high level composable graphical patterns to support systematic designs for complex systems. The investigation of possible links between TCOZ and TA may benefit both techniques. For TCOZ, TA's tool support can be reused to check timing properties. For TA, a set of composable graphical patterns can be defined based on the semantics of the TCOZ constructs, so that those patterns can be re-used in a generic way. This paper firstly defines the composable TA graphical patterns, and then presents sound transformation rules and a tool for projecting TCOZ specifications into TA. A case study of a railroad crossing system is demonstrated.

Keywords: Modeling and specification formalisms

1 Introduction

The specification of complex real-time systems requires powerful mechanisms for modeling state, concurrency and real-time behavior. Integrated formal methods (IFM) are well suited for presenting complete and coherent requirement models for complex systems. An important research agenda in IFM is the combination of Z/Object-Z [6] with CSP/TCSP [13] such as Timed Communicating Object Z (TCOZ) [9], Circus [16] and Object-Z + CSP [14]. However, the challenge is how to analyze and verify these models with tool support. We believe one effective approach is to project the integrated requirement models into multiple domains so that existing specialized tools in these corresponding domains can be utilized to perform the checking and analyzing tasks.

TCOZ is an integrated formal specification language which builds on the strengths of the Object-Z and TCSP notations for modeling both the state, process and timing aspects of complex systems. Rather than to develop a single tool support for TCOZ from scratch, we believe a better approach is to reuse existing tools. The specialized graph-based modeling technique, Timed Automata (TA) [1], is powerful in designing

---

[*] author for correspondence: haoping@comp.nus.edu.sg

real-time models with multiple clocks and has well developed automatic tool support i.e., KRONOS [4] and UPPAAL [2]. However, one weakness of TA is the lack of high level composable graphical patterns to support systematic designs for complex real-time systems. The investigation of possible links between TCOZ and TA may be beneficial to both techniques. For TCOZ, TA's tool support can be reused to check real-time constraints. For TA, a set of composable graphical patterns can be defined based on the semantics of the TCOZ constructs so that those patterns can be used as a generic framework for developing complex TA design models.

This paper is organized as follows. Section 2 introduces TCOZ and Timed Automata. Section 3 presents a set of composable TA patterns with their formal definitions (specified in Z). Section 4 presents the transformation rules with their correctness proof and a Java tool for projecting TCOZ (in XML format) to TA (also in XML of UPPAAL). Section 5 conducts a case study of a railroad crossing system. The last section gives the conclusion.

2 TCOZ and TA

2.1 TCOZ

TCOZ is essentially a blending of Object-Z with TCSP, 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 [12] 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 TCSP and Object-Z features may be found elsewhere [9]. The formal semantics of TCOZ (presented in Z) is also documented [10].
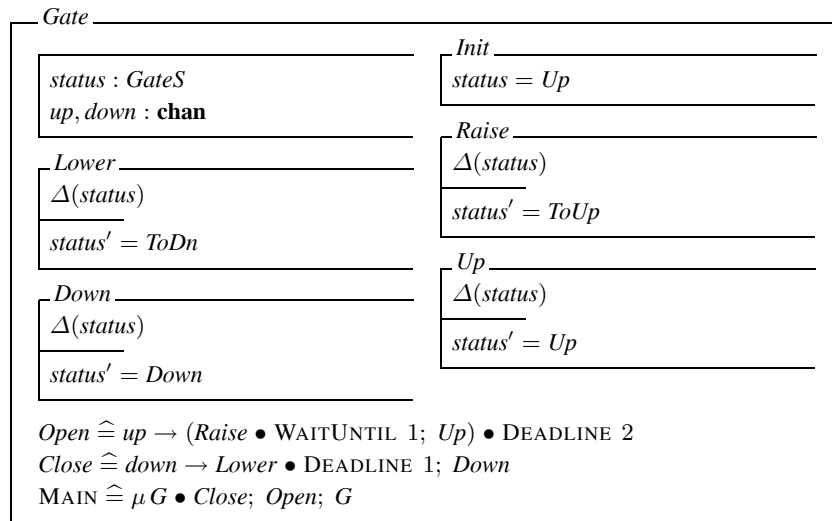
Timing and Channels: In TCOZ, all timing information is represented as real valued measurements. TCOZ adopts all TCSP timing operators, for instance, *timeout* and *wait*. In order to describe the timing requirements of operations and sequences of operations, a deadline command has been introduced. If *OP* is an operation specification (defined through any combination of CSP process primitives and Object-Z operation schemas) then $OP \bullet \text{DEADLINE}\, t$ 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 WAITUNTIL operator is a dual to the deadline operator. The process $OP \bullet \text{WAITUNTIL}\, t$ performs *OP*, but will not terminate until at least time t. In this paper, when the term TCOZ timing constructs is mentioned, it means TCSP constructs with extensions, i.e., DEADLINE and WAITUNTIL.

