Timed automata patterns

Jin Song DONG

Ping HAO

Shengchao QIN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Wang YI

Timed Automata Patterns

Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi

Abstract—Timed Automata have proven to be useful for specification and verification of real-time systems. System design using Timed Automata relies on explicit manipulation of clock variables. A number of automated analyzers for Timed Automata have been developed. However, Timed Automata lack composable patterns for high-level system design. Specification languages like Timed Communicating Sequential Process (CSP) and Timed Communicating Object-Z (TCOZ) are well suited for presenting compositional models of complex real-time systems. In this work, we define a set of composable Timed Automata patterns based on hierarchical constructs in time-enriched process algebras. The patterns facilitate the hierarchical design of complex systems using Timed Automata. They also allow a systematic translation from Timed CSP/TCOZ models to Timed Automata so that analyzers for Timed Automata can be used to reason about TCOZ models. A prototype has been developed to support system design using Timed Automata patterns or, if given a TCOZ specification, to automate the translation from TCOZ to Timed Automata.

Index Terms—Timed Automata, timed patterns, TCOZ, UPPAAL.

1 Introduction

Specification and verification of real-time systems are important research topics that have practical implications. A popular approach for specifying real-time systems relies on the graphical notation Timed Automata [2], [35]. Timed Automata are powerful in designing real-time models with explicit clock variables. Real-time constraints are captured by explicitly setting/resetting clock variables. A number of mechanized verification support for Timed Automata have proven to be successful (e.g., UPPAAL, KRONOS, TEMPO, RED, and Timed COSPAN). However, designing and verifying real-time systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. High-level requirements for real-time systems are often stated in terms like *deadline*, *time-out*, and *wait until*, partly evidenced by the case studies presented in [31], [16], [13], [34], [19]. In industrial case studies of real-time system verification, system requirements are often structured into phases, which are then composed sequentially, in parallel, alternatively, etc. [25], [32]. Unlike Statechart or process algebras, Timed Automata lack high-level composable patterns (besides parallel composition) for hierarchical design. Timed Automata users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. The process is tedious and error prone.

Process algebras, on the other hand, are known for their bottom-up composibility, which is important to fight against the complexity of the system design. In the literature, a number of language-based process algebras have proven to be useful for specifying complex hierarchical real-time systems. Timed Communicating Sequential Processes (Timed CSPs [40]), which extend the classic CSP [27], support a rich set of compositional constructs to capture system requirements, e.g., time-out, timed interrupt, timed-event prefixing, etc. Together with compositional constructs like external/nondeterministic choice, recursion, and interrupt, Timed CSP offers a powerful mechanism for designing complex real-time systems. Timed Communicating Object-Z (TCOZ [36]) is an integrated high-level formal specification language that builds on the strengths of Timed CSP and combines it with Object-Z (OZ) [17] for modeling data and functional aspects of complex systems. In addition, new compositional constructs have been introduced, e.g., deadline and wait until. The downside of being expressive is that it is highly nontrivial to mechanically validate TCOZ models. For safety-critical systems, exposing a possible violation of system requirements at the specification level is very important, especially for hard timing constraints. The challenge of verifying time-enriched process algebras has long been recognized. A model checker named FDR [39] has been developed to verify CSP. However, there is no satisfactory verification support for either Timed CSP or TCOZ.

Based on the time-proven compositional constructs in Timed CSP/TCOZ, we develop a set of composable time patterns for Timed Automata which facilitates high-level system design using Timed Automata. The patterns cover a large class of common real-time behavior patterns and yet can be composed to form new useful patterns. The patterns are formally defined. By following the formal semantics (which are originated from their images in time-enriched process algebras), a high-level design composed of multiple timed patterns can be flattened to ordinary Timed Automata automatically. Thus, existing tools like UPPAAL can be used to verify time patterns with little computational overhead. On the other hand, the patterns allow a

- *J.S. Dong, P. Hao, and J. Sun are with the School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543, Republic of Singapore. E-mail: {dongjs, haoping, sunj}@comp.nus.edu.sg.*
- *S. Qin is with the Department of Computer Science, Durham University, Science Laboratories, South Road, Durham DH1 3LE, UK. E-mail: shengchao.qin@durham.ac.uk.*
- *W. Yi is with the School of Scince and Engineering, North Eastern University, P.R. China ans the Department of Information Technology, Uppsala University, Box 337,751 05, Uppsala, Sweden. E-mail: yi@it.uu.se.*

Manuscript received 10 Aug. 2007; revised 12 May 2008; accepted 19 June 2008; published online 23 July 2008.
Recommended for acceptance by R. Cleaveland.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-08-0239.
Digital Object Identifier no. 10.1109/TSE.2008.52.

systematic translation from Timed CSP/TCOZ models to Timed Automata so that the state-of-art verification mechanism for Timed Automata can be used to validate Timed CSP/TCOZ models.

This work identifies the strengths and weaknesses of Timed Automata and timed process algebras and develops new techniques for system modeling, as well as verification. Instead of comparing the expressiveness of the two notations (which is of theoretical interest and has been partially done in [38]), we focus on developing practical techniques that are beneficial to both TCOZ/Timed CSP and Timed Automata users. A closely related work is the hierarchical Timed Automata proposed in [11]. The notion of hierarchical Timed Automata in [11] is based the notion of *Statechart*. Two kinds of Timed Automata composition have been discussed, i.e., named and-states and or-states. In our work, the common time patterns introduced in Timed CSP and beyond have been formally defined, which serve as a library of building blocks for complex real-time systems. This work is also related to works on verification of time-enriched process algebra. In [15], Dong et al. have developed a model checker for Timed CSP based on constraint solving. In Brooke's PhD dissertation [8], a preliminary PVS encoding of Timed CSP was presented which relies heavily on user interaction. Another closely related area of research is Hoenicke and Olderog's work on the integration of CSP, OZ, and Duration Calculus (DC) (named CSP-OZ-DC [28]). By transforming the DC part of a system model into a Timed Automaton, CSP-OZ-DC benefits from Timed Automata's tool support. For instance, UPPAAL has been used to verify CSP-OZ-DC models. In our work, UPPAAL is used to verify TCOZ models. However, our main contribution is the development of the generic Timed Automata patterns, i.e., we not only borrow Timed Automata's tool support for TCOZ verification but also lend TCOZ's hierarchical constructs to facilitate systematic Timed Automata designs. Furthermore, the work in [28] mainly focuses on the smooth integration of the underlying semantic models of CSP, OZ, and DC and its use for verifying properties of CSP-OZ-DC specifications. Another related formalism for modeling real-time systems is TRIO [20]. This notation uses a notion of interface diagram that is different from the Timed-CSP-featured TCOZ notation in modeling dynamic behaviors. TRIO has been compared to TCOZ [36]. Similar to our work, some other real-time formalisms have been translated or projected to other formalisms for model-checking [29], [23]. This work is also related to works on compositional modeling of real-time systems [4], [3], [41], [22] and works on specification patterns in general [18], [30].

This paper is a revised and extended version of our conference paper [14], in which the link between Timed Automata and TCOZ has been investigated. A few timed patterns have been briefly introduced to show our initial ideas of reusing Timed Automata's tool support to verify TCOZ models. This paper substantially extends [14] with a range of common timed patterns, as well as a formal transition from TCOZ to Timed Automata and its correctness proof. We further enhance our previous work with illustrative examples, as well as a complex case study, which demonstrate the applicability of our approach. The remainder of the paper is organized as follows: Section 2 briefly introduces the relevant features of TCOZ and Timed Automata. Section 3 presents our main contribution, i.e., a set of composable Timed Automata patterns with formal semantics. We also demonstrate how new patterns can be composed from the existing ones. Section 4 shows how TCOZ or Timed CSP users may benefit from our patterns, by defining a formal translation from TCOZ to Timed Automata. A JAVA application for automating the projection is briefly discussed. Section 5 shows how Timed Automata users benefit from the patterns. We demonstrate a hierarchical Timed Automata design using a railcar system. Section 6 concludes this work. Throughout the paper, a process means a TCOZ or Timed CSP process and automata refer to Timed Automata unless otherwise stated.

2 BACKGROUND

2.1 Timed Automata

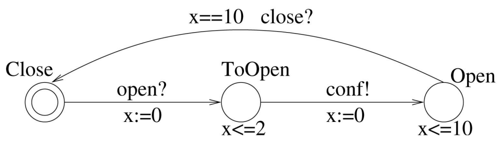
Timed Automata were proposed in [2] as an extension of the automata-theoretic approach to the modeling of real-time systems. Since then, the theory and verification tool support of Timed Automata has been an intensive field of research in computer science. A Timed Automaton is a finite automaton equipped with a finite set of clocks. Clocks are continuous real-valued functions of time that precisely record the time elapsed. All clocks advance at the same pace. Real-time system behaviors are captured using Timed Automata by explicitly resetting clocks and comparing a clock reading with constants.

Definition 1 (Timed Automaton). *A Timed Automaton A is a 7-tuple* $(S, init, F, \Sigma, C, Inv, T)$, *where S is a finite set of states,* $init \in S$ *is an initial state,*[1] $F \subseteq S$ *is a set of final states,* Σ *is a set of actions/events, C is a finite set of clocks,* Inv *is a proposition assignment function that, for each state, gives a proposition true at the state, and* $T : S \times \Sigma \times 2^C \times \Phi(C) \times S$ *is the transition relation. A transition* $(s, a, \lambda, \delta, s')$ *represents a transition from state s to state* s' *on event 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.* $\Phi(C)$ *is a set of clock constraints that is defined by the following grammar:* $\varphi := true | x \leq c | c \leq x | x < c | c < x | \varphi_1 \wedge \varphi_2$, *where x is a clock and c is a real number.*

Note that F might be empty if the Timed Automata never terminates (i.e., no deadlock state). The semantics of a Timed Automaton is defined as a transition system. A run of the timed automaton starts with the initial state. The control may remain at a state s as long as the state $Inv(s)$ is not violated. A transition $(s, a, \lambda, \delta, s')$ is enabled if and only if the control is at state s, the guard condition δ is satisfied (by the valuation of the clocks), and the event a is enabled. The transition is *unguarded* if δ is true. After taking the transition, the control moves to state s' and the clocks in λ reset.

Example. The following shows a sample Timed Automaton specifying a railcar door (the railcar system will be specified in Section 5).

1. If there were multiple initial states, a unique initial state is created with internal transitions to the ordinary initial states.



For the sake of readability, the states are labeled with names, e.g., *Open*, *Close*, and *ToOpen*. We assume that the door can be closed instantly and, hence, there is no state named *ToClose*. Initially, the control is at state *Close*, indicated by the double-lined circle. The transition is fired once an input is received over channel *open* (as the transition is unguarded), the control goes to state *ToOpen*, and clock x is reset. Within 2 seconds (constrained by the state invariant), the control goes to state *Open*. An output on the channel *conf* takes place along with the transition. The door remains open for 10 seconds, i.e., passengers have 10 time units for boarding, and, then, the control goes to state *Close*.

