The Semantics and Tool Support of OZTA

J.S. Dong[1] P. Hao*[1] S.C. Qin[2] X. Zhang[1]

[1] National University of Singapore
{dongjs,haoping,zhangxi5}@comp.nus.edu.sg
[2] University of Durham, UK
shengchao.qin@durham.ac.uk

Abstract. In this work, we firstly enhance OZTA, a combination of Object-Z and Timed Automata, by introducing a set of timed patterns as language constructs that can specify the dynamic and timing features of complex real-time systems in a systematic way. Then we present the formal semantics in Unifying Theories of Programming for the enhanced OZTA. Furthermore, we develop an OZTA tool which can support editing, type-checking of OZTA models as well as projecting OZTA models into TA models so that we can utilize TA model checkers, e.g., Uppaal for verification.

Keywords: Timed Patterns, Semantics, Tool and Verification

1 Introduction

The specification of complex real-time systems requires powerful mechanisms for modelling state, concurrency and real-time behavior. Integrated formal methods are well suited for presenting complete and coherent requirement models for complex systems. This research area has been active for a number of years (e.g. [4, 3]) with a particular focus on integrating state based and event based formalisms (e.g. [9, 17]). However, the challenge is how to provide a systematical semantic model for the integrated formal languages, and how to analyze and verify these models with tool support? For the first issue, we believe UTP [13] is particularly well suited for giving formal semantics for the integrated specification languages and it has been used to define other integrated formalisms[12, 13]. For the second issue, 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.

OZTA [6] is an integrated formal language which builds on the strengths of Object-Z(OZ) [8, 16] and Timed Automata(TA) [1, 18] 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. One novel aspect of OZTA is its communication mechanism which supports partial and sometime synchronization [6].

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

The basic OZTA notation has been briefly described in an introductory paper [6] and this paper enhances the OZTA notation by extending its automaton part with time pattern structures. However the main purpose of this paper is to formalize the semantics of OZTA and present an OZTA tool we developed for its editing, type-checking and projection.
The rest of the paper is organized as follows, section 2 presents syntax of OZTA with extension of timed patterns; section 3 provides semantics of OZTA; section 4 shows the tool support of OZTA; and lastly section 5 gives the conclusion.

2 Extending OZTA with Timed Patterns

OZTA specifications are combinations of Object-Z schemas with Timed automatons. Timed Automata, with powerful mechanisms for designing real-time models using multiple clocks, has well developed automatic tool support. However, 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 close to design rather than specification. In our previous paper [7], we studied time automata 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 paper before presenting the semantics of OZTA, we firstly give a full version of the OZTA syntax, in which the automaton part of the the OZTA notation is extended with timed pattern structures. The enhanced specification of OZTA syntax with the notion of timed patterns is presented as follows:

$$
\begin{aligned}
&\mathit{Specification} ::= \mathit{CDecl};\ \ldots;\ \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{Inv}(x,n) \mid [(\mathit{Event})][(\mathit{Reset}(x))][(\mathit{Guard}(x,n))] \bullet \mathit{TA} \mid \mathit{Wait}\ (x,n)\\
&\qquad \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}\\
&\qquad \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} \mid \mathit{StaCtr}\\
&\mathit{Event} ::= \mathit{Event} \mid \mathit{Event}! \mid \mathit{Event}?\\
&\mathit{Reset} ::= (\_ := \_)\langle\langle \mathit{Clock} \times N \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\\
&\qquad \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}\\
&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
\end{aligned}
$$

in which, the argument x represents a certain clock, and n is a natural number; *StaCtr* represents a control state and *StaOp* is an operation state corresponding to an Object-Z operation; $\mathit{State} \bullet \mathit{Inv}(x,n)$ specifies a state with a local invariant; Event, $\mathit{Reset}(x)$,

$Guard(x, n)$ are transition labels, which respectively specifies event (Event! is an output event, Event? is an input event), clock reset and clock constraint; the three branches of S respectively represent the construct of handshaking synchronization, partial synchronization and sometime synchronization; the rest of the TA expressions are the timed automata patterns which can be directly utilized to construct timed automata.