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 the 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 to TCOZ reduces the need to reference other classes in class definitions, thereby enhancing the modularity of system specifications.

Active Objects and Semantics: 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 behavior of the two objects can be represented as $ob_1 \;|||\; ob_2$, which means $ob_1.\text{MAIN} \;|||\; ob_2.\text{MAIN}$.

The details of the blended state/event process model forms the basis for the TCOZ denotational semantics [10]. In brief, the semantic approach identifies the notions of operation and process by providing a process interpretation of the Z operation schema construct. Operation schemas are modeled 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.

A Railroad Crossing Gate Example: The use of TCOZ is illustrated by a railroad crossing gate class as following (later a case study on this system will be conducted). The essential behaviors of this railroad crossing gate are to open and close itself according to its external commands (events) *up* and *down*. A free type *GateS* is used to capture the status of a gate:

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

┌─ *Gate* ───
│
│ ┌──────────────────────┐   ┌─ *Init* ───────────────┐
│ │ $status : GateS$ │   │ $status = Up$ │
│ │ $up, down : \textbf{chan}$ │   └────────────────────────┘
│ └──────────────────────┘   ┌─ *Raise* ──────────────┐
│ ┌─ *Lower* ────────────┐   │ $\Delta(status)$ │
│ │ $\Delta(status)$ │   ├────────────────────────┤
│ ├──────────────────────┤   │ $status' = ToUp$ │
│ │ $status' = ToDn$ │   └────────────────────────┘
│ └──────────────────────┘   ┌─ *Up* ─────────────────┐
│ ┌─ *Down* ─────────────┐   │ $\Delta(status)$ │
│ │ $\Delta(status)$ │   ├────────────────────────┤
│ ├──────────────────────┤   │ $status' = Up$ │
│ │ $status' = Down$ │   └────────────────────────┘
│ └──────────────────────┘
│
│ $Open \mathrel{\widehat{=}} up \rightarrow (Raise \bullet \text{WAITUNTIL } 1;\; Up) \bullet \text{DEADLINE } 2$
│ $Close \mathrel{\widehat{=}} down \rightarrow Lower \bullet \text{DEADLINE } 1;\; Down$
│ $\text{MAIN} \mathrel{\widehat{=}} \mu\, G \bullet Close;\; Open;\; G$
└──

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.

2.2 Timed Automata

Timed Automata [1, 3] 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. The set of clock constraints $\Phi(X)$ is defined by the following grammar:

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

A timed automaton A is a tuple $(S, S_0, \Sigma, X, I, E)$, where S is a finite set of states; S_0 is a set of initial states and a subset of S; Σ is a set of actions/events; X is a finite set of clocks; I is a mapping that labels each location s in S with some clock constraint in $\Phi(X)$; E, a subset of $S \times S \times \Sigma \times 2^X \times \Phi(X)$, is the set of switches. A switch $\langle s, s', a, \lambda, \delta \rangle$ 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 X that specifies when the switch is enabled.

For example, the railroad crossing gate can be designed in Figure 1. The gate is open in state *Up* and closed in state *Down*. It communicates with its controller through the events *up* and *down*. The states *ToUp* and *ToDown* denote the opening and the closing of the gate. The gate responds to the event *down* by closing within 1 time unit, and responds to the event *up* within 1 to 2 time units.
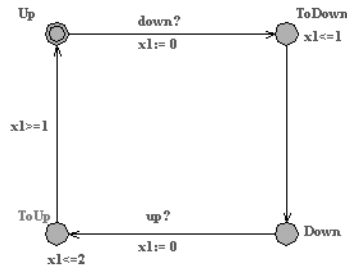

Fig. 1. The gate automaton

UPPAAL UPPAAL [2] is a tool for modeling, simulation and verification of real-time systems. It consists of three main parts: a system editor, a simulator and a model checker. The system editor provides a graphical interface of the tool, to allow easier maintenance. Its output is an XML representation of time automatons. The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design (or modeling) stages and thus provides an inexpensive mean of fault detection prior to verification by the model checker which covers the exhaustive dynamic behavior of the system. The model checker is to check invariant and bounded liveness properties by exploring the symbolic state space of a system. UPPAAL is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or

shared variables. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical.