A Timed Automata specification may consist of a network of Timed Automata. A state in the product of two Timed Automata is a pair of states, each representing the progress one of the Timed Automata has made so far. A transition in the product is either a local transition of either automaton or a synchronization between the two components. The synchronization is *locking*; for an input and output pair, a component can input/output if and only if the other component can output/input. The following formally defines the parallel composition. For simplicity, we assume that there are no common clock variables. Note that parallel composition is the only Timed Automata pattern in the original proposal of Timed Automata [2].

Definition 2 (Parallel Composition). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1, 2\}$ be two Timed Automata. The parallel composition of A_1 and A_2 is*

$$para(A_1, A_2) = (S_1 \times S_2, (init_1, init_2), F_1 \times F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, Inv, T),$$

where $\forall (s_1, s_2) : S_1 \times S_2 \bullet Inv((s_1, s_2)) = Inv_1(s_1) \wedge Inv_2(s_2)$ and T is the smallest transition relation satisfying the following conditions:[2]

- *If $(s, a, \lambda, \delta, s') \in T_i \wedge a \notin \Sigma_{3-i}$ where $i \in \{1, 2\}$, then $((s, t), a, \lambda, \delta, (s', t)) \in T$.*
- *If $(s_1, a, \lambda, \delta, s_2) \in T_i$ and $(s'_1, a, \lambda', \delta', s'_2) \in T_{3-i}$, then $((s_1, s'_1), a, \lambda \cup \lambda', \delta \wedge \delta', (s_2, s'_2)) \in T$.*

The indexed parallel composition of multiple automata is written as $para(A_1, A_2, \ldots, A_n)$, where $A_1, \ldots, A_n$ are timed automata.

By definition, parallel composition is communicative and distributive tracewise. A number of Timed Automata verifiers based on the model checking technique have been developed. In this work, we focus on the popular UPPAAL [33]. UPPAAL is a tool for the 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 for the tool. Typically, a system description consists of a set of instances of Timed Automata declared from process templates and of some global data, such as global clocks, variables, and synchronization channels. The simulator is a validation tool that enables the examination of possible dynamic executions of a system. The model checker verifies invariants 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 restricted subset of Timed CTL [1] formulas for networks of Timed Automata. Note that Timed Automata in UPPAAL extend the original one [2] with handy constructs like urgent states and committed states. No time elapsing is allowed at an urgent state, i.e., an urgent state is labeled with an additional invariant that the control cannot reside at the state. We write $(s, urgent) \in Inv$ to mean that state s is urgent. In the following, states in the timed patterns may be marked as urgent states so as to make it easier to prove the soundness of the patterns and to simplify the flattened Timed Automata (see later examples). An urgent state is drawn as a circle with a "U" inside.

2. In UPPAAL, a pair of input and output results in a silent transition. Synchronization among multiple parties is mimicked using the notion of committed states.

2.2 Time Communicating Object Z

A number of time-enriched process algebras have been proposed. In this work, we focus on TCOZ [36], which, compared to others, contains a larger set of compositional constructs. TCOZ is designed to present a complete and coherent specification of systems with not only complicated control flows but also complex data and functional requirements. It is essentially a blending of OZ with Timed CSP, for the most part preserving them as proper sublanguages of the blended notation. TCOZ is novel in that it includes timing primitives, properly separates process control and data/algorithm issues, fully integrates notions of refinement from both languages, supports the modeling of true multithreaded concurrency, and distinguishes the notion of active and passive objects. In the following, we walk through the main features of TCOZ using an illustrative example. A detailed introduction to TCOZ and its Timed CSP and OZ features may be found in [36].

Example. Fig. 1 presents a TCOZ specification of a timed message queue. It contains a single class named *TimedQueue*. The anonymous schema called *state schema* identifies the data space of the class. In particular, the variable *item* is a sequence of messages. *MSG* is an uninterpreted data type. *in* and *out* of reserved type **chan** identifies the communication interfaces of this class, i.e., *in* and *out* are channels connecting objects of this class to its environment. Channels that share the same base name in different classes are connected implicitly. T_l, T_j, and T_o are time constants which are used to constrain the duration of a data operation. The schema named *INIT* contains the predicate that identifies the initial data state, i.e., *items* is empty. The two schemas *Add* and *Del* are *operation schemas.* They define the data operations that may update the valuation of the data variables. The *predicate part* of an operation (the part under the horizontal line) specifies an operation using a predicate composed of primed and unprimed versions of the data variables, as well as inputs/outputs. The primed variables denote the valuation of the variable after the operation. In particular, the predicate part of operation

TimedQueue

$items : \mathrm{seq}\, MSG;\ in, out : \mathbf{chan}$

$T_l, T_j, T_o : \mathbb{N}$

INIT

$items = \langle\,\rangle$

Add

$\Delta(items)$

$i? : MSG$

$items' = items \frown \langle i? \rangle$

Del

$\Delta(items)$

$i! : MSG$

$items \neq \langle\,\rangle \Rightarrow items = \langle i! \rangle \frown items'$

$items = \langle\,\rangle \Rightarrow items' = \langle\,\rangle$

$Join \mathrel{\widehat=} [i : MSG] \bullet in?i \rightarrow Add \bullet \textsc{Deadline}\ T_j$

$Leave \mathrel{\widehat=} [items \neq \langle\,\rangle] \bullet out!head(items) \rightarrow Del \bullet \textsc{Deadline}\ T_l$

$\textsc{Main} \mathrel{\widehat=} \mu Q \bullet (Join \,\square\, Leave) \rhd\{T_o\}\ (Del \bullet \textsc{Deadline}\ T_l);\ Q$

Fig. 1. Timed message queue specification.

Add means that the sequence of messages after the operation should be the original sequence of messages followed with the input. The remaining part is a set of process definitions which specifies the dynamic behavior of objects of this class. A process is defined by an equation. The left-hand side is the process name, e.g., $Join$, $Leave$, and $Main$. Note that $\widehat=$ means "is defined as." The right-hand side is the process expression that defines the process. For instance, process $Join$ behaves as follows: after getting an input i from channel in (the part $[i : MSG]$ is a guard condition saying that i must be of type MSG), the operation *Add* is performed within T_j time units (constrained by the DEADLINE expression). The $Leave$ operation specifies that if the sequence $items$ is not empty ($items \neq \emptyset$), the first element in $items$ ($head(items)$) shall be sent over channel out and, then, operation Del is performed to remove this element from $items$ in T_l time units. The process named MAIN identifies the behavior of instances of the class after initialization. In particular, the $Main$ process is defined as a recursion (as a μ function). The process construct $\square$ means choice and $\rhd\{T_o\}$ means *time-out*. The process reads as "the system may either behave as prescribed by $Join$ or $Leave$ or, if nothing happens in T_o time units, then the system performs operation Del in T_l time units and then repeats from the beginning." If no input or output is performed in T_o time units, then the first message is lost.

In this work, we focus on the process aspects of TCOZ, which borrows and extends the modeling power of Timed CSP. Table 1 shows a list of process constructs in TCOZ. STOP denotes a process that deadlocks and does nothing. SKIP denotes a process that terminates successfully. Process WAIT t delays the system for exactly t time units. Process $e \rightarrow P$ is initially willing to engage in event e and behaves as P afterward. A process $e@t \rightarrow P(t)$ behaves just like $e \rightarrow P$ except that the engage time of e is recorded in t. The event e can be a channel communication of the form $c!a$ or $c?a$. In order to capture timing requirements naturally, TCOZ supports common timing constraints like *deadline* and *wait until*. Process $P \bullet \textsc{Deadline}\ t$ is constrained to terminate within t time units. The WAITUNTIL operator is a dual to DEADLINE. Process $P \bullet \textsc{WaitUntil}\ t$ behaves as P but will not terminate before t time units elapse. If P terminates early, it idles until t time units elapse. Processes may be compositional. The sequential composition $P;Q$ behaves as P until it terminates and then behaves as Q. An external choice is written as $P \,\square\, Q$. Often, external choices are guarded by prefixing or conditionals. The internal choice $P \sqcap Q$ behaves as either P or Q nondeterministically. The parallel composition of two processes is written as $P|[E]|Q$, where E is a set of events. Events in E are synchronized by P and Q. If E is the empty set, the two processes interleave, written as $P|||Q$. Process $P \rhd \{t\} Q$ behaves as P if P makes a move before t time units elapse; otherwise, Q takes control. $P \bigtriangleup e \rightarrow Q$ behaves as P until event e is engaged and, then, P is interrupted and Q takes control. Process $P \bigtriangleup \{t\} Q$ is called *timed interrupt*. It behaves as P until t time units elapse and then switches to Q. Both interruptions are preemptive. Recursion is defined as a μ function. The semantics of recursion is defined as Tarski's weakest fixed point. It is known that unbounded recursion in CSP (and, therefore, Timed CSP and TCOZ) may result in irregular languages. Therefore, we restrict ourselves to tail recursion, i.e., a special case of recursion in which only the last operation of a process is a recursive call.

TABLE 1
Time-Enriched Process Constructs

Notation	Explanation
STOP	deadlock
SKIP	terminate immediately
Op	invocation of an operation schema
WAIT t	delay termination by t
$e \to P$	communicate a then do P
$e@t \to P(t)$	communicate a at time t then do P
$c?e \to P$	input a on channel c and then do P
$c!e \to P$	output a from channel c and then do P
$P \bullet$ DEADLINE t	P must terminate before time t
$P \bullet$ WAITUNTIL t	after P idle until time t
$P;\ Q$	perform P till termination then Q
$P \Box Q$	perform the first enabled of P and Q
$P \sqcap Q$	perform either of P and Q
$P \vert[E]\vert Q$	synchronize P and Q on events from E
$P \vert\vert\vert Q$	interleaving of P and Q
$P \rhd \{t\} Q$	if P does not begin by t, perform Q
$P \bigtriangleup e \to Q$	perform P until e, then perform Q
$P \bigtriangleup \{t\} Q$	perform P until time t, then perform Q
$\mu X \bullet P(X)$	recursion, X is a fixed point

3 Timed Automata Patterns

High-level real-time system requirements are often stated using terms like *deadline*, *time-out*, and *wait until*, which can be regarded as common timing constraint patterns. TCOZ, designed to capture high-level system requirements, has a rich set of hierarchical constructs that capture those common timing patterns. In contrast, Timed Automata users often need to manually cast those timing patterns into a set of clock variables with carefully calculated clock constraints. In this section, a rich set of composable Timed Automata patterns is defined formally in order to facilitate high-level system design using Timed Automata. Established compositional patterns introduced in the classic CCS, CSP, Timed CSP, and TCOZ are presented. The usefulness of the compositions is evidenced by early case studies on using those formalisms for system specification and analysis [31], [16], [13], [34]. Because we aim for efficient verification and compositional modeling, the patterns are designed to be simple, intuitive, and congruent to timed process algebra operators.