2.1 The Pattern Structure

Each of the pattern expressions has a graphic presentation. Some TA patterns are presented in Figure 1 - 4, the rest can be found in [7]. 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 1 demonstrates how two timed automatons can

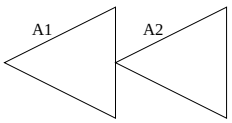


Fig. 1. Sequential Composition ';'

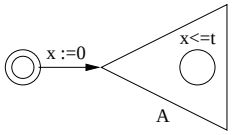


Fig. 2. Deadline 'Deadline(x, t)'

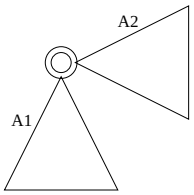


Fig. 3. External Choice '□'

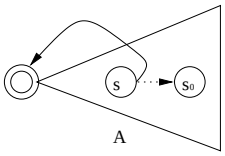


Fig. 4. Recursion'$u\ s_0 \bullet A(s_0)$'

be sequentially composed. 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 2 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. Figure 3 shows the external choice pattern of two timed automatons A_1 and A_2 which share an initial state, and the environment has the choice to trigger one of them by different external events. Figure 4 illustrates the *recursion* pattern of a timed automaton A, s_0 is the fixed point, The recursion is achieved by diverting all the transitions from pointing to s_0 to the initial state of A.

2.2 An Example: Frog Puzzle Game

A traditional frogs puzzle game is that: given seven stones, three white frogs at left facing right and three black frogs at 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 complex the puzzle, we add

some timing constraints to the moves of frogs, i.e., each frog takes at least 1 time units 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. The OZTA model of this frog puzzle is given as follow,

$Posn == 1..7$

Puzzle

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

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

INIT

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

BlackMove

$\Delta(bf, nf)$

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

$Lose_1$

$\Delta(win)$

$win' = \text{false}$

Win

$\Delta(win)$

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

WhiteMove

$\Delta(wf, nf)$

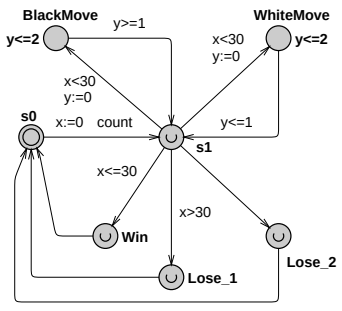
$\neg(bf = \{1,2,3\} \land wf = \{5,6,7\} \land nf = 4)$
$nf'\ \overline{leftw}\ nf$
$wf' = wf \cup \{nf\} - \{nf'\}$

$Lose_2$

$\Delta(win)$

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

x, y : **clock**



$$rightb : Posn \leftrightarrow Posn \qquad leftw : Posn \leftrightarrow Posn$$

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

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

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:

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

To illustrate the synchronization mechanism of OZTA, we consider several puzzle-solving systems:

The handshaking synchronization operator $\leftrightarrow$ indicates that the two switches labelled *count* in the objects of $p0$, $p1$ were identical, i.e., the automata must synchronize on these switches, as illustrated in Figure 5(1). The product of the two timed automata effectively ensures that the two puzzles start at same time point in the competition while operate independently and concurrently.

$PuzzleC_1$

$$p0, p1 : Puzzle$$

$$(p0 \parallel p1) \bullet \{p0.count \leftrightarrow p1.count\}$$

The partial synchronization operator $\rightarrow$ indicates that whenever the $p0.count$ is taken, then there must be synchronization with the switch $p1.count$. However, the switch $p1.count$ can occur independent of the switch $p0.count$. The partial synchronization between $p0$ and $p1$ is illustrated in Figure 5(2).

$PuzzleC_2$

$$p0, p1 : Puzzle$$

$$(p0 \parallel p1) \bullet \{p0.count \rightarrow p1.count\}$$

The sometime synchronization operator $\nleftrightarrow$ indicates that when any of the switches $p0.count$ or $p1.count$ is taken there may or may not be synchronization with the switch $p1.count$ or $p0.count$ respectively. The sometime synchronization between $p0$ and $p1$ is illustrated in Figure 5(3).