3 Composable TA Patterns

High level real-time system requirements often need to state the system timing constraints in terms of *deadline*, *timeout*, *waituntil* and etc which can be regarded as common timing constraint patterns. For example, "task *A* must complete within *t* time period" is a typical one (*deadline*). TCOZ is a good candidate for specifying the requirements of complex real-time systems because it has the composable language constructs that directly capture those common timing patterns. On the other hand, if TA is considered to be used to capture real-time requirements, then one often need 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 towards 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 (one objective of this paper), but also TA can sysmatically capture high level requirements by utilizing those composable TA patterns.

Since the current semantics of TCOZ [10] is specified in Z, we define a set of composable TA patterns also in the same meta notation Z. First of all, we give the definition of TA in Z as follows.

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

$$\begin{array}{|l}
\hline
\quad \mathcal{S}_{TA} \\
\hline
S : \mathbb{P}\, State; \quad i, e : State \\
I : State \nrightarrow \Phi \\
T : \mathbb{P}\, Transition \\
\hline
i, e \in S \wedge \operatorname{dom} I = S \\
\forall s, s' : state;\ l : label \bullet (s, l, s') \in T \Rightarrow s, s' \in S \\
\hline
\end{array}$$

There are four basic types, i.e., $\mathbb{T}$, *State*, *Event*, and *Clock*, in which $\mathbb{T}$ is the set of positive real numbers; Φ defines the types of clock constraints, in which a *true* type is added to represent the empty clock constraints; *Label* models transition conditions, in which $\mathbb{P}\, Event$ is a set of enabling events, and $\mathbb{P}\, Clock$ gives a set of clocks to be reset, and Φ specifies clock constraints. $\mathcal{S}_{TA}$ defines a timed automaton, in which i and e represent its initial states and terminal states respectively; I defines local clock invariants on states; and T models transitions.

Some TA patterns together with their formal definitions in Z are presented in Figure 2 - Figure 5, the rest can be found in the technical report [5]. 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. For example, Figure 2 demonstrates how two sequential timed automatons A_1, A_2 can be composed together. By linking the terminal state of A_1 with the initial state of A_2, the resultant automaton passes control from A_1 to A_2 when A_1 goes to its terminal state. Figure 3 shows one of the common timing constraint patterns – *deadline*. There is a single 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 unit for every state of A. Thus the timing constraint expressed by this automaton is that A should terminate no later than t time units.
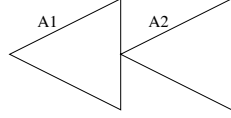

Fig. 2. Sequential Composition

$$seqcom : \mathcal{S}_{TA} \times \mathcal{S}_{TA} \to \mathcal{S}_{TA}$$

$$\forall A_1, A_2 : \mathcal{S}_{TA} \bullet$$
$$seqcom(A_1, A_2) = \langle\!|$$
$$S \; \widehat{=} \; A_1.S \cup A_2.S,$$
$$i \; \widehat{=} \; A_1.i, e \; \widehat{=} \; A_2.e, I \; \widehat{=} \; A_1.I \cup A_2.I,$$
$$T \; \widehat{=} \; A_1.T \cup A_2.T \cup \{(A_1.e, (\tau, \varnothing,$$
$$true), A_2.i)\} \; |\!\rangle$$

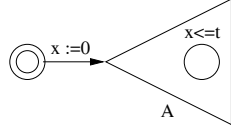**Fig. 3.** Deadline

$$deadline : \mathcal{S}_{TA} \times \mathbb{T} \to \mathcal{S}_{TA}$$

$$\forall A : \mathcal{S}_{TA}; \; t : \mathbb{T}; \; \exists x : Clock; \; i_0 : State \bullet$$
$$deadline(A, t) = \langle\!|$$
$$S \; \widehat{=} \; A.S \cup \{i_0\}, i \; \widehat{=} \; i_0, e \; \widehat{=} \; A.e,$$
$$I \; \widehat{=} \; \{s : A.S \bullet (s, x <= t \wedge A.I(s))\},$$
$$T \; \widehat{=} \; A.T \cup \{(i, (\tau, \{x\}, true), A.i)\} \; |\!\rangle$$

These 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 the timed systems. Furthermore, these patterns provide an interchange media for transforming TCOZ specifications into TA designs.