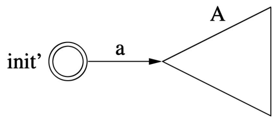
3.1 Patterns

In the following, the Timed Automata patterns are introduced one by one. First, the formal definition is presented, followed by an intuitive graphic presentation. Graphically, an automaton is abstracted as a triangle. The left vertex of the triangle or a circle attached to the left vertex represents the initial states. The right edge represents the final states. The parallel composition of two automata is represented as two triangles that are side by side. In the following, τ denotes an internal unobservable transition. For simplicity, a function/relation is interpreted as a set of tuples (as in the Z language [43]). We assume that, for any automaton $A = (S, init, F, \Sigma, C, Inv, T)$, the domain of Inv and T is always restricted to S. Note that not all of the patterns are essential as some of them can be generated by composing other patterns. We will thus introduce the fundamental ones and then a set of derived ones.

Definition 3 (Event Prefixing). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let a be an event. The event-prefixing pattern, i.e., the Timed Automaton in which e precedes any action of A, written as $eventprefix(a, A)$, is*

$$(S \cup \{init'\}, init', F, \Sigma \cup \{a\}, C, Inv \cup \{(init', true)\},$$
$$T \cup \{(init', a, \emptyset, true, init)\}),$$

where $init'$ is a fresh state.

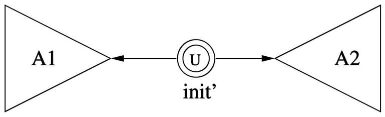


Event prefixing is considered one of the most commonly used basic patterns, e.g., a task is preceded by some event. In the above pattern, event a (which may be a synchronization barrier or variable assignment) must be engaged before a certain task (which is modeled as A) must be carried out. The fresh state $init'$ is connected to the initial state in A by a transition labeled with a.

Definition 4 (Internal Choice). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1, 2\}$ be two Timed Automata. The internal choice of A_1 and A_2, written as $intchoice(A_1, A_2)$, is*

$$(S_1 \cup S_2 \cup \{init'\}, init', F_1 \cup F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2,$$
$$Inv_1 \cup Inv_2 \cup \{(init', urgent)\}, T'),$$

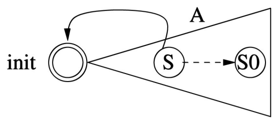
where $init'$ is a fresh state and $T' = T_1 \cup T_2 \cup \{(init', \tau, \emptyset, true,\ init_1), (init', \tau, \emptyset, true, init_2)\}$.



The above pattern captures nondeterminism, i.e., the system described nondeterministically behaves as either A_1 or A_2. Nondeterminism allows us to abstract away irrelevant details of the system [27]. By introducing a fresh state $init'$ and connecting $init'$ to initial states of both automata by unguarded transitions labeled with τ, the choice is made internally and, thus, nondeterministically. Because internal choice is symmetric and associative, we write $intchoice(A_1, A_2, \ldots, A_n)$ to denote the nondeterministic choice among multiple timed automata. Note that $init'$ is urgent and, thus, no time elapsing is allowed. The rule is that only additional states are marked urgent. Marking states urgent allows us to systematically simplify composed timed patterns (see example in Section 3.2).

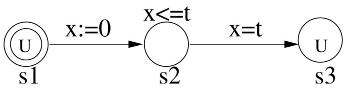
Definition 5 (Recursion). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let S_0 be a set of states such that $S_0 \subset S$.*

The recursion pattern, i.e., the Timed Automaton containing recursive invocation of A at a state in S_0, written as $recur(A, S_0)$, is $(S \setminus S_0, init, F \setminus S_0, \Sigma, C, Inv, T')$, where $T' = T \cup \{(s, a, \lambda, \delta, init) | \exists s_0 : S_0 \bullet (s, a, \lambda, \delta, s_0) \in T\}$.



Recursion is used to introduce infinite behaviors, which is commonly used for specifying nonterminating reactive systems. Given a timed automaton A and a set of states S_0 where there is a recursive call, the pattern is constructed by diverting all of the transitions to a state in S_0 to the initial state. Note that we request that S_0 must be a proper subset of S so as to prevent divergence. The dotted arrow represents a transition that originally leads to a state in S_0. In the resulting automaton, such transitions are redirected to the initial state of A. This construction only handles tail recursion. Note that we write $recur(A)$ to denote $recur(A, A.F)$, i.e., the second argument is defaulted to be the final states of A.

Definition 6 (Delay). *Let t be a positive real number. The Timed Automaton that delays the execution by t time units, written as delay(t), is $(\{s_1, s_2, s_3\}, s_1, \{s_3\}, \emptyset, \{x\}, Inv, T)$, where $Inv = \{(s_1, urgent), (s_2, x \leq t), (s_3, urgent)\}$ and $T = \{(s_1, \tau, \{x\}, true, s_2), (s_2, \tau, \emptyset, x = t, s_3)\}$.*

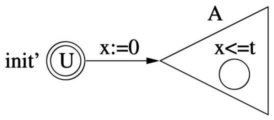


The above pattern is used to delay the execution (of the system) by exactly t time units, e.g., one of the common requirements for timed systems. We remark that internal choice $\sqcap$, event prefixing $a \rightarrow P$, recursion, and WAIT may be regarded as the fundamental blocking of Timed CSP processes. This is justified by combining previous results on deriving a normal form for CSP/Timed CSP processes. In [9], it is proven that all CSP processes can be transformed into a normal form, which composes only internal choices, event prefixing, and process referencing.[3] In [12], it is proven that the only fundamental building block of Timed CSP processes (besides those of CSP) is WAIT. We may thus prove that patterns that correspond to Timed CSP operators can be generated from the fundamental ones (by transforming them into the normal form) by combining the above results and proving a congruence theorem. In addition to patterns corresponding to Timed CSP constructs, two additional patterns are introduced, namely, *waituntil* and *deadline*. The *wait-until* pattern can be generated by composing other patterns (as we shall show later), whereas the *deadline* pattern cannot. Thus, we regard the following *deadline* pattern as fundamental too. Nonetheless, we give the definitions of patterns corresponding to all TCOZ operators for user convenience as well as efficiency reasons, i.e., a commonly used pattern may be manually optimized.

Definition 7 (deadline). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let t be a positive real number. The deadline pattern $deadline(A, t)$ is*

$$(S \cup \{init'\}, init', F, \Sigma, C \cup \{x\}, \{(init', urgent)\}$$
$$\cup \{(s, inv) | s \in S \wedge inv = (Inv(s) \wedge x \leq t)\},$$
$$T \cup \{(init', \tau, \{x\}, true, init)\}),$$

where $init'$ is a fresh state and x is a fresh clock.



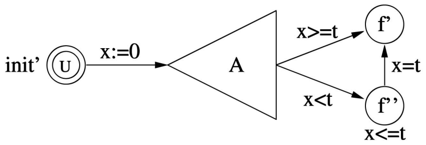
The above pattern is used to capture the requirement that some task must be finished by a certain time. It is more of a requirement than a design. Given a Timed Automaton A, if it is constrained to finish within t time units, all states in A are labeled with an invariant $x \leq t$ in which x is a fresh clock, which is reset when the control enters the automaton. The leftmost state is the fresh state $init'$. The local invariant $x \leq t$ covers each state of the timed automaton and, thus, A must terminate no later than t time units. This pattern may introduce timelocks [6]. In particular, there might be *time-actionlocks* (i.e., situations in which neither time nor action transitions can be performed) or *zeno-timelocks* (i.e., situations in which time is unable to pass beyond a certain point, but actions continue to be performed). Detecting and resolving timelocks is a highly nontrivial task (refer to [6] for sufficient conditions and sufficient and necessary conditions for timelock-freeness). Nonetheless, we choose to introduce this pattern simply because deadline is a common real-time requirement. We may verify (e.g., using the tool presented in [7] or UPPAAL) that A terminates within t time units in all circumstances (so as to prevent timelocks). Alternative ways of capturing deadlines are presented in [3], [26].

Definition 8 (wait until). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let t be a positive real number. The wait-until pattern, written as $waituntil(A, t)$, is*

$$(S \cup \{init', f', f''\}, init', \{f'\}, \Sigma, C \cup \{x\},$$
$$Inv \cup \{(init', urgent), (f', true), (f'', x \leq t)\}, T'),$$

where $init'$, f', and f'' are fresh states, x is a fresh clock, and

$$T' = T \cup \{(init', \tau, \{x\}, true, init)\}$$
$$\cup \{(f, \tau, \emptyset, x \geq t, f') | f \in F\} \cup \{(f, \tau, \emptyset, x < t, f'') | f \in F\}$$
$$\cup \{(f'', \tau, \emptyset, x = t, f')\}.$$



The above pattern is used to constrain that a certain task must take certain time units to finish. The automaton is constrained to finish its process no earlier than t time units. Given the automaton A, if the process of A finishes earlier than t time units, then the system idles at a fresh state f'' until exactly t time units elapses. Else if the process of A takes more than (or exactly) t time units, the system proceeds to final state f' without further waiting. Note that this pattern can be generated from the *delay* pattern and the parallel composition pattern, i.e., it is straightforward to prove that $waituntil(A, t)$ is equivalent to $para(A, delay(t))$.

3. Or a chaotic process $\perp$, which is irrelevant.

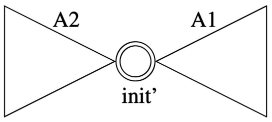
Definition 9 (External Choice). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1,2\}$ be two Timed Automata. An external choice of A_1 and A_2, written as $extchoice(A_1, A_2)$, is $para(A'_1, A'_2)$, where*

$$A'_1 = (S_1, init_1, F_1, \Sigma_1, C_1, Inv_1, \{(s, \tau, \lambda, \delta \wedge x \le 0, s') | (s, \tau, \lambda, \delta, s') \in T_1\} \cup \{(s, x := -1, \lambda, \delta \wedge x \le 0, s') | (s, a, \lambda, \delta, s') \in T_1\})$$

and

$$A'_2 = (S_2, init_2, F_2, \Sigma_2, C_2, Inv_2, \{(s, \tau, \lambda, \delta \wedge x \ge 0, s') | (s, \tau, \lambda, \delta, s') \in T_2\} \cup \{(s, x := 1, \lambda, \delta \wedge x \ge 0, s') | (s, a, \lambda, \delta, s') \in T_2\}),$$

where $x : \{-1, 0, 1\}$ is a fresh control variable.

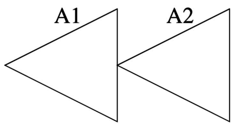


The above pattern is useful when the choice is resolved by the first action of either A_1 or A_2. In the composition, the system may initially take a transition enabled in either A_1 or A_2. An initial state in the composition is a pair of initial states $(init_1, init_2)$, which is labeled transitions from either $init_1$ or $init_2$. Given automata $A_1, \ldots, A_n$, the indexed external choice is written as $extchoice(A_1, \ldots, A_n)$ as $extchoice$ is symmetric and associative. As external choice is an operator of CSP, by applying the result in [9], it can be resolved to a normal form containing only the fundamental building blocks.

