Improved BDD-based Discrete Analysis of Timed Systems

Truong Khanh Nguyen[2], Jun Sun[1], Yang Liu[2], Jin Song Dong[2] and Yan Liu[2]

[1] Information System Technology and Design,
Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] School of Computing
National University of Singapore
{truongkhanh,liuyang,dongjs,yanliu}@comp.nus.edu.sg

Abstract. Model checking timed systems through digitization is relatively easy, compared to zone-based approaches. The applicability of digitization, however, is limited mainly for two reasons, i.e., it is only sound for *closed* timed systems; and clock ticks cause state space explosion. The former is mild as many practical systems are subject to digitization. It has been shown that BDD-based techniques can be used to tackle the latter to some extent. In this work, we significantly improve the existing approaches by *keeping the ticks simple* in the BDD encoding. Taking advantage of the 'simple' nature of clock ticks, we fine-tune the encoding of ticks and are able to verify systems with many ticks. Furthermore, we develop a BDD library which supports not only encoding/verifying of timed state machines (through digitization) but also composing timed components using a rich set of composition functions. The usefulness and scalability of the library are demonstrated by supporting two languages, i.e., closed timed automata and Stateful Timed CSP.

1 Introduction

Model checking of real-time systems has been studied extensively. One popular approach is zone abstraction [1, 2]. The scalability and effectiveness of the zone-based approach have been proved with successful industrial applications, e.g., [3]. Meanwhile, it is known that for a large class of timed verification problems, correctness can be established using an integral model of time (digital clocks) as oppose to a dense model of time [4]. For instance, Lamport argued that model checking of real-time systems can be really simple if digitization is adopted [5]. Digitization translates a real-time verification problem to a discrete one by using clock ticks to represent elapsed time. The advantage is that the techniques which are developed for classic automata verification can be applied without the added complexity of zone operations. One particularly interesting example is model checking with the assumption of non-Zenoness. A timed execution is Zeno if infinitely many discrete steps are taken within finite time. For obvious reasons, Zeno executions are impractical and must be ruled out during the system verification. It is, however, nontrivial to check whether an execution is Zeno or not based on zone graphs [6, 7]. The problem is much simpler with digitization. An execution of a digitized system is non-Zeno if and only if it contains infinitely many clock ticks. Thus a

finite-state system is non-Zeno if on any of its control cycles, time advances with at least one time unit. In other words, this cycle contains at least one clock tick transition, which can be determined efficiently with cycle-detection algorithms. Further, the experiment in [8] showed that BDD-based model checking of digitized systems is more robust with the increment in the number of processes, compared with zone-based approaches.

The disadvantage of digitization is that the number of reachable states of the digitized system is an increasing function of the number of clock ticks, which is determined by the upper-bound of the timing constraints. The experiments in [5] showed that UPPAAL has a clear advantage (over TLC or Spin in verifying the digitized systems) when the time upper-bound is bigger than 10. The same experiments showed that the symbolic model checker SMV is more robust with the increment in time upper-bounds. The question is then: Can BDD-based symbolic model checker scale better with large time upper-bounds? In [9], it has been shown that the size of BDD is very sensitive to time upper-bounds through a theoretical analysis. As a result, the time upper-bounds were thus kept very small in their experiments, i.e., no more than 16.

In this work, we re-visit the problem in order to develop efficient model checking techniques for timed systems. Our investigation shows that if we keep clock ticks simple, *by avoiding clock variables altogether*, we are able to obtain a small BDD encoding of all ticks in a system which scales significantly better than existing approaches. We are able to verify systems with time upper-bounds in the order of thousands. Furthermore, to make this technique available for different timed modeling languages, we build a BDD library for encoding and composing digitized timed systems. The motivation is that complex timed systems are often composed of many components at multiple levels of hierarchies. We propose to use *timed finite-state machines* (TFSMs) to model timed system components, which are designed to capture useful system features like different ways of communication among system components. Next, we define a rich set of system composition functions accordingly based on TFSMs. The library further complements the previous approaches (e.g., UPPAAL, Rabbit [8]) by supporting linear temporal logic (LTL), LTL with weak/strong fairness, non-Zenoness, etc. The usefulness of the library is evidenced by showing that it can be readily used to build model checkers for two different timed modeling languages, e.g., closed timed automata and Stateful Timed CSP [10].