Composing TA Patterns

New patterns can be composed from the existing ones. For example, given a specification "Task A' is repeated every t_0 time units provided that A' is guaranteed to terminate
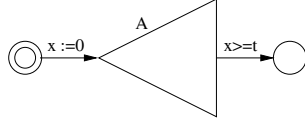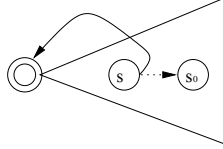
Fig. 4. Waituntil

$waituntil : \mathcal{S}_{TA} \times \mathbb{T} \to \mathcal{S}_{TA}$

$\forall A : \mathcal{S}_{TA};\ t : \mathbb{T};\ \exists x : Clock;\ i_0, e_0 : State \bullet$
$\quad waituntil(A, t) = \langle\!|$
$\quad S \mathrel{\widehat{=}} A.S \cup \{i_0, e_0\},$
$\quad i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, I \mathrel{\widehat{=}} A.I,$
$\quad T \mathrel{\widehat{=}} A.T \cup \{(A.e, (\tau, \varnothing, x >= t), e),$
$\qquad\qquad (i, (\tau, \{x\}, true), A.i)\} \ |\!\rangle$

Fig. 5. Recursion

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

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

before t_0 time unit", obviously, the TA model of this specification can be seen as a new pattern which can be composed by three existing patterns - *deadline*, *waituntil* and *recursion*, as shown in Figure 6, in which clock x is used to give time constraints for both the *deadline* pattern and the *waituntil* pattern, assuming A is the automaton equivalent to the TCOZ process A'.
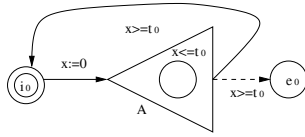

Fig. 6. Periodic Repeat

$PeriodicRepeat : \mathcal{S}_{TA} \times \mathbb{T} \to \mathcal{S}_{TA}$

$\forall A, A_0 : \mathcal{S}_{TA};\ t_0 : \mathbb{T};\ e_0 : State \mid e_0 = A_0.e$
$\quad \wedge A_0 = waituntil(deadline(A, t_0), t_0) \bullet$
$\quad PeriodicRepeat(A, t_0) = recursion(waituntil$
$\quad (deadline(A, t_0), t_0), e_0)$

According to the definition of *deadline*, *waituntil* and *recursion* patterns, the resultant automaton can be derived as follows:

$\mathcal{A}(P) = PeriodicRepeat(A, t_0) = \langle\!|$
$\quad S \mathrel{\widehat{=}} A.S \cup \{i_0, e_0\}, i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0,$
$\quad I \mathrel{\widehat{=}} \{s : A.S \bullet (s, x <= t_0 \wedge A.I(s))\},$
$\quad T \mathrel{\widehat{=}} \{(i, (\tau, \{x\}, true), A.i)\} \cup \{A.e, (\tau, \varnothing, x >= t_0), i)\} \cup A.T - \{A.e, (\tau, \varnothing,$
$\qquad x >= t_0), e_0)\} \ |\!\rangle$

4 Transformation Rules, Correctness & Tool

In this section, we will define a set of rules for mapping TCOZ to Timed Automata and provide the correctness proof for this transformation. A Java tool to automate the transformation process is implemented and illustrated.