Definition 10 (Sequential Composition). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1,2\}$ be two Timed Automata. The sequential composition of A_1 and A_2, written as $seq(A_1, A_2)$, is*

$$(S_1 \cup S_2, init_1, F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, Inv_1 \cup Inv_2 \cup \{(s, urgent) | s \in F_1\}, T),$$

where $T = T_1 \cup T_2 \cup \{(f, \tau, \emptyset, true, init_2) | f \in F_1\}$.



Given two Timed Automata A_1 and A_2, the above shows the sequential composition. Sequential composition is commonly used to accomplish two tasks/jobs in order. By linking the final states of A_1 (i.e., the right edge) with the initial state of A_2 (i.e., the left vertex), the resulting automaton passes control from A_1 to A_2 immediately after A_1 terminates. The transition from a final state in A_1 to an initial state in A_2 is unguarded and labeled with τ. If A_1 is nonterminating, the final states in A_2 may not be reachable.

Definition 11 (Time Out). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1,2\}$ be two Timed Automata. Let t be a positive real number. The time-out pattern, written as $timeout(A_1, A_2, t)$, is*

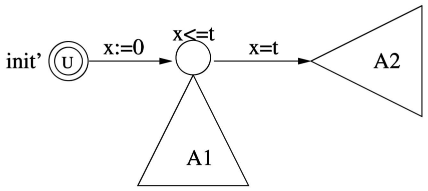
$$(S_1 \cup S_2 \cup \{init'\}, init', F_1 \cup F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2 \cup \{x\}, Inv', T'),$$

where $init'$ is a fresh state, x is a fresh clock,

$$Inv' = \{(init', urgent)\} \cup \{(init_1, Inv_1(init_1) \wedge x \le t)\} \cup \{(s, Inv_1(s)) | s \in S_1 \setminus \{init_1\}\} \cup \{(s, Inv_2(s)) | s \in S_2\}$$

and

$$T' = T_1 \cup T_2 \cup \{(init', \tau, \{x\}, true, init)\} \cup \{(init_1 \tau, \emptyset, x = t, init_2)\}.$$



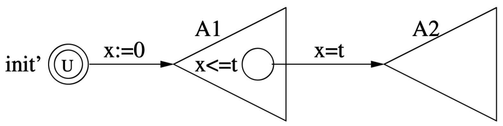
Time out is a common behavior pattern in real-time systems, which is partially evidenced by different ways proposed to model it [21], [5], [40]. The fresh clock x is reset along the transition from the fresh initial state to an initial in A_1. Each initial state in A_1 is constrained to make a move no later than t time units. If the control moves out the initial state before t time units, the system behaves as prescribed by A_1. Otherwise, after exactly t time units, A_2 takes over the control. If A_1 may make a move at exactly time t, the system nondeterministically takes one of the transitions and prevents the other from happening. The *timed-out* pattern can be generated using the *external choice* pattern and the *delay* pattern.

Definition 12 (Timed Interrupt). *Let $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ where $i \in \{1,2\}$ be two Timed Automata. Let t be a positive real number. The timed-interrupt pattern, i.e., the Timed Automaton in which A_1 is interrupted by A_2 after A_1 starts execution for t time units, written as $timeinter(A_1, A_2, t)$, is $(S_1 \cup S_2 \cup \{init'\}, init', F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2 \cup \{x\}, Inv', T')$, where $init'$ is a fresh state, x is a fresh clock,*

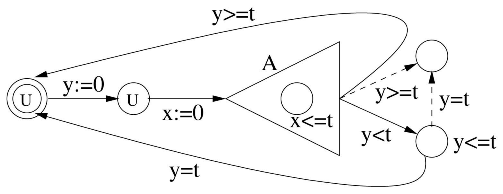
$$Inv' = \{(init', urgent)\} \cup \{(s, Inv_1(s) \wedge x \le t) | s \in S_1\} \cup Inv_2,$$

and

$$T' = T_1 \cup T_2 \cup \{(init', \tau, \{x\}, true, init_1)\} \cup \{(s_1, \tau, \emptyset, x = t, init_2) | s_1 \in S_1\}.$$



The pattern is composed of two automata, A_1 and A_2. The fresh state $init'$ and clock x are used similarly as in the previous patterns. Every state in A_1 is constrained to make a move before t time units. After t time units elapse, the control transfers from one of the state in A_1 (not necessarily a final state) to the initial state in A_2. Note that the interruption is preemptive, i.e., transitions in A_1 are prevented from happening once t time units have elapsed.



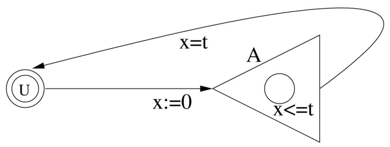
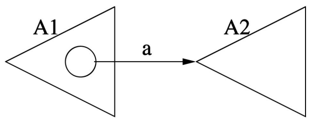


Fig. 2. Composing timed patterns.

Definition 13 (event interrupt). *Let* $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ *where* $i \in \{1, 2\}$ *be two Timed Automata. Let* a *be an event. The event-interrupt pattern, i.e., the Timed Automaton in which* A_1 *is interrupted whenever event* a *engages and then the control transfers to* A_2*, written as* $evtinter(A_1, A_2, a)$*, is*

$$(S_1 \cup S_2, init_1, F_1 \cup F_2, \Sigma_1 \cup \Sigma_2 \cup \{a\}, C_1 \cup C_2, Inv_1 \cup Inv_2, T'),$$

where $T' = T_1 \cup T_2 \cup \{(s_1, a, \{x\}, true, init_2) | s_1 \in S_1\}$.

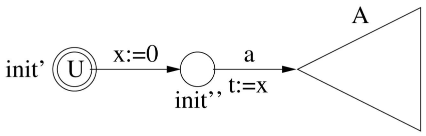


Event interrupt is similar to timed interrupt except that, because there are no timing constraints involved, no additional state or clock is necessary. Instead, an unguarded transition labeled with a is created from each state in A_1 to an initial state in A_2.

Definition 14 (Timed-Event Prefixing). *Let* $A = (S, init, F, \Sigma, C, Inv, T)$ *be a Timed Automaton. Let* a *be an event. Let* t *be a positive real number. The timed-event-prefixing pattern, i.e., the Timed Automaton in which* a *engages at time* t *and precedes any action of* A*, written as* $teventprefix(a, t, A)$*, is*

$$(S \cup \{init', init''\}, init', F, \Sigma \cup \{a\}, C \cup \{x\}, Inv \cup \{(init', urgent), (init'', true)\}, T'),$$

where $init', init''$ *are fresh states and* $T' = T \cup \{(init', \tau, \{x\}, true, init''), (init'', a; t := x, \emptyset, true, init)\}$).



Because t is the time taken from the moment this pattern is enabled to the time event a is engaged, a fresh clock x is initiated whenever the automaton is enabled. The reading of x at the time when a is engaged is recorded in t.

3.2 Composing Patterns

Our experience [16], [13], [37] shows that the patterns presented above cover most of the common timed patterns. Nonetheless, the patterns are by no means complete. New useful patterns can be added into our library or composed from the existing ones. In the following, we show how to build new patterns by composing multiple existing patterns.

Example. Assume that the requirement is that "a task must be repeated every t time units (the hidden requirement is that the task must terminate before t time units)." A new pattern, named *periodic-repeat* can be generated by composing the *deadline*, *wait-until*, and *recursion* patterns. Let $A = (S, init, F, \Sigma, C, Inv, T)$ be the automaton modeling the task. By Definition 7, $deadline(A, t)$ is

$$(S \cup \{i_1\}, i_1, F, \Sigma, C \cup \{x\}, \{(i_1, urgent)\} \cup \{(s, Inv(s) \wedge x \leq t) | s \in S\}, T \cup \{(i_1, \tau, \{x\}, true, init)\}).$$

By Definition 8,

$$waituntil(deadline(A, t)) = (S_1, init_1, F_1, C_1, Inv_1, T_1),$$

where

$$\begin{aligned} S_1 &= S \cup \{i_1, i_2, f_1, f_2\}; init_1 = i_2; F_1 = \{f_1\}; \\ &\Sigma_1 = \Sigma; C_1 = C \cup \{x, y\}, \\ Inv_1 &= \{(s, Inv(s) \wedge x \leq t) | s \in S\} \cup \{(i_1, urgent), (i_2, urgent), \\ &(f_1, true), (f_2, x \leq t)\}, \\ T_1 &= T \cup \{(i_1, \tau, \{x\}, true, init)\} \cup \{(i_2, \tau, \{y\}, true, i_1)\} \\ &\cup \{(f, \tau, \emptyset, y \geq t, f_1) | f \in F\} \cup \{(f, \tau, \emptyset, y < t, f_2) | f \in F\} \\ &\cup \{(f_2, \tau, \emptyset, y = t, f_1)\}. \end{aligned}$$

The *periodic-repeat* pattern is

$$periodicrepeat(A, t) = recur(waituntil(deadline(A, t))),$$

which is denoted as $(S_1, init_2, F_2, \Sigma_2, C_2, Inv_2, T_2)$. By Definition 5, we have

$$\begin{aligned} S_2 &= S \cup \{i_1, i_2, f_2\}; init_2 = i_2; F_2 = \emptyset; \Sigma_2 = \Sigma; \\ C_2 &= C \cup \{x, y\}, \\ Inv_2 &= \{(s, Inv(s) \wedge x \leq t) | s \in S\} \cup \{(i_1, urgent), \\ &(i_2, urgent), (f_2, x \leq t)\}, \\ T_2 &= T \cup \{(i_1, \tau, \{x\}, true, init\} \cup \{(i_2, \tau, \{y\}, true, i_1)\} \\ &\cup \{(f, \tau, \emptyset, y \geq t, i_2) | f \in F\} \cup \{(f, \tau, \emptyset, y < t, f_2) | f \in F\} \\ &\cup \{(f_2, \tau, \emptyset, y = t, i_2)\}. \end{aligned}$$

Note that, because the only final state f_1 is replaced by a recursive call at the last step, the automaton is nonterminating (as expected) and, hence, has no final states. The transformation is visualized in Fig. 2. The resulting automaton is the one on the left. Because state i_1 (the second state from left) is urgent, at any moment $x = y$, the automaton is simplified to be the right one (τ transitions have been removed). The two diverted transitions are merged into one because they are only enabled when the reading of clock x or y is t.

It is important that the Timed Automata patterns combine with the bottom-up composibility (of process algebra style) because complex real-time systems may be naturally modeled as collections of subcomponents at different abstraction levels. In the bottom-up design process, a subcomponent of reasonable complexity (and which is flattened) may be modeled as a Timed Automaton. The system can then be naturally composed from the modeling of the subcomponents. The Timed Automata patterns provide user-friendly templates to build such hierarchical modeling. In the top-down design process, the modeling of a complex system can be generated by refining the components, step by step, using appropriate patterns until it is simple enough to be modeled as a flattened Timed Automaton. As a reasonable price to pay, a system design based on Timed Automata patterns may require extra states or clocks, for instance, Definitions 7, 8, and 14 introduce new states and clocks. However, our experience shows that the extra states and clocks often can be removed using simple optimization procedures, e.g., τ-transition reduction (if the outgoing transitions of a state are all τ-transitions, remove the state and redirect all incoming transitions), urgent states reduction, or reusing clocks that are no longer referenced (which is common as clocks are local to one Timed Automaton).