$PuzzleC_3$

$$p0, p1 : Puzzle$$

$$(p0 \parallel p1) \bullet \{p0.count \nleftrightarrow p1.count\}$$

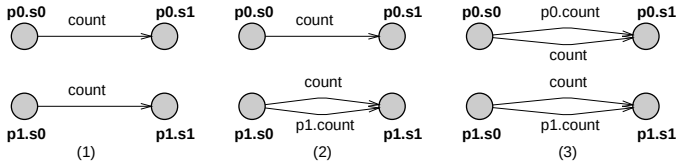


Fig. 5. Handshaking, Partial and Sometime Synchronization

3 The Semantics of OZTA

Before building the semantic model for OZTA, we need to choose an appropriate model of time. There are two typical time models: a discrete model and a continuous model. The current semantic model for OZTA [6] is a primitive operational semantics based on continuous time without pattern features. To make our model with the extension of timed patterns and 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 [14] on the semantics for time Circus [12] and Qin, Dong and Chin's work [13] on the semantics for TCOZ.

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 [13]. The key difference is that we now take consideration of 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 terminated or has reached an intermediate stable state. The automaton is *deadlock* when ok' is false.

– $wait$, $wait'$: *Boolean*. When $wait$ is true, it states that the automaton starts in an intermediate state. When $wait'$ is true, the automaton has not terminated; when it is false, it indicates a final observation.

– $state$, $state'$: $Var \rightarrow Value$. 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 automaton 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 tuple, where the first element of the tuple is the sequence of events that occurred during the time unit, and the second one is the associated set of refusals at the end of the same time unit. The set *Event* denotes all possible communicating events.

– *trace*: 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 and flat :

$$\text{seq}(\text{seq}(Event \times \mathbb{P}Event) \rightarrow \text{seq}\, Event$$
$$flat(\langle\rangle) \mathrel{\widehat{=}} \langle\rangle \qquad flat(\langle(es, ref)\rangle \frown tr) \mathrel{\widehat{=}} es \frown flat(tr)$$

– $cval, cval'$: $Clock \rightarrow N \cup \{NULL\}$. Among which *Clock* denotes all clock variables; N is the set of natural number; *NULL* is a number of no meaning, denoting 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*.

$$no\_interact(s) \mathrel{\widehat{=}} s = \langle\rangle$$

– The operator $\circ$ is the composition of two sequentially made observations. For two observation predicate $P(v, v')$, $Q(v, v')$, 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.
– The predicate $clock\_update(x, n)$ denotes that the value of clock variable x is updated to a natural number n.

$$clock\_update(x, n) \mathrel{\widehat{=}} cval' = cval \oplus \{x \mapsto n\}$$

3.2 The Semantics of Automata with Patterns

In this section, the observation model for OZTA automata is developed. We use *TA* to stand for the semantics of an automaton *TA* instead of the term $[\![TA]\!]$ in UTP. Before we go into the detail of the semantics for each Automata expressions, A healthiness condition **R** must be satisfied by the semantics predicate TA for any automaton, which is defined as,

$$\mathbf{R}(TA) \mathrel{\widehat{=}} TA = (TA \wedge tr \overset{t}{\preceq} tr')$$

$tr \overset{t}{\preceq} tr'$ states that, given two timed traces, tr and tr', tr' is an expansion of tr [13].