4.1 Mapping TCOZ Processes into TA Patterns

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

Definition 1. *We define the mapping function $\mathcal{A}$ from TCOZ processes to TA as follows.*

- *If $P = $ SKIP, then $\mathcal{A}(P) = \langle\!\vert\ S \mathrel{\widehat{=}} \{i_0, e_0\}, i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, I \mathrel{\widehat{=}} \varnothing, T \mathrel{\widehat{=}} \{(i, (\tau, \varnothing, true), e)\}\ \vert\!\rangle$*
- *If $P = $ STOP, then $\mathcal{A}(P) = \langle\!\vert\ S \mathrel{\widehat{=}} \{i_0, e_0\}, i \mathrel{\widehat{=}} i_0, e \mathrel{\widehat{=}} e_0, I \mathrel{\widehat{=}} \varnothing, T \mathrel{\widehat{=}} \varnothing\ \vert\!\rangle$*
- *If $P = a@t \to P_0$, then $\mathcal{A}(P) = tprefix(a, t, \mathcal{A}(P_0))$*
- *If $P = P_0 \bullet $ DEADLINE t, then $\mathcal{A}(P) = deadline(\mathcal{A}(P_0), t)$*
- *If $P = P_0 \bullet $ WAITUNTIL t, then $\mathcal{A}(P) = waituntil(\mathcal{A}(P_0), t)$*
- *If $P = $ WAIT t, then $\mathcal{A}(P) = wait(t)$*
- *If $P = P_1 \rhd \{t\}P_2$, then $\mathcal{A}(P) = timeout(\mathcal{A}(P_1), \mathcal{A}(P_2), t)$*
- *If $P = P_1 \triangledown \{t\}P_2$, then $\mathcal{A}(P) = tinterrupt(\mathcal{A}(P_1), \mathcal{A}(P_2), t)$*
- *If $P = \mu N \bullet P(N)$, then $\mathcal{A}(P) = recursion(\mathcal{A}(P(N)), N)$*
- *If $P = P_1;\ P_2$, then $\mathcal{A}(P) = seqcom(\mathcal{A}(P_1), \mathcal{A}(P_2))$*
- *If $P = P_1 \sqcap P_2$, then $\mathcal{A}(P) = intchoice(\mathcal{A}(P_1), \mathcal{A}(P_2))$*
- *If $P = P_1 \square P_2$, then $\mathcal{A}(P) = extchoice(\mathcal{A}(P_1), \mathcal{A}(P_2))$*
- *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. Note that UPPAAL also adopts channels as its synchronization mechanism for the interaction between automatons, which is equivalent to the approach taken in TCOZ. 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 the DEADLINE. For example, the translation rule on the DEADLINE primitive, $P_0 \bullet$ DEADLINE t describes the process which has the same effect as P_0, but is constrained to terminate no later than t.

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 with time constraints in TCOZ specification can be completely translated into timed automata. The following provides the transformation rules for TCOZ classes/objects:

- In UPPAAL, every object is represented by an automaton. To fully represent behaviors of all the instances of a class, every instance (object) of a TCOZ class 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.
- Each operation schema in a TCOZ class is projected as an atomic state in its associated automaton instead of a triangle.

4.2 Correctness

This subsection is devoted to the soundness proofs for our mapping rules from TCOZ processes to structuralized Timed Automata. We shall prove that any source process in TCOZ and its corresponding target Timed Automaton preserve the same semantics under a bisimulation equivalence relation.

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

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

where $\mathcal{C} \,\widehat{=}\, \mathcal{P} \times \mathbb{T}$ is the set of configurations. 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. While $\longrightarrow_1 \subseteq (\mathcal{C} \times (\Sigma^\tau \cup \mathbb{T}) \times \mathcal{C})$ is the set of transitions. The operational rules are given in our technical report [5].

In order to derive observable behaviors of TCOZ processes, we define a new abstract transition system as follows:

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

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',

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

where the relation $\stackrel{\tau}{\longrightarrow}^*_1$ is the sequential composition of zero or finite number of $\stackrel{\tau}{\longrightarrow}_1$.

Now we construct an abstract transition system for our target formalism, Timed Automata. A "normal" transition system associated with timed automata ([1, 3]) can be

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

Notice that $\mathcal{S} \,\widehat{=}\, S \times V$ denotes all possible states of the transition system. 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. While the set $\longrightarrow_2 \subseteq \mathcal{S} \times (\Sigma^\tau \cup \mathbb{T}) \times \mathcal{S}$ comprises two kinds of transitions: a time passing move or an action move (Please refer to [5] for more details).