We evaluate the efficiency of the library using benchmark systems with different settings. In the first experiment, systems are modeled and verified with an increment in time upper-bounds. The objective is to show that, by taking advantage of characteristics of clock ticks, our library is reasonably robust with larger number of clock ticks than Rabbit. In the second experiment, the systems are verified with the increment in the number of processes so as to show that our model checker scales up better than model checkers like UPPAAL. Lastly, we show that our model checker verifies LTL properties, with/without non-Zenoness, efficiently.

The rest of the paper is organized as follows. Section 2 presents the design of the library. Section 3 presents the work on supporting two languages. Section 4 evaluates the performance of the library. Section 5 concludes the work.

2 System Models and BDD Encoding

A timed system may be built from the bottom up by gradually composing system components. We propose to model system components using timed finite-state machines (TFSMs), which are designed to capture a variety of system features. In the following, we introduce TFSM and system compositions based on TFSMs. Furthermore, we show abstractly how to generate a BDD encoding of TFSMs in a compositional way.

2.1 Timed Finite-State Machines

Definition 1. *A TFSM is a tuple $\mathcal{M} = (GV, LV, S, init, Act, Ch, T)$ such that GV is a set of finite-domain shared variables; LV is a set of finite-domain local variables such that $GV \cap LV = \varnothing$; S is a finite set of control states; $init \in S$ is the initial state; Act is the alphabet; Ch is a set of synchronous channels*[3]*; and T is a labeled transition relation. A transition label is of the form $[guard]evt\{prog\}$ where $guard$ is an optional guard condition constituted by variables in GV and LV; evt is either an event name, a channel input/output or the special $tick$ event (which denotes 1-unit elapsed time); and $prog$ is an optional transaction, i.e., a sequential program which updates global/local variables.*

A transaction (which may contain program constructs like if-$then$-$else$ or $while$-do) associated with a transition is to be executed atomically. A non-atomic operation is thus to be broken into multiple transitions. TFSMs support many system features. For instance, TFSMs may communicate with each other through shared variables GV, multi-party event synchronization (common events in parallel composition are synchronized) or pair-wise channel communication.

The semantics of $\mathcal{M}$ is a labeled transition system $(C, init_c, \rightarrow)$ such that C contains finitely many configurations of the form (σ_g, σ_l, s) such that σ_g is the valuation of GV and σ_l is the valuation of LV and $s \in S$ is a control state; $init_c = (init_g, init_l, init)$ where $init_g$ is the initial valuation of GV and $init_l$ is the initial valuation of LV; and $\rightarrow$ is defined as follows: for any (σ_g, σ_l, s), if $(s, [guard]e\{prog\}, s') \in T$, then $(\sigma_g, \sigma_l, s) \xrightarrow{e} (\sigma'_g, \sigma'_l, s')$ if the following holds: $guard$ is true given σ_g and σ_l; e is not a synchronous channel input/output; and $prog$ updates σ_g and σ_l to σ'_g and σ'_l respectively. Notice that synchronous input/output cannot occur on its own. Rather, it must be jointly performed by different TFSMs which execute concurrently. Furthermore, $\rightarrow$ contains transitions labeled with events to be synchronized, which later will be synchronized with corresponding transitions from other TFSMs. We remark that timing constraints are captured explicitly by allowing/disallowing transitions labeled with $tick$. For instance, an urgent state is a state which disallows ticks.

Example 1. Fig. 1 shows a TFSM which models a process in Fischer's mutual exclusion protocol. The double-lined circle denotes the initial state. GV contains two variables. Variable id denotes the identifier of the latest process attempting to access the critical session. It is initially 0, which means that no process is attempting. Variable

[3] Asynchronous channels can be mimicked using shared variables.