4 Translating Timed Process Algebra to Timed Automata

A practical implication of the Timed Automata patterns is that process-algebra-based specification languages like Timed CSP or TCOZ can be systematically translated to Timed Automata so as to benefit from the verification mechanism of Timed Automata. In this section, a translation from TCOZ to Timed Automata is defined (which implies a translation from Timed CSP to Timed Automata). The following defines the Timed Automaton interpretation of the primitive processes.

Definition 15 (primitives). *Let Op be a data operation (or an operation schema):*

$$\begin{aligned} A_{skip} &= (\{i,f\}, i, \{f\}, \{\tau\}, \emptyset, \{(i, urgent), (f, true)\},\\ &\quad \{(i, \tau, \emptyset, true, f)\}),\\ A_{stop} &= (\{i\}, i, \emptyset, \emptyset, \emptyset, \{(i, true)\}, \emptyset),\\ A_{Op} &= \big(\{i,f\}, i, \{f\}, \{e_{Op}\}, \emptyset, \{(i, true), (f, true)\},\\ &\quad \big\{(i, e_{Op}, \emptyset, \mathrm{pre}\ Op, f)\big\}\big), \end{aligned}$$

where e_{Op} is an atomic event representing the data operation and pre Op is its precondition [43].

A_{skip} allows an unguarded transition from its initial state to the final state. Because the initial state is urgent, the automaton terminates immediately. A_{stop} allows no transitions at all (and so, no final state is reachable). A data operation is presented as a Timed Automaton with only one transition. The transition is guarded with the precondition of the operation and labeled with an event that represents the atomic data change. The control may reside at the initial state for some time as the data operation may not be instantaneous.

Definition 16 (translation). *Let $\mathcal{P}$ be the set of all TCOZ processes. Let $\mathcal{A}$ be the set of all Timed Automata. A translation is a function $M : \mathcal{P} \rightarrow \mathcal{A}$ defined as follows:*

$$\begin{array}{ll} M(\textsc{Skip}) & = A_{skip}\\ M(\textsc{Stop}) & = A_{stop},\\ M(Op) & = A_{Op},\\ M(a \rightarrow P) & = eventprefix(a, M(P)),\\ M(a@t \rightarrow P) & = teventprefix(a, t, M(P)),\\ M(\textsc{Wait}\ t) & = delay(t),\\ M(P \bullet \textsc{WaitUntil}\ t) & = waituntil(M(P), t),\\ M(P \bullet \textsc{Deadline}\ t) & = deadline(M(P), t),\\ M(P \triangleright \{t\} Q) & = timeout(M(P), M(Q), t),\\ M(P \triangle \{t\} Q) & = timeinter(M(P), M(Q), t),\\ M(P \triangle a \rightarrow Q) & = evtinter(M(P), M(Q), a),\\ M(P; Q) & = seq(M(P), M(Q)),\\ M(P \Box Q) & = extchoice(M(P), M(Q)),\\ M(P \sqcap Q) & = intchoice(M(P), M(Q)),\\ M(P \| Q) & = para(M(P), M(Q)),\\ M(\mu X \bullet P(X)) & = recur(M(P(X)), S_X), \end{array}$$

where P and Q are processes, a is an event, t is a positive real number, and S_X is a set of states where X is invoked.

Note that recursion in the above definition relies on S_X. A recursive process is translated like a normal process except that, whenever there is a recursive call X, instead of unfolding X as a Timed Automaton, a state is created and the state is added to S_X. The rest of the translation is mostly self-explanatory. For instance, $P \bullet \textsc{Deadline}\ t$ constrains that P must terminate no later than t. This process is translated to $deadline(M(P), t)$ in which $M(P)$ is the Timed Automaton capturing the behaviors of process P. The resulting automaton differs from $M(P)$ in that it must terminate before t time units, which is the semantics of the TCOZ expression $P \bullet \textsc{Deadline}\ t$. Other mapping rules can be explained similarly. By applying M iteratively, a process is translated to a flattened Timed Automaton straightforwardly.

Example. We use the timed queue example to illustrate the translation and hint how the timed patterns can be applied to facilitate system design using Timed Automata. If TCOZ is used to specify the queue in the first place, by applying M to the MAIN process, a Timed Automaton specification of the queue can be generated step by step. Because the generated Timed Automata are behaviorally equivalent to the TCOZ processes (shown below), the analysis results on the generated Timed Automata apply to the original TCOZ specification:

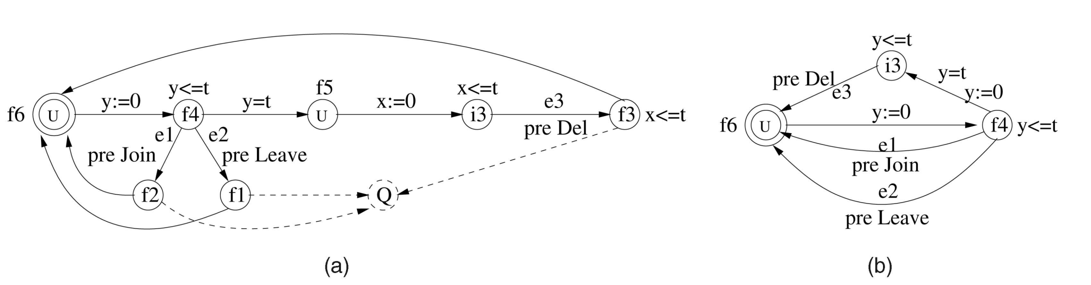


Fig. 3. Timed Automaton of a timed queue where $e1$, $e2$, and $e3$ are e_{Join}, e_{Leave}, and e_{Del}, respectively.

$$\begin{aligned}M(\text{MAIN}) &= M(\mu Q \bullet (Join \,\Box\, Leave) \rhd \{T_o\} \\ &\quad (Del \bullet \text{DEADLINE } T_l); Q) \\ &= recur(M(Join \,\Box\, Leave) \rhd \{T_o\} \\ &\quad (Del \bullet \text{DEADLINE } T_l); Q), S_Q) \\ &= recur(seq(M(Join \,\Box\, Leave) \rhd \{T_o\} \\ &\quad (Del \bullet \text{DEADLINE } T_l)), M(Q), S_Q) \\ &= recur(seq(timeout(M(Join \,\Box\, Leave), \\ &\quad M(Del \bullet \text{DEADLINE } T_l), T_o), M(Q)), S_Q) \\ &= recur(seq(timeout(extchoice(M(Join), \\ &\quad M(Leave)), deadline(M(Del), T_l), T_o), M(Q)), S_Q) \\ &= recur(seq(timeout(extchoice(A_{Join}, A_{Leave}), \\ &\quad deadline(A_{Del}, T_l), T_o), M(Q)), S_Q).\end{aligned}$$

As discussed above, a recursive call shall not be unfolded. For instance, $M(Q)$ in the above is viewed a Timed Automaton with only one state (which is both the initial and final state), which later will be removed because of the *recursion* pattern. By applying the definitions in Section 3.1 in a bottom-up manner, $M(\text{MAIN})$ is flattened to the one in Fig. 3b. Note that state Q corresponds to the recursive call of process Q. The Timed Automaton can be further reduced to Fig. 3a by removing τ-transitions (e.g., the transitions from $f2$, $f1$, and $f3$ to $f6$) and the urgent state $f5$ and reusing clocks (e.g., instead of x, reuse y).

Alternatively, Timed Automata users can do a top-down design using the patterns in the same modular way that TCOZ users do hierarchical system design, that is, to assume the availability of lower level processes or Timed Automata that behave in a certain way, specify the system in terms of the lower level processes or Timed Automata (e.g., at the top level, a designer may intuitively decide that the queue must be recursive and therefore apply the *recursion* pattern), and then iteratively refine each lower level process or Timed Automaton until they are simple enough to be modeled as a simple process or a flattened Timed Automaton.

In the following, we prove that the translation M is sound, i.e., given any process P, the automaton $M(P)$ is behaviorally equivalent to P. We show that there is a weak bisimulation relation (which implies behavioral equivalence) between the transition system semantics of a process and the automaton. This also justifies that the patterns capture intuitive meanings.

Definition 17 (TCOZ Semantics). *Let $\mathcal{P}$ be the set of all TCOZ processes. Let P be a process. Let $\mathbb{T}$ be the time domain. $TS_P^1 = (\mathcal{P} \times \mathbb{T}, (P,t), \Sigma \cup \mathbb{T}, \longrightarrow_1)$ is a labeled transition system where $\mathcal{P} \times \mathbb{T}$ is the state space, $(P,t) : \mathcal{P} \times \mathbb{T}$ is an initial state, Σ is a set of events, and $\longrightarrow_1$ is a labeled transition relation that is defined by the rules in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.52, where $c \xrightarrow{a}_1 c' \equiv (c,a,c') \in \longrightarrow_1$.*

Most of the rules (i.e., the operational semantics) can be referenced in Schneider's book [40], except those rules that are TCOZ-specific, e.g., DEADLINE and WAITUNTIL. In the next definition, τ-transitions in TS_P^1 are abstracted away because only observable behaviors are concerned.[4]

Definition 18 (Observable TCOZ Semantics). *Let $\mathcal{P}$ be the set of all TCOZ processes. Let P be a process. Let $\mathbb{T}$ be the time domain. $TS_P^2 = (\mathcal{P} \times \mathbb{T}, (P,t), \Sigma \cup \mathbb{T}, \Longrightarrow_1)$ is a labeled transition system where $\mathcal{P} \times \mathbb{T}$ is the state space, $(P,t) : \mathcal{P} \times \mathbb{T}$ is an initial state, Σ is the set of possible events including the internal event τ, and $\Longrightarrow_1$ is a labeled transition relation such that for any $c, c' : \mathcal{P} \times \mathbb{T}$,*

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

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

TS_P^2 differs from TS_P^1 in that all τ-transitions are hidden. Next, we define the semantics of Timed Automata. A transition system semantics of Timed Automata has been defined in [2], [10].

Definition 19 (Timed Automata semantics). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let V be the valuations of clocks. $TS_A^1 = (S \times V, (init, v_0), \Sigma \cup \mathbb{T}, \longrightarrow_2)$. $\widehat{=} S \times V$ is the state space. The initial state $s_0 = (init, v_0)$ comprises the initial state init and a zero valuation v_0. $\longrightarrow_2 \subseteq \mathcal{S} \times (\Sigma \cup \mathbb{T}) \times \mathcal{S}$ is a transition relation comprising either time passing $(s,v) \xrightarrow{\delta}_2 (s, v+\delta)$ or, if the transition $(s, a, \lambda, \phi, s')$ is enabled, an action execution $(s,v) \xrightarrow{\delta}_2 (s', v')$.*

Similarly, we define the transition system where all τ-transitions are hidden.