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

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

The only difference from TS^1_{TA} lies in the 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',

$$s \stackrel{a}{\Longrightarrow}_2 s' \,\widehat{=}\, \exists s_1, s_2 \cdot s \stackrel{\tau}{\longrightarrow}^*_2 s_1 \stackrel{a}{\longrightarrow}_2 s_2 \stackrel{\tau}{\longrightarrow}^*_2 s'$$
$$s \stackrel{\delta}{\Longrightarrow}_2 s' \,\widehat{=}\, \exists s_1, s_2 \cdot s \stackrel{\tau}{\longrightarrow}^*_2 s_1 \stackrel{\delta}{\longrightarrow}_2 s_2 \stackrel{\tau}{\longrightarrow}^*_2 s'$$

Now we define a bisimular relation between TS^2_{TCOZ} and TS^2_{TA} as below:

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

(1) $c \stackrel{\alpha}{\Longrightarrow}_1 c'$ implies there exists s' such that $s \stackrel{\alpha}{\Longrightarrow}_2 s'$, and $c' \approx s'$;
(2) $s \stackrel{\alpha}{\Longrightarrow}_2 s'$ 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 under the bisimulation relation.

Theorem 1 (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 By structural induction on process P.

- $P = \text{SKIP}$, or $P = \text{STOP}$. The proof is trivial.
- $P = \text{WAIT} \, t_0$. We know $\mathcal{A}(P) = wait(t_0)$. We show the condition (1) holds in Definition 2. The condition (2) can be demonstrated similarly. The process P can perform a time passing move (δ). 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 $\langle i, v_0 \rangle$ moves to their terminal states and preserve the bisimulation as well.
- Other cases are presented in the report [5] due to space constraints. □

4.3 Implementation

The translation process can be automated by employing XML/XSL technology. In our previous work [15], 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 an XML representation of Timed Automata, the automatic projection of the TCOZ model (in ZML) to TA model (in UPPAAL XML) can be developed as a tool in Java.

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 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 some states, their transitions 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 expression 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. 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.

5 Case Study : 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 [7] and used as a case study in many formal systems. It is a system which operates 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.

5.1 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 basic types for the status of the train and controller are defined as follows:

$TrainS ::= ToIn \mid In \mid Out$
$ControllerS ::= TrIn \mid TrOut \mid GtClose \mid GtOpen$

The TCOZ specification of *Gate* class has been presented in Section 2, the following provides the formal specification of *Train* and *Controller* class.