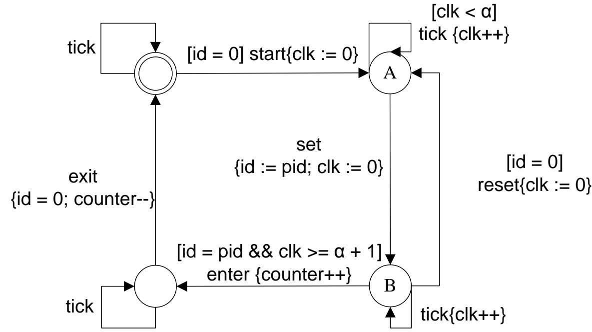

Fig. 1. A TFSM model with clock variables

$counter$ counts the number of processes currently accessing the critical session. By design, $counter$ should be always less than 2. The local variable pid is a unique process identifier which is a constant. In addition, variable $clk \in LV$ is a clock variable which tracks the passage of time. Initially, the TFSM awaits until $id = 0$ and then performs event $start$. At state A, it can set id to its pid (indicating that it is trying to get into the critical session). Event set must occur within α time units as the $tick$ transition is guarded by $clk < \alpha$. At state B, the TFSM waits for at least $\alpha + 1$ time units and then checks whether id is still same as its pid. If so, it enters the critical session; otherwise, it restarts from the beginning via the $reset$ event. □

TFSM can be encoded in BDD following the standard approach. That is, a BDD can be used to encode symbolically the system configuration including valuation of global and local variables as well as the control states. Using two sequences of Boolean variables $\overrightarrow{x}$ and $\overrightarrow{x}'$ (which represent system configurations before and after a transition respectively), transitions of TFSMs can be encoded as BDDs constituted by $\overrightarrow{x}$ and $\overrightarrow{x}'$. An encoded transition is of the form: $g \wedge e \wedge t$ such that g (over $\overrightarrow{x}$) is the encoded guard condition; e is the encoded event and t (over $\overrightarrow{x}$ and $\overrightarrow{x}'$) is the encoded transaction. Interested readers are referred to [11] for details on encoding TFSM.

The encoding of a TFSM is a tuple $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$. $\overrightarrow{V}$ is a set of unprimed Boolean variables encoding global variables, event names including the clock tick, channel names, and channel buffers, which are calculated for the whole system before encoding. $\overrightarrow{v}$ is a set of variables encoding local variables and local control states; $Init$ is a formula over $\overrightarrow{V}$ and $\overrightarrow{v}$, which encodes the initial valuation of the variables. $Trans$ is a set of encoded transitions *excluding tick transitions*. Out (In) is a set of encoded transitions labeled with synchronous channel output (input). Note that transitions in Out and In cannot occur by itself, but must be paired with corresponding input/output communication of other components. Out and In are separated from the rest of the transitions so that they can be matched with corresponding input/output transitions later. Lastly, $Tick$ is a BDD which encodes all the tick transitions. Note that tick transitions must be synchronized among all concurrent TFSMs. Keeping tick transitions separated allows us to realize dedicated optimizations (see below).

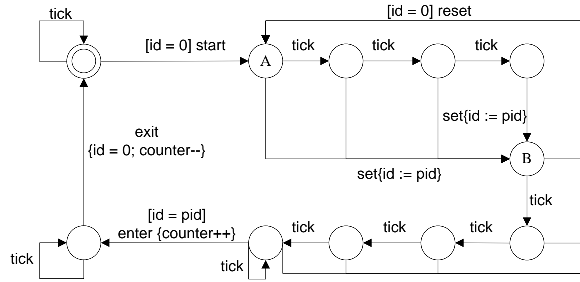

Fig. 2. A TFSM model without clock variables

2.2 Keeping Ticks Simple

In order to handle systems with large time upper-bounds, it is important that we keep the encoding of tick transitions small. There are different ways of capturing timing constraints. For instance, in Fig. 1, the timing constraints at state A and B are captured by using 'clock' clk, i.e., by increasing clk with the tick transitions and guarding system transitions with conditions on clk. Another way of modeling timing constraints is to use only tick transitions without clock variables. For instance, assuming α is 3 in Example 1, Fig. 2 models the same TFSM without clock variables. At state A, at most three tick transitions are allowed to occur before event set occurs, which captures that set must occur within three time units.