State and Control Operation

- State Operation
  $StaOp \mathrel{\widehat{=}} \Delta(b), a : T \bullet Pred(u, v') \mathrel{\widehat{=}} ok' \land \neg\ wait' \land no\_interact(trace) \land (\forall x : \mathrm{dom}\, cval \mid cval(x) \neq NULL \bullet clock\_update(x, \#tr' - \#tr)) \land ((\exists val_1 \bullet state' = state \oplus \{a \mapsto val_1\}) \circ (\exists val \bullet state' = state \oplus \{a \mapsto val\} \land Pred(state(u), state'(v'))))$
  In an operation state, time may progress, no event occurs, state will be updated. *NULL* means the clock has no value, and it has not been initialized yet.
- Control Operation
  $StaCtr \mathrel{\widehat{=}} ok' \land \neg\ wait' \land no\_interact(trace) \land (\forall x : \mathrm{dom}\, cval \mid cval(x) \neq NULL \bullet clock\_update(x, \#tr' - \#tr))$
  In a control state, time may progress, no event occurs and no state updates.
- Urgent state
  $StaU \mathrel{\widehat{=}} (StatOP \lor StaCtr) \land \#tr' = \#tr$
  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
  $StaI \mathrel{\widehat{=}} ok' \land \neg wait' \land tr = \langle\rangle \land no\_interact(trace) \land \forall x : \mathrm{dom}\, cval \bullet cval(x) = NULL)$
  The sequence of observations of an OZTA model starts from an initial state. The value of each clock variable is initially set to *NULL*.

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.
$State \bullet (x < n) \mathrel{\widehat{=}} x \in \mathrm{dom}\, cval \land (State \land (cval(x) + \#tr' - \#tr) < n \land (\forall c : \mathrm{dom}\, cval \mid cval(c) \neq NULL \bullet clock\_update(x, cval(c) + \#tr' - \#tr)) \lor Stop)$
$State \bullet (x <= n) \mathrel{\widehat{=}} x \in \mathrm{dom}\, cval \land (State \land (cval(x) + \#tr' - \#tr) \leq n \land (\forall c : \mathrm{dom}\, cval \mid cval(c) \neq NULL \bullet clock\_update(x, cval(c) + \#tr' - \#tr)) \lor Stop)$

Clock Reset $Reset(x) \mathrel{\widehat{=}} ok' \land \neg wait' \land \#tr' = \#tr \land state' = state \land x \in \mathrm{dom}\, cval \land clock\_update(x, 0)$
It can also be described in this way,
$Reset(x) \bullet TA \mathrel{\widehat{=}} Reset(x);\ TA$
Consecutive clock reset operations are combined into one atomic reset operation.

Event $Event \mathrel{\widehat{=}} ok' \land \neg wait' \land trace = \langle Event \rangle \land state' = state \land \#tr' = \#tr$
It can also be described in this way,
$Event \bullet TA \mathrel{\widehat{=}} Event;\ TA$

Clock Constraint An automaton can be guarded by clock constraints. The clock-guarded automaton $Guard(x, n) \bullet TA$ behaves as TA if the condition $Guard(x, n)$ is initially satisfied.
$Guard(x, n) \bullet TA \mathrel{\widehat{=}} (\exists x : Clock \bullet x \in \mathrm{dom}\, cval) \land (Guard(x, n) \land TA \lor \neg\ Guard(x, n) \land Stop)$
It enjoys the following properties:

- $\mathit{false} \bullet \mathit{TA} = \mathit{Stop}$
- $\mathit{true} \bullet \mathit{TA} = \mathit{TA}$
- $\mathit{Guard}(x,n) \bullet \mathit{Stop} = \mathit{Stop}$
- $\mathit{Guard}_1(x_1,n_1) \bullet (\mathit{Guard}_2(x_2,n_2) \bullet \mathit{TA}) = (\mathit{Guard}_1(x_1,n_1) \wedge \mathit{Guard}_2(x_2,n_2)) \bullet \mathit{TA}$
- $\mathit{Guard}(x,n) \bullet (\mathit{TA}_1;\ \mathit{TA}_2) = (\mathit{Guard}(x,n) \bullet \mathit{TA}_1);\ \bullet \mathit{TA}_2$

Wait The Wait construct specifies an automaton in which time idles for n time units then terminates.
$\mathit{Wait}(x,n) \mathrel{\widehat{=}} \mathit{ok}' \wedge \neg\ \mathit{wait}' \wedge \#tr' - \#tr = n \wedge (\forall i : \#tr' < i < \#tr \bullet \mathit{no\_interact}(\pi_1(tr'(i))))$
It is subjected to the following laws.