Definition 20 (Observable Timed Automata Semantics). *Let $A = (S, init, F, \Sigma, C, Inv, T)$ be a Timed Automaton. Let V be the valuation of the clocks. $TS_A^2 = (S \times V, (init, v_0), \Sigma \cup \mathbb{T}, \Longrightarrow_2)$. $S \times V$ is the state space. The initial state $s_0 = (init, v_0)$ comprises the initial state init and*

4. We assume that the processes are divergence free, i.e., only finite numbers of consecutive τ transitions are possible.

a zero valuation v_0. $\Longrightarrow_2 \subseteq \mathcal{S} \times (\Sigma \cup \mathbb{T}) \times \mathcal{S}$ *is a labeled transition relation such that, for all* $s, s' : S \times V$,

- $s \overset{a}{\Longrightarrow}_2 s' \mathrel{\widehat{=}} \exists s_1, s_2 \bullet s \overset{\tau}{\longrightarrow}{}^*_2\, s_1 \overset{a}{\Longrightarrow}_2 s_2 \overset{\tau}{\longrightarrow}{}^*_2\, s'$, *and*
- $s \overset{\delta}{\Longrightarrow}_2 s' \mathrel{\widehat{=}} \exists s_1, s_2 \bullet s \overset{\tau}{\longrightarrow}{}^*_2\, s_1 \overset{\delta}{\Longrightarrow}_2 s_2 \overset{\tau}{\longrightarrow}{}^*_2\, s'$,

where the transition relation $\overset{\tau}{\longrightarrow}{}^*_2$ *is the sequential composition of at most a finite number of* $\overset{\tau}{\longrightarrow}_2$.

With the above preparation, a bisimilar (homomorphic) relation between TS^2_P and TS^2_A is readily defined.

Definition 21 (Bisimulation). *Let* $TS_i = (S_i, init_i, \Sigma_i, T_i)$ *where* $i \in \{1, 2\}$ *be two labeled transition systems. For any* $s_1 \in S_1$ *and* $s_2 \in S_2$, $s_1 \approx s_2$ *if and only if*

- $\forall (s_1, a, s'_1) : T_1 \bullet \exists s'_2 \in S_2 \bullet (s_2, a, s'_2) \in T_2 \wedge s'_1 \approx s'_2$ *and*
- $\forall (s_2, a, s'_2) : T_2 \bullet \exists s'_1 : S_1 \bullet (s_1, a, s'_1) \in T_1 \wedge s'_1 \approx s'_2$.

TS_1 *and* TS_2 *are bisimilar, written as* $TS_1 \approx TS_2$, *if and only if* $init_1 \approx init_2$.

The following theorem states that the translation in Definition 16 is sound, i.e., the source and target transition systems are bisimilar. The proof is presented in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.52.

Theorem 1 (soundness). *Let* P *be a TCOZ process. Let* $TS^2_P = (\mathcal{P} \times \mathbb{T}, (P, t), \Sigma \cup \mathbb{T}, \Longrightarrow_1)$ *as defined above. Let* $A = M(P) = (S, init, F, \Sigma, C, Inv, T)$. *Let* $TS^2_A = (S \times V, (init, v_0), \Sigma \cup \mathbb{T}, \Longrightarrow_2)$ *as defined above.* $TS^2_P \approx TS^2_A$.

4.1 Implementation

In order to assist system development using either TCOZ or Timed Automata, we have developed a prototype supporting functionalities including automated translation from TCOZ specification to Timed Automata (for TCOZ users to benefit from the supporting tools of Timed Automata) and high-level system design using Timed Automata (refer to the next section).

The translation from TCOZ to Timed Automata is automated by employing XML and Java technology. In our previous work, the syntax of the Z family languages (i.e., Z/OZ/TCOZ) has been defined using XML schemas, which is named ZML [42]. A number of tools based on XML representation of TCOZ models have been developed. UPPAAL also supports an XML representation of Timed Automata. Hence, the translation is reduced to a conversion from one XML file to another. Our tool reads a TCOZ specification represented in XML and outputs an XML representation of a Timed Automaton specification conforming to the DTD syntax defined in UPPAAL. The translation rules are used as a design document to transform the process aspects of a TCOZ specification. Worthy of mention is that, because TCOZ allows synchronization among multiple parties, whereas UPPAAL allows only pairwise synchronization, committed states are used to achieve broadcasting communication in the target UPPAAL model (detailed information can be found in [33]). In addition, the following handles the data aspects. A channel communication may carry a value in TCOZ, which is not possible in UPPAAL. Value passing in UPPAAL is achieved by global variable assignments along the Timed Automata transitions. A class in TCOZ is translated to a Timed Automaton template in UPPAAL (by translating the MAIN process of the class). Instances of the class are translated as instances of the template. The predicate in the *INIT* schema of a TCOZ class is labeled with the initial state of the resultant Timed Automaton. UPPAAL supports limited data types, e.g., bounded integers, arrays of bounded integers, record types, scalars, etc. Thus, only those data types are allowed for the translation at the current stage.

5 Case Study: Railcar System

The system design based on TCOZ is to identify objects and their data attributes from the system requirements, encapsulate relevant data operations, as well as dynamic behavior patterns, in the respective object modeling, and, finally, compose different objects to form the system specification. Because of the bottom-up composibility of process algebra (which TCOZ is based on), modeling the dynamic behaviors in TCOZ is compositional and natural. The system design based on Timed Automata patterns enjoys similar composibility, i.e., starting with building Timed Automata designs of fractions of the system and then composing the basic Timed Automata using the timed patterns. In this section, a Railcar System is used to demonstrate a high-level real-time system design using both Timed Automata patterns and TCOZ. This system was inspired by the railcar system presented in [24].

5.1 High-Level System Design Using Timed Automata

In the Railcar System, there are four (possibly more) terminals that 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 terminals. The railcar is equipped with a destination board to receive internal requests and to indicate each destination terminal. There is a central control that receives, processes, and sends data to the terminals and the railcar so that external requests from any terminal can be fulfilled. Additional timing requirements are added into the original model [24] because quantitative timing is an important aspect of the system, as illustrated in the following scenarios:

- Before the railcar leaves a terminal, it sends a signal to depart to the terminal. The terminal prepares for the railcar to depart and responds to the railcar within 5 seconds. The railcar then leaves the terminal and cruises toward the destination.
- When the railcar comes to a stop at a terminal, it opens its door for exactly 10 seconds. After that, it closes the door and begins to wait for either internal passenger 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.

Designing the system from scratch using ordinary Timed Automata is nontrivial. The system is composed of multiple objects, each of which are assigned with different behavior patterns, e.g., the controller is responsible for handling

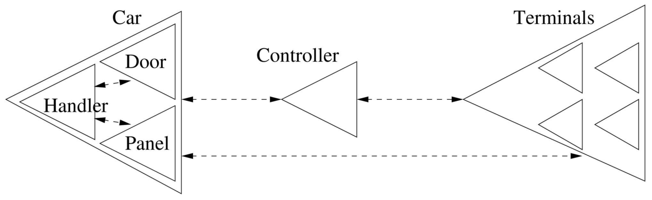


Fig. 4. A structural design.

different tasks. Nonetheless, the requirements of the system are naturally captured in a hierarchical way [24]. Using the timed patterns, we achieve a similar approach to model the system as in [24]. The Railcar System has three components, i.e., a railcar, the terminals, and a central controller. The railcar is composed of three basic components, i.e., a car destination panel, a car door, and a car handler. There is a button at each terminal so that passengers can request for a service. The central controller maintains a record of all external requests and dispatches external requests to the railcar if there is any. All system components are connected by channels. Fig. 4 shows a hierarchical blueprint of the system, where a triangle represents a Timed Automaton and a double-arrowed dotted line means that there are communications in between. Each triangle will be detailed in the following. For each component, we will first identify the basic behaviors (of manageable complexity) and then compose them to build the system step by step.

Let $terminal(i)$, where $i \in \{1, 2, 3, 4\}$, be the four terminals. Let $tb\_i : \mathbb{B}$ be a local variable to $terminal(i)$ representing the button at the ith terminal. $tb\_i = true$ if and only if the button is lit. The four automata in Fig. 5a show the basic behaviors of a terminal. In automaton $PressButton(i)$, where $i \in \{1, 2, 3, 4\}$, an external request *enters* the external request queue in the central controller only if the button has not been pressed before. Once pressed, the light is on (i.e., $tb\_i := true$). In $ButtonOff$, once the controller informs the terminal that the request has been serviced, the light goes off. In $TerApproach$, event $approach\_req$ signals the approaching of the car, and the

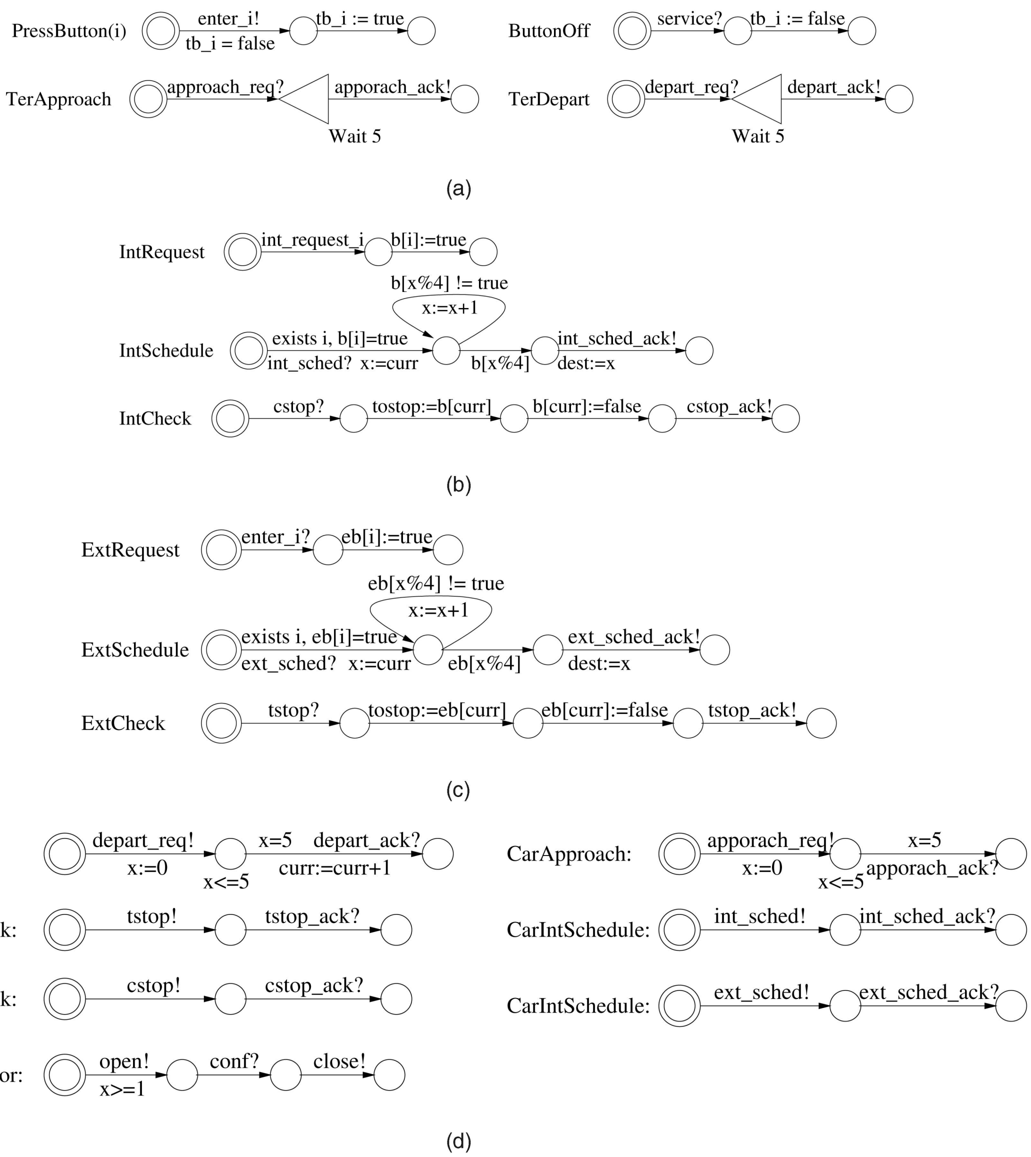


Fig. 5. Basic behavior patterns.