We argue that clock variables should be avoided altogether if possible for the following reason. Without clock variables, both the tick transitions and other transitions become simpler since there is no need to introduce a new variable clk; or have transactions to increment clk or to have transition guards on clk. Moreover by generating explicitly the model with tick transitions, we can reduce the state space of the problem. For instance, given the encoding in Figure 1, the total number of potential states (i.e., the product of the control state and the clock value) is 20, whereas with the encoding in Figure 2, it is only 11. This latter encoding thus allows us to save one boolean variable in encoding of one TFSM. This reduction is due to the fact that the latter encodes more 'domain knowledge'. For instance, some of the 20 states are in fact not reachable (e.g., state $(A, clk = 4)$ assuming α is 3) or bi-similar to each other (e.g., state $(init, clk = 0)$ and $(init, clk = 1)$ where $init$ is the initial state).

However if tick transitions are used instead of the clock variables, the number of tick transitions in one TFSM is bigger, linear to the product of all clock ceilings in that TFSM. If we store $Tick$ as a disjunctive partitioned transition function [12], the number of BDDs to encode tick transitions in a system can grow exponentially. Given a system with n TFSMs, each of which has m tick transitions, $Tick$ of the resulted composition has m^n BDDs which are implicitly disjuncted. As a result, the number of BDD-based pre-image and post-image operations grows exponentially too. Thus we store $Tick$ as a single BDD to encode all the tick transitions in a TFSM. It reduces the time spending on BDD-based computation by taking one complex operation instead of m^n simpler

#proc		4	5	6	7	8
time (s)	without clock variables	0	0	0.1	0.2	0.4
	with clock variables	0.6	15	513	×	×
memory (Mb)	without clock variables	21	22	23	24	26
	with clock variables	32	70	425	×	×

Table 1. Compare two different approaches of encoding timing constraints

operations. Lastly, we compare the two different approaches of encoding timing constraint (i.e., with or without clock variables) and show that avoiding clock variables leads to a smaller BDD (as suggested by the memory consumption) and subsequently significantly more efficient verification (as suggested by the verification time). Table 1 summarizes the experiment results on Fischer's protocol using the model in Figure 1 and 2. Thus, in the following, we always avoid clock variables whenever possible.

2.3 System Composition

A complicated system may consist of many components at multiple levels of hierarchies; and components at the same level may be composed in many ways. In the following, we define a few common system composition functions and show how to generate encodings of these functions without constructing the composed TFSM. We fix two TFSMs $\mathcal{M}_i = (GV, LV_i, S_i, init_i, Act_i, Ch_i, T_i)$ where $i \in \{0, 1\}$, and $\mathcal{B}_i = (\overrightarrow{V}, \overrightarrow{v}_i, Init_i, Trans_i, Out_i, In_i, Tick_i)$ which encodes $\mathcal{M}_i$ respectively. Notice that $\overrightarrow{v}_0$ and $\overrightarrow{v}_1$ are disjoint and $\overrightarrow{V}$ is always shared.

Parallel Composition The parallel composition of $\mathcal{M}_0$ and $\mathcal{M}_1$ is a TFSM $\mathcal{M} = (GV, LV, S, init, Act, Ch, T)$ such that $LV = LV_0 \cup LV_1$; $S = S_0 \times S_1$; $init = (init_0, init_1)$; $Act = Act_0 \cup Act_1$; $Ch = Ch_0 \cup Ch_1$; T is the minimum transition relation such that for any $(s_0, [g_0]e_0\{prog_0\}, s_0') \in T_0$; $(s_1, [g_1]e_1\{prog_1\}, s_1') \in T_1$,

- if $e_0 \notin (Act_0 \cap Act_1) \cup \{tick\}$, $((s_0, s_1), [g_0]e_0\{prog_0\}, (s_0', s_1)) \in T$;
- if $e_1 \notin (Act_0 \cap Act_1) \cup \{tick\}$, $((s_0, s_1), [g_1]e_1\{prog_1\}, (s_0, s_1')) \in T$;
- $((s_0, s_1), [g_0 \wedge g_1]e_0\{prog_0;\ prog_1\}, (s_0', s_1')) \in T$ if $e_0 = e_1$ and $e_0 \in (Act_0 \cap Act_1) \cup \{tick\}$. In order to prevent data race, we assume that $prog_0$ and $prog_1$ do not conflict, i.e., update the same variables to different values.
- if $e_0 = ch!v$ is an output on channel ch with value v; and $e_1 = ch?x$ is a matching channel input, $((s_0, s_1), [g_0 \wedge g_1]ch.v\{prog_0;\ prog_1\}, (s_0', s_1')) \in T$;
- if $e_1 = ch!v$ is a channel output; and $e_0 = ch?x$ is a matching channel input, $((s_0, s_1), [g_0 \wedge g_1]ch.v\{prog_1;\ prog_0\}, (s_0', s_1')) \in T$;