- $\textsc{Wait}\, n_1;\ \textsc{Wait}\, n_2 = \textsc{Wait}(n_1 + n_2)$
- $\textsc{Stop} \bullet \mathit{Timeout}(x,n) \bullet \mathit{TA} = \textsc{Wait}\, n;\ \mathit{TA}$

Deadline The Deadline construct $\mathit{TA} \bullet \mathit{Deadline}$ imposes a timing constraint on a timed automaton, which requires that TA should terminate no later than n time units.
$\mathit{TA} \bullet \mathit{Deadline}(x,n) \mathrel{\widehat{=}} (\mathit{ok} \wedge x \in \mathrm{dom}\, \mathit{cval} \wedge \mathit{clock\_update}(x,0)) \circ (\mathit{TA} \wedge \#tr' - \#tr \leq n)$

WaitUntil The WaitUntil construct $\mathit{TA} \bullet \mathit{WaitUntil}(x,n)$ constrains automation TA to finish its process no less than n time units.
$\mathit{TA} \bullet \mathit{WaitUntil}(x,n) \mathrel{\widehat{=}} (\mathit{TA} \wedge (\#tr' - \#tr \geq n)) \vee ((\exists tr_o \bullet tr \preceq tr_o \preceq tr' \wedge \#tr_o - \#tr < n) \wedge ((\mathit{ok} \wedge x \in \mathrm{dom}\, \mathit{cval} \wedge \mathit{clock\_update}(x,0)) \circ \mathit{TA}[tr_o/tr', \mathit{true}/\mathit{ok}', \mathit{false}/\mathit{wait}'] \circ \mathit{Wait}(x, n - (\#tr_o - \#tr))[tr_o/tr]))$

Timeout The Timeout construct $\mathit{TA}_1 \bullet \mathit{Timeout}(x,n) \bullet \mathit{TA}_2$ specifies that if no transition has been triggered for n time units in timed automaton TA_1, then TA_1 will be timeout and the control will be passed to TA_2.
$\mathit{TA}_1 \bullet \mathit{Timeout}(x,n) \bullet \mathit{TA}_2 \mathrel{\widehat{=}} (\mathit{ok} \wedge x \in \mathrm{dom}\, \mathit{cval} \wedge \mathit{clock\_update}(x,0)) \circ ((\mathit{TA}_1 \wedge \mathit{no\_interact}(\mathit{trace}) \wedge \#tr' - \#tr \leq n) \vee (\exists k : \#tr < k \leq tr + n, \exists tr_o \bullet \pi_1(tr'(k)) \neq \langle\rangle \wedge tr \preceq tr_o \wedge \#tr_o - \#tr = k \wedge (\forall i : \#tr < i < \#tr + k \bullet \mathit{no\_interact}(\pi_1(tr'(i))) \wedge tr_o(i) = tr'(i)) \wedge \mathit{TA}_1[tr_o/tr]) \vee (\exists tr_o \bullet tr \preceq tr_o \wedge \#tr_o - \#tr = n \wedge (\forall i : \#tr < i < \#tr + n \bullet \mathit{no\_interact}(\pi_1(tr'(i))) \wedge tr_o(i) = tr'(i)) \wedge \mathit{TA}_2[tr_o/tr]))$

Recursion We define the semantics of recursion same as [13],
$\mu X \bullet \mathit{TA}(X) \mathrel{\widehat{=}} \sqcap\{X \mid X \sqsupseteq \mathit{TA}(X)\}$, where X is the fixed point.

Parallel Composition The parallel composition of two automatons represents all the possible behaviors of both automatons 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 $TA_1 \ |[E]| \ TA_2 \bullet S$, where E denotes the set of events that TA_1 and TA_2 will communicate with, 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 UTP [10]:

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

Take note that 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 automatons. This is to allow each of the automatons 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,

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

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.