Train: The basic behavior of the train component is to communicate with controller with its passing information.
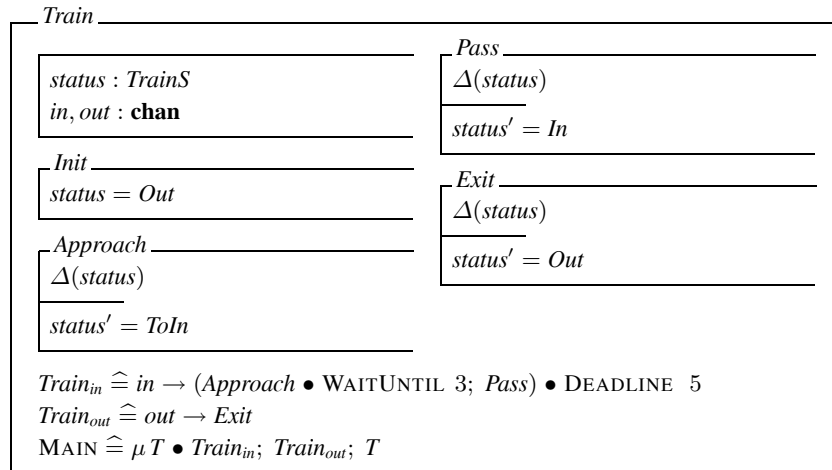
Central Controller: The central controller is the crucial part of the system, actively communicating with the train, light and gate. The *Controller* class is modeled as follows:

```
┌─ Controller ──────────────────────────────────────────────────────┐
│ ┌─────────────────────────────┐ ┌─ INIT ─────────────────────────┐ │
│ │ status : ControllerS        │ │ status = TrOut                 │ │
│ │ up, down, in, out : chan    │ │                                │ │
│ └─────────────────────────────┘ └────────────────────────────────┘ │
│ ┌─ Arrive ────────────────────┐ ┌─ GateClose ────────────────────┐ │
│ │ Δ(status)                   │ │ Δ(status)                      │ │
│ │ ─────────                   │ │ ─────────                      │ │
│ │ status′ = TrIn              │ │ status′ = GtClose              │ │
│ └─────────────────────────────┘ └────────────────────────────────┘ │
│ ┌─ Leave ─────────────────────┐ ┌─ GateOpen ─────────────────────┐ │
│ │ Δ(status)                   │ │ Δ(status)                      │ │
│ │ ─────────                   │ │ ─────────                      │ │
│ │ status′ = TrOut             │ │ status′ = GtOpen               │ │
│ └─────────────────────────────┘ └────────────────────────────────┘ │
│                                                                    │
│ $Train_{in} \mathrel{\widehat{=}} in \rightarrow Arrive;$   $Train_{out} \mathrel{\widehat{=}} out \rightarrow Leave$ │
│ $Close \mathrel{\widehat{=}} down \rightarrow GateClose;$   $Open \mathrel{\widehat{=}} up \rightarrow GateOpen$ │
│ MAIN $\mathrel{\widehat{=}} \mu\,C \bullet (Train_{in} \bullet$ DEADLINE 1) $\bullet$ WAITUNTIL 1; $Close; Train_{out} \bullet$ │
│       DEADLINE 1; $Open;\ C$                                        │
└────────────────────────────────────────────────────────────────────┘
```
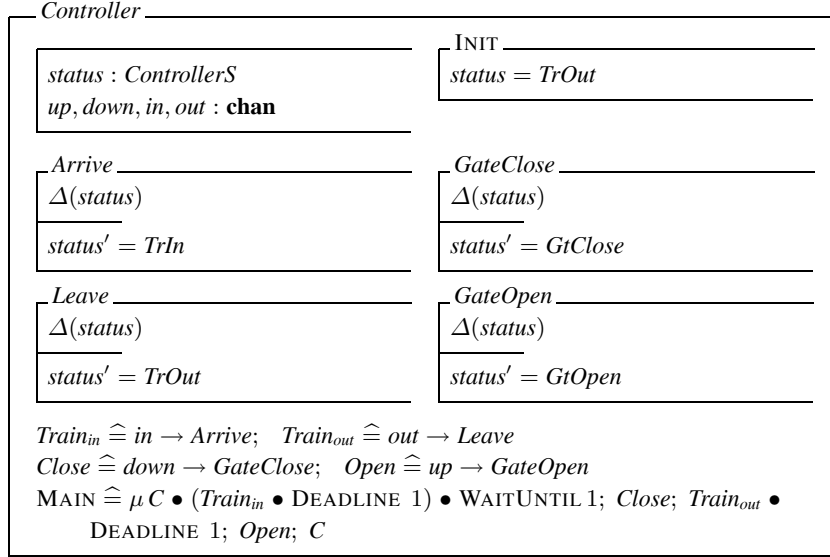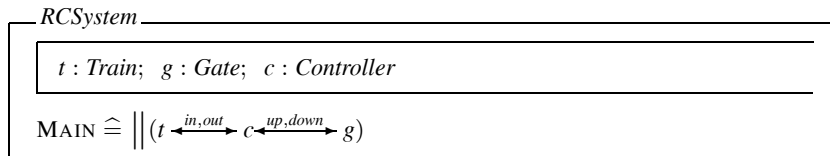
The attribute *status* keeps the records of the train's passing information in the system. When the train sends an *in* signal, the *status* of the controller changes from *TrOut* to *TrIn*. When the train has passed the crossing and sent an *out* signal to the controller, the *status* of the controller 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.

```
┌─ RCSystem ────────────────────────────────────────────────────────┐
│ ┌────────────────────────────────────────────────────────────────┐ │
│ │ t : Train;   g : Gate;   c : Controller                        │ │
│ └────────────────────────────────────────────────────────────────┘ │
│ MAIN $\mathrel{\widehat{=}} \big\|\,(t \xleftarrow{in,out} c \xleftarrow{up,down} g)$ │
└────────────────────────────────────────────────────────────────────┘
```

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:

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

5.2 Translation

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

First of all, for the whole RCS system, three automatons 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 *Gate* class, it has four operation schema, each one is mapped into a state, namely, *Up*, *ToDown*, *Down*, and *ToUp* as shown in Figure 1, 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* process of *Gate* class according to the translation rules on DEADLINE and WAITUNTIL primitives. A clock is generated to guard the atomic process *Lower* to be finished 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 simplified the resultant automaton.