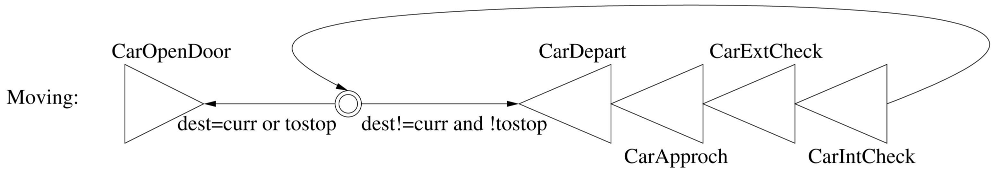


Fig. 6. Composed behaviors (1).

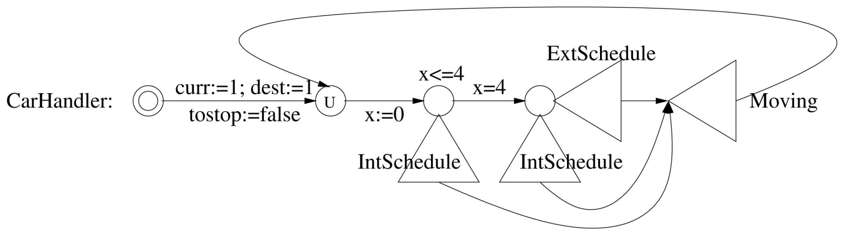


Fig. 7. Composed behaviors (2).

terminal does the necessary preparation (abstracted as a $delay$ in this modeling) and then acknowledges. Last, in $TerDepart$, before the car departs, the terminal is informed by $depart\_req$ and acknowledges by $depart\_ack$. The composed behavior of a terminal is

$$terminal(i) = recur(extchoice(PressButton, ButtonOff, TerApproach, TerDepart)),$$

where PressButton = extchoice(PressButton(1), PressButton(2), PressButton(3), PressButton(4)). The terminals are specified as Terminals = para(terminal(1), terminal(2), terminal(3), terminal(4)).

We model the behaviors of the car similarly. The door has been modeled in Section 2.1. The panel in the car is composed of four buttons. Let b be an array of Boolean variables. $b[i] = true$, where $i \in \{1, 2, 3, 4\}$, means the ith button has been lit already. Let $curr$ and $dest$ be two global variables that record the station the car is approaching and the destination, respectively. Let $tostop : \mathbb{B}$ be a global variable that says whether to stop or not at the next terminal. The automata in Fig. 5b model the basic behaviors of the panel. In automaton $IntRequest$, a passenger in the car may press a button on the panel (modeled as event $int\_request\_i$, where $i \in \{1, 2, 3, 4\}$) and the button is lit afterward. Note that the button may be pressed even if it has been pressed already. In $IntSchedule$, if there is an internal request, the car panel computes the next destination once an input is received on channel int_sched (from the car handler). The destination is set to be the next requested terminal. In $IntCheck$ (when the car is approaching the next terminal), the car handler orders the panel to check whether there is an internal request for the next terminal and decides whether to stop at the next terminal. The overall behaviors of the panel are specified as $Panel = recur(extchoice(IntRequest, IntSchedule, IntCheck))$.

The behavior patterns of the central controller are similar to those of the panel except that the car panel handles internal requests, whereas the controller handles external requests. The automata in Fig. 5c model its basic behaviors. Let eb be an array of Boolean variables that represents whether the external buttons have been pressed and lit. The overall behaviors of the controller are specified as Controller = recur(extchoice(ExtRequest, ExtSchedule, ExtCheck)).

The automata in Fig. 5d model the basic computational logic of the car handler. In order to specify its overall behaviors, we make use of the timed patterns. The compositional Timed Automaton named $Moving$, presented in Fig. 6, specifies the behaviors of the car handler after getting a request (either from internal or external). If the destination is the current terminal, the car door is opened. Otherwise, the car leaves the terminal and heads toward the destination. Once approaching the next terminal, the car checks if there is a newly arrived request from external or internal for the approaching terminal. Afterward, the automaton repeats from the beginning. Note that this hierarchical automaton has already been partially flattened for simplicity.

The overall behavior of the car handler can be composed from the above basic one. Initially, the car handler is idling at some terminal. It waits for an internal request first. If an internal request arrives within 4 seconds, it starts to serve the request. Otherwise, it waits for either an internal or external request and then starts moving. Once it reaches the destination, it repeats from the beginning. The overall behavior of the car handler is represented as follows:

$$CarHandler = recur(seq(timeout(IntSchedule, extchoice(IntSchedule, ExtSchedule), 4), Moving)).$$

Graphically, it is drawn as shown in Fig. 7. Last, the system is the parallel composition of the three components, i.e., para(Terminals, Controller, para(Panel,Door,CarHandler)).

5.2 Analysis of the TCOZ Modeling

Based on the strength of OZ and Timed CSP, TCOZ supports object-oriented design of data structures, as well as compositional design of dynamic behaviors, as demonstrated in [37], [36]. Because modeling using TCOZ (which shares the same bottom-up nature as using the Timed Automata patterns) is largely irrelevant to this paper, we show instead how the TCOZ specification is translated to Timed Automata systematically using our prototype and then verified using UPPAAL. The relevant part of the TCOZ specification is presented in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.52.

Each active class in the TCOZ model is projected to a Timed Automaton template, namely, a terminal template for the class $Terminal$; a car door, a car destination panel, and a car handler template for the class $CarDoor$, $CarPanel$, and $CarHandler$, respectively; and a controller template for the class $Controller$. The terminal template has four instances, which represents the four different terminals according to the TCOZ specifications. The other templates have only one instance. The translation is a straightforward application of Definition 16. We use the $Terminal$ class as an example to show the identification of the states, transitions, guards, and synchronization. Its processes mainly have an external choice pattern and a recursion pattern as defined by its MAIN process:

$$recur(extchoice(M(PressDown), M(DownOff), M(CarApproach), M(CarDepart)), S_P).$$

By applying the translation to each of the processes $PressDown$, $DownOff$, $CarApproach$, and $CarDepart$, we obtain the Timed Automaton template of the class $Terminal$. The above translation is automated 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. Similarly, Timed Automata are generated from other system components, e.g., the template for class $Door$ is shown in Section 2.1. The translation result is stored in an XML format, which can be readily imported by UPPAAL. Now, we can use the simulator and verifier of UPPAAL to simulate the system, as well as to model check some invariants and real-time properties. The key point of the railway system is to provide efficient services. The following are some of the properties that can be formally specified and verified using UPPAAL:

- *Prop1.* 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 a Timed CTL as a bounded liveness property.
- *Prop2.* The door must be open for no more than 10 seconds.
- *Prop3.* The system is deadlock free.
- *Prop4.* Whenever the railcar is moving, the car door is closed.

UPPAAL verified that these properties hold for this given model. As for reference, the time consumption for analyzing each of the properties is listed below. The experiment results are obtained on the Windows XP platform with 2 Gbyte memory and Intel 3.0 GHz CPU. N is the number of terminals.

Properties	Time (N=3)	Time (N=4)
Prop1	8.3s	250.0s
A[] Prop2	1.0s	11.2s
A[] Prop3	2.6s	29.1s
A[] Prop4	1.1s	10.3s

In summary, the Timed Automata patterns allow us to enjoy a hierarchical system design using Timed Automata, i.e., to apply a divide-and-conquer strategy to deal with the complexity of the system design. A high-level design (as the one above where highly coupled system behaviors are encapsulated in one component) is easier to understand and maintain. A high-level system design can be automatically flattened (by applying the definitions plus optimization techniques for reducing the number of states, as well as clocks) and then verified using existing tools like UPPAAL. Compared to the system design using timed patterns directly, which solely focuses on dynamic behaviors, the TCOZ-based design is heavy in modeling data and functional aspects of the system. Because the timed patterns are closely related to compositional operators in TCOZ, the modeling of dynamic behaviors using timed patterns and TCOZ share similar ideas.

6 Conclusion

For the last decades, a variety of formal modeling/specification languages have been proposed for real-time system design and verification. The various modeling and verification techniques all have similarities and differences to some degree. It is important for the formal method community to understand how various techniques differ from each another and how they may benefit from each other. The techniques under consideration, namely, TCOZ/Timed CSP and Timed Automata, deal with behavioral real-time aspects of systems. Lying at each end of the spectrum of formal modeling techniques, TCOZ is designed for the structural specification of high-level complex system requirements, whereas Timed Automata are used to design timed models with simple clock constraints but with highly automated verification support. In this work, we studied both formalisms from a practical point of view, i.e., how can they help each other? We are not arguing which of the two is superior. Instead, we enrich TCOZ with verification support by translating its models to Timed Automata and enrich Timed Automata with composable patterns for high-level system design.

The main contribution of the work is the rich set of timed patterns, which covers all common hierarchical system behaviors like $deadline$, $timeout$, and $timedinterrupt$. These patterns are formally defined so as to achieve composibility in the graphical representations. Moreover, we have shown that new timed patterns may be composed naturally using our timed patterns. These patterns not only provide a proficient interchange media for translating time-enriched process algebra specifications into Timed Automata but also provide a generic reusable framework for developing real-time systems solely using Timed Automata. As shown in Section 5, by decomposing a complex system to subcomponents of manageable size and then composing the subcomponents using timed patterns, these patterns offer a systematic way of fighting the great complexity in system design.