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

$s \underset{E\ S}{\|} t$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

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. Passive class does not have its own thread of control and its state and operations (processes) are available for use by its controlling object.
In order to address issues like class encapsulation and dynamic typing that are essential for object-orientation, a class model is established which is very similar with [13, 11] except that the TCSP operations are replaced with timed automatons. More detailed information on the semantics of class model can be referred to [13].

4 OZTA Tool

This section introduces the tool **OZTA** we developed for OZTA notation.
OZTA is a tool for modelling, type-checking and projecting complex real-time systems. It mainly consists of four components, i.e., a GUI editor, a type checker, a LaTeX code

generator and an Uppaal translator. The input language is based on the syntax and semantics we presented in the previous sections. The output of **OZTA** can either be an XML representation of OZTA models or LaTeX source files of OZTA models; **OZTA** can also generate projections of OZTA models which is ready to be taken as input for simulation and verification in Uppaal.
Figure 6 provides an overview of **OZTA**:

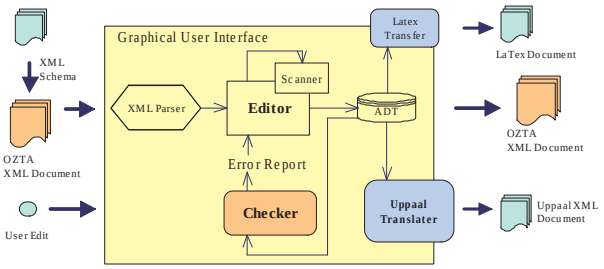


Fig. 6. Class Diagram of **OZTA**

4.1 GUI Editor with pattern support

The graphical editor has a main editing panel which consists of a schema editing part and a timed automaton editing part. Implemented with the timed patterns, the editor can support a more systematic design of timed automata. Automatons are generated in a top-down way. Firstly an abstracted default automaton $\mathcal{A}$ of an external choice pattern is automatically generated on the TA editing panel according to its established schema part of the model. Each branch of $\mathcal{A}$ is also an abstracted automaton and respectively represents one of the operation schemas defined on the schema editing panel. The designer can later embody these branches by recursively applying certain patterns until the behavior of the automaton meets its requirements.

4.2 Type Checker

The major functionalities of our **OZTA** type checker are to check syntax errors and to check static semantic errors in the OZTA specification. A full set of type checking rules can be found in our technical report [5].

4.3 LaTeX Code Generator

This generator outputs the LaTeX source file and EPS files for an OZTA model, which can be directly complied and viewed in LaTeX tools such as WinEdt.

4.4 Translator

An Uppaal translator is developed and integrated with **OZTA**. It extracts TA and state variables information from OZTA notation and generates an XML representation of Uppaal model for further embodiment and verification.

OZTA to Uppaal Uppaal is a useful integrated tool for modelling, simulation and verification of real-time systems. The simulation in Uppaal enables examination of possible dynamic executions of a system during early design (or modelling) 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. Its model checker is to check 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 description language of Uppaal is a timed automaton extended with a set of locally declared clocks, variables and constants. By projecting an OZTA model to a TA model, we can reuse Uppaal to simulate the dynamic behaviors the OZTA model and verify its various kinds of properties.

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 Uppaal model by regarding the OZTA operation states as abstracted automatons 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 automatons by adding the data information.

For example, in the frog puzzle game, we map the state variables bf, wf, nf of its OZTA model 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 in this way as shown in Figure 7.

Although our projection can handle most of the TA information of an OZTA model, one limitation needed to be pointed out is that, there is no verification tool yet which can support checking the properties related with the partial synchronization and sometime synchronization due to the novelty of this concept.

Model-Checking OZTA models 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 actually holds for this given model. Solutions of the puzzle can be visualized in Uppaal's simulator by running its diagnostics trace.