This gate automaton can be automatically generated by our translation tool and visualized in UPPAAL as "process gate" in Figure 7. In the same way, we can get the train and controller automatons as "process train" and "process controller".
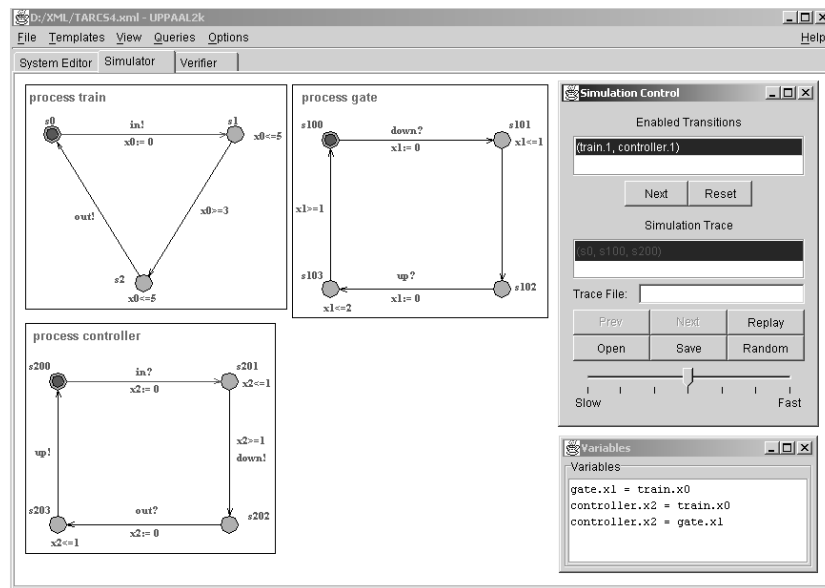

Fig. 7. Simulation

5.3 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 [8], for which UPPAAL implements model-checking algorithms.

From a safety critical perspective, the key point of the RCS is to provide guaranteed safety and efficient services. These properties can be formally interpreted from our model as:

- safety properties - The properties state that whenever the train is in, the gate is down. It can be translated into the TCTL formula in UPPAAL as follows:

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

- efficient service properties - 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.

6 Conclusion

TCOZ and TA lie at each end of the spectrum of formal modeling techniques. 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 tools support.

The investigation on the strengths and links between those two modeling techniques leads us to an interesting research result, i.e., timed composable patterns (reminiscence of 'design patterns' in object-oriented modeling). In this paper, these patterns are formally defined in Z and the process algebra-like compositional nature are 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 designing real-time systems in TA alone.

One possible future work would be to encode those timed patterns as icons in the model checker tool, such as UPPAAL, so that the complex timed models can be built systematically in UPPAAL.

Since TCOZ is a superset of TCSP, one consequence of this work is that a semantic link and a practical translation tool from TCSP to TA has been achieved so that TA tools i.e. UPPAAL can also be used to check TCSP timing properties. In this context, this work complements the recent pure theoretical investigation [11] on the expressiveness of TCSP and closed timed automata.

Acknowledgements

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. J. Bengtsson, K. G. Larsen, F. Larsson, and P. Pettersson avd Y. Wang. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control*, pages 232–243. Springer, 1996.
3. Albert M. K. Cheng. *Real-time systems : scheduling, analysis, and verification*. John Wiley and Sons, 2002.
4. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, pages 208–219. Springer, 1996.
5. J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. TCOZ to Timed Automata. Technical Report TRC6/03, School of Computing, National University of Singapore, 2003. http://nt-appn.comp.nus.edu.sg/fm/tcoz2ta/tr.zip.
6. R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
7. 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, Reai-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE Computer Society Press.
8. 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.
9. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
10. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.
11. J. Ouaknine and J. Worrell. Timed CSP = Closed Timed Automata. In *Proceedings of EXPRESS 02*, volume 38(2) of *ENTCS*, 2002.
12. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
13. 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. Huixing, 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.
14. G. Smith. An integration of real-time object-z and csp for specifying concurrent real-time systems. In M. Butler, L. Petre, and K. Sere, editors, *IFM 2002*, page 267C285. Springer-Verlag, 2002.
15. J. Sun, J. S. Dong, J. Liu, and H. Wang. A formal object approach to the design of zml. *Annals ol Software Engineering*, 13:329–356, 2002.
16. J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.