One of the future works is to develop fully automated optimization techniques that could maximally reduce the number of states and clocks while flattening the Timed Automata patterns. This is important because the number of clocks (and states) has a significant impact on the performance of tools like UPPAAL. Another future work is to integrate our prototype with UPPAAL for user convenience. A future work of theoretical interest is to fully compare the expressiveness of timed process algebras like Timed CSP and Timed Automata so as to gain better support for verifying real-time systems.

ACKNOWLEDGMENTS

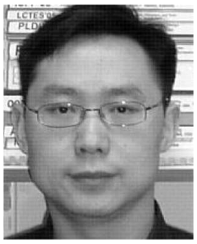
The authors would like to thank Hugh Anderson, Roger Duke, Sun Jing, Wang Hai, Lars Grunske, and the anonymous referees for their insightful comments on this work. This research is partially supported by the research grant R-252-000-201-112 funded by National University of Singapore. Jun Sun is the corresponding author for this paper.

REFERENCES

[1] R. Alur, C. Couroubetis, and D.L. Dill, "Model-Checking for Real-Time Systems," *Proc. Fifth Ann. IEEE Symp. Logic in Computer Science,* pp. 414-425, 1990.
[2] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, pp. 183-235, 1994.
[3] S. Bornot and J. Sifakis, "Relating Time Progress and Deadlines in Hybrid Systems," *Proc. Int'l Workshop Hybrid and Real-Time Systems,* pp. 286-300, 1997.
[4] S. Bornot, J. Sifakis, and S. Tripakis, "Modeling Urgency in Timed Systems," *Proc. Int'l Symp. Compositionality: The Significant Difference,* pp. 103-129, 1997.
[5] H. Bowman, "Modelling Timeouts without Timelocks," *Proc. Fifth Int'l AMAST Workshop Formal Methods for Real-Time and Probabilistic Systems,* pp. 334-353, 1999.
[6] H. Bowman and R. Gómez, "How to Stop Time Stopping," *Formal Aspect of Computing,* vol. 18, no. 4, pp. 459-493, 2006.
[7] H. Bowman, R. Gómez, and L. Su, "A Tool for the Syntactic Detection of Zeno-Timelocks in Timed Automata," *Electronic Notes in Theoretical Computer Science,* vol. 139, no. 1, pp. 25-47, 2005.
[8] P. Brooke, "A Timed Semantics for a Hierarchical Design Notation," PhD dissertation, Univ. of York, 1999.
[9] S.D. Brooke, "A Model for Communicating Sequential Processes," PhD dissertation, Oxford Univ., 1983.
[10] A.M.K. Cheng, *Real-Time Systems: Scheduling, Analysis, and Verification.* John Wiley & Sons, 2002.
[11] A. David and M.O. Möller, "From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata," Technical Report RS-01-11, BRICS, Mar. 2001.
[12] J. Davies, *Specification and Proof in Real-Time CSP.* Cambridge Univ. Press, 1993.
[13] J.S. Dong, N. Fulton, L. Zucconi, and J. Colton, "Formalizing Process Scheduling Requirements for an Aircraft Operational Flight Program," *Proc. First IEEE Int'l Conf. Formal Eng. Methods,* pp. 161-169, 1997.
[14] J.S. Dong, P. Hao, S.C. Qin, J. Sun, and W. Yi, "Timed Patterns: TCOZ to Timed Automata," *Proc. Sixth Int'l Conf. Formal Eng. Methods,* pp. 483-498, 2004.
[15] J.S. Dong, P. Hao, J. Sun, and X. Zhang, "A Reasoning Method for Timed CSP Based on Constraint Solving," *Proc. Eighth Int'l Conf. Formal Eng. Methods,* pp. 342-359, 2006.
[16] J.S. Dong, B.P. Mahony, and N. Fulton, "Modeling Aircraft Mission Computer Task Rates," *Proc. World Congress on Formal Methods,* p. 1855, 1999.
[17] R. Duke and G. Rose, "Formal Object Oriented Specification Using Object-Z," *Cornerstones of Computing,* Macmillan, 2000.
[18] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st Int'l Conf. Software Eng.,* pp. 411-420, 1999.
[19] H. Fuhrmann, J. Koch, J. Rennhack, and R.v. Hanxleden, "The Aerospace Demonstrator of DECOS," *Proc. Eighth Int'l IEEE Conf. Intelligent Transportation Systems,* pp. 19-24, 2005.
[20] C. Ghezzi, D. Mandrioli, and A. Morzenti, "Trio: A Logic Language for Executable Specifications of Real-time System," *J. Systems and Software,* vol. 12, no. 2, pp. 107-123, May 1990.
[21] *UML Resource Page,* Object Management Group, http://www.omg.org/uml/, 2008.
[22] V. Gruhn and R. Laue, "Patterns for Timed Property Specifications," *Electronic Notes in Theoretical Computer Science,* vol. 153, no. 2, pp. 117-133, 2006.
[23] L. Grunske, K. Winter, and R. Colvin, "Timed Behavior Trees and Their Application to Verifying Real-Time Systems," *Proc. 18th Australian Software Eng. Conf.,* pp. 211-222, 2007.
[24] D. Harel and E. Grey, "Executable Object Modeling with Statecharts," *Computer,* vol. 30, no. 7, pp. 31-42, July 1997.
[25] K. Havelund, A. Skou, K.G. Larsen, and K. Lund, "Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL," *Proc. 18th IEEE Real-Time Systems Symp.,* pp. 2-13, 1997.
[26] I.J. Hayes and M. Utting, "Deadlines Are Termination," *Proc. IFIP Working Conf. Programming Concepts and Methods,* 1998.
[27] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.
[28] J. Hoenicke and E.-R. Olderog, "Combining Specification Techniques for Processes, Data and Time," *Proc. Third Int'l Conf. Integrated Formal Methods,* pp. 245-266, 2002.
[29] F. Jahanian and A.K. Mok, "A Graph-Theoretic Approach for Timing Analysis and Its Implementation," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 961-975, Aug. 1987.
[30] S. Konrad and B.H.C. Cheng, "Real-Time Specification Patterns," *Proc. 27th Int'l Conf. Software Eng.,* pp. 372-381, 2005.
[31] L.M. Lai and P. Watson, "A Case Study in Timed CSP: The Railroad Crossing Problem," *Proc. Int'l Workshop Hybrid and Real-Time Systems,* pp. 69-74, 1997.
[32] K.G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing Real-Time Embedded Software Using UPPAAL-TRON: An Industrial Case Study," *Proc. Fifth ACM Int'l Conf. Embedded Software,* pp. 299-306, 2005.
[33] K.G. Larsen, P. Pettersson, and Y. Wang, "Uppaal in a Nutshell," *Int'l J. Software Tools for Technology Transfer,* vol. 1, nos. 1-2, pp. 134-152, 1997.
[34] M. Lindahl, P. Pettersson, and Y. Wang, "Formal Design and Analysis of a Gearbox Controller," *Springer Int'l J. Software Tools for Technology Transfer,* vol. 3, no. 3, pp. 353-368, 2001.
[35] N.A. Lynch and F.W. Vaandrager, "Action Transducers and Timed Automata," *Formal Aspects of Computing,* vol. 8, no. 5, pp. 499-538, 1996.
[36] B. Mahony and J.S. Dong, "Timed Communicating Object Z," *IEEE Trans. Software Eng.,* vol. 26, no. 2, pp. 150-177, Feb. 2000.
[37] B.P. Mahony and J.S. Dong, "Network Topology and a Case Study in TCOZ," *Proc. 11th Int'l Conf. Z Users,* pp. 308-327, 1998.
[38] J. Ouaknine and J. Worrell, "Timed CSP = Closed Timed Epsilon-Automata," *Nordic J. Computing,* vol. 10, no. 2, pp. 99-133, 2003.
[39] A.W. Roscoe, *The Theory and Practice of Concurrency.* Prentice Hall, 1997.
[40] S. Schneider, *Concurrent and Real-Time Systems.* John Wiley & Sons, 2000.
[41] J. Sifakis, "The Compositional Specification of Timed Systems—A Tutorial," *Proc. 11th Int'l Conf. Computer Aided Verification,* pp. 2-7, 1999.
[42] J. Sun, J.S. Dong, J. Liu, and H. Wang, "A Formal Object Approach to the Design of ZML," *Annals of Software Eng.,* vol. 13, pp. 329-356, 2002.
[43] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

Jin Song Dong received the bachelor's (first-class honors) and PhD degrees in computing from the University of Queensland in 1992 and 1996, respectively. From 1995 to 1998, he was a research scientist at the Commonwealth Scientific and Industrial Research Organisation, Australia. Since 1998, he has been with the School of Computing at the National University of Singapore (NUS), where he is currently an associate professor and a member of the PhD supervisors at the NUS Graduate School. He is on the editorial board of the *Formal Aspects of Computing Journal* and *Innovations in Systems and Software Engineering, A NASA Journal.* His research interests include formal methods and software engineering. Some of his papers can be found at http://www.comp.nus.edu.sg/~dongjs.

Ping Hao received the bachelor's degree in telecommunication from the Huazhong University of Science and Technology, China, in 2000 and the PhD degree in computer science from the National University of Singapore (NUS) in 2008. From 2005 to 2006, she was a research assistant in the Software Engineering Laboratory, NUS. Her research interests include formal specification languages, verification, and software engineering. More details about her research and background can be found at http://www.comp.nus.edu.sg/~haoping.



Shengchao Qin received the BSc and PhD degrees from the School of Mathematical Sciences, Peking University, in 1997 and 2002, respectively. He was a research fellow under the Singapore-MIT alliance at the National University of Singapore from July 2002 to December 2004. Since 2005, he has been a lecturer in the Department of Computer Science at Durham University, United Kingdom. His research interests include formal methods, programming languages, and embedded systems. More information about his research can be found at http://www.dur.ac.uk/shengchao.qin.

Jun Sun received the bachelor's and PhD degrees from the School of Computing, National University of Singapore (NUS), in 2002 and 2006, respectively. From 2005 to 2006, he was a research fellow in the School of Computing, NUS. Since 2006, he has been a Lee Kuan Yew postdoctoral fellow in the Department of Computer Science, NUS. His research interests are mainly in formal system specification, verification, and synthesis. In particular, he has been working with a variety of different specification languages and notations in order to develop practical tools for elegant system development. More details about his research and background can be found at http://www.comp.nus.edu.sg/~sunj.

Wang Yi received the PhD degree in computer science from the Chalmers University of Technology, Sweden, in 1991. He is a professor in real-time systems at Uppsala University, Sweden, and a Changjiang professor at North Eastern University, China. From 1991 to 1992, he was a postdoctoral fellow at Aalborg University, Denmark, before he joined the faculty of science and technology at Uppsala University, where he was a lecturer from 1992 to 1994, an associate professor from 1994 to 2000, and has been a professor since 2000. His interests are mainly in the modeling and verification of concurrent, distributed, and real-time systems, in particular, techniques and tools for model-based design, analysis, and implementation of embedded systems and their industrial applications. He is a cofounder of three software tools for modeling and verification, including UPPAAL, a widely used model checker for timed systems. His publications can be found at http://user.it.uu.se/~yi/.