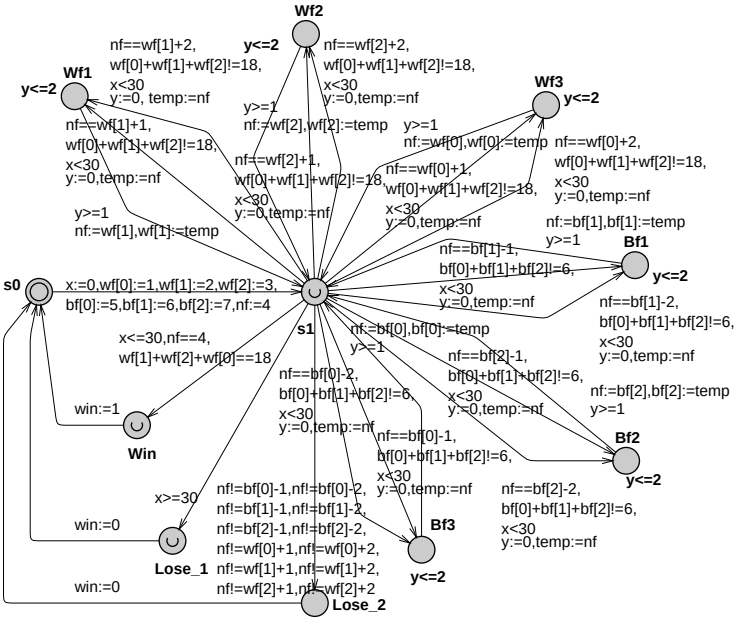


Fig. 7. Frog Puzzle Model in Uppaal

5 Conclusion

The contributions of the paper are listed as follows:

- We enhanced OZTA notation by introducing a set of timed patterns as language construct that can specify the dynamic and timing features of complex real-time systems in a systematic way.
- We presented a semantic model of OZTA in Unifying Theories of Programming which provides the semantic foundation for language understanding, reasoning and tool construction.
- We constructed an OZTA tool which can support 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 verification.

In our future work, we plan to further enhance our **OZTA** 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. We are also interested to study other projections, e.g., OZTA to Alloy, so that various properties of an OZTA model can be analyzed in the projected domains. Another future research work would be, based on our UTP semantics, to extend and link some proof systems [15] of Object-Z for reasoning about OZTA models.

Acknowledgement

We would like to thank Chen Qian, and He Kang for their part of work on the coding of the OZTA tool.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
3. E. Boiten, J. Derrick, and G. Smith, editors. *IFM'04: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, April 2004.
4. M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2002.
5. J. S. Dong, P. Hao, S. C. Qin, and X. Zhang. OZTA. Technical report TRC6/05, School of Computing, National University of Singapore, 2005. http://nt-appn.comp.nus.edu.sg/fm/ozta.
6. 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.
7. J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi. Timed Patterns: TCOZ to Timed Automata. In *The 6th IEEE International Conference on Formal Engineering Methods*, Seattle, USA, 2004.
8. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.
9. C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [2].
10. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
11. Z. Liu J. He and X. Li. A relational model for specification of object-oriented systems. Technical report 262, UNU/IIST, 2002.
12. A. Cavalcanti J. Woodcock. The Semantics of Circus. In *The 2th International Conference on Z and B*, LNCS 2272, pages 184–203. Springer-Verlag, 2002.
13. 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.
14. A. Sherif and J. He. Towards a Timed Model for Circus. In *The 2th IEEE International Conference on Formal Engineering Methods*, Shanghai, China, 2002.
15. G. Smith. Reasoning about Object-Z specifications. In *the Proceedings of Asia-Pacific Software Engineering Conference (APSEC '95)*, pages 794–804. IEEE Computer Society Press, 1995.
16. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
17. J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *2nd International Conference on Z and B*, volume 2272 of *Lect. Notes in Comput. Sci.*, pages 184–203. Springer-Verlag, 2002.
18. X.Nicollin, J.Sifakis, and S.Yovine. Compiling Real-time Specifications into Extended Autoamta. In *IEEE TSE Special Issue on Real-Time Systems*, volume 18(9), pages 794–804, 1999.