Notice that a channel input/output from $\mathcal{M}_i$ may be matched with an output/input from $\mathcal{M}_{1-i}$ to form a transition in T. It is promoted to Ch at the same time because a channel input/output from $\mathcal{M}_i$ may synchronize with another TFSM in the rest of the system. In the contrast, an event in $(Act_0 \cap Act_1) \cup \{tick\}$ must be synchronized by both machines. If $Act_0 \cap Act_1 = \varnothing$, then $\mathcal{M}_0$ and $\mathcal{M}_1$ communicate only through shared variables or channels, which is often referred to as interleaving. For instance, Fischer's protocol is the interleaving of multiple TFSMs defined in Fig. 1.

Let $(\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the BDD encoding the parallel composition of $\mathcal{B}_0$ and $\mathcal{B}_1$. We have $\overrightarrow{v} = \overrightarrow{v}_0 \cup \overrightarrow{v}_1$; $Init = Init_0 \wedge Init_1$. $Trans$ contains three kinds of transitions.

- local transition: if $g_i \wedge e_i \wedge t_i$ is a transition in $Trans_i$ and e_i is an event which is not to be synchronized (i.e., $e_i \notin (Act_0 \cap Act_1) \cup \{tick\}$), $Trans$ contains a transition $g_i \wedge e_i \wedge t_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$, where $(\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ denotes that the local variables of $\mathcal{B}_{1-i}$ are unchanged.
- channel communication: if $g_i \wedge e_i \wedge t_i$ is a transition in Out_i; and $g_{1-i} \wedge e_{1-i} \wedge t_{1-i}$ is a transition in In_{1-i}; and e_i and e_{1-i} are matching channel input/output, $Trans$ contains a transition $g_i \wedge g_{1-i} \wedge e_i \wedge t_i \wedge t_{1-i}$[4].
- barrier synchronization: if $g_i \wedge e_i \wedge t_i$ is a transition in $Trans_i$ and $g_{1-i} \wedge e_i \wedge t_{1-i}$ is a transition in $Trans_{1-i}$ and $e_i \in (Act_0 \cap Act_1)$ is a synchronization barrier and t_i and t_{1-i} do not conflict, $Trans$ contains transition $g_i \wedge g_{1-i} \wedge e_i \wedge t_i \wedge t_{1-i}$.

Out/In contains a transition $g_i \wedge e_i \wedge t_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ if $g_i \wedge e_i \wedge t_i$ is a transition in Out_i/In_i respectively. These transitions could be paired with matching input/output from other TFSMs running in parallel later. Lastly, $Tick$ contains the transition $g_i \wedge g_{1-i} \wedge tick \wedge t_i \wedge t_{1-i}$ if $g_i \wedge tick \wedge t_i$ is a transition in $Tick_i$ and $g_{1-i} \wedge tick \wedge t_{1-i}$ is in $Tick_{1-i}$.

Unconditional Choice An unconditional choice between $\mathcal{M}_0$ and $\mathcal{M}_1$ is a TFSM $\mathcal{M} = (GV, LV, S, init, Act, Ch, T)$ such that $LV = LV_0 \cup LV_1$; $S = ((S_0 \cup \{done\}) \times (S_1 \cup \{done\}))$; $init = (init_0, init_1)$; $Act = Act_0 \cup Act_1$; $Ch = Ch_0 \cup Ch_1$; and T is the minimum transition relation defined as follows. Notice that we introduce a special state $done$ which denotes the state of one component after the other component is chosen. For any $(s_0, [g_0]e_0\{prog_0\}, s'_0) \in T_0$; any $(s_1, [g_1]e_1\{prog_1\}, s'_1) \in T_1$,

- if $e_0 = e_1 = tick$, $((s_0, s_1), [g_0 \wedge g_1]tick\{prog_0;\ prog_1\}, (s'_0, s'_1)) \in T$;
- if $e_0 \neq tick$, $((s_0, s), [g_0]e_0\{prog_0\}, (s'_0, done)) \in T$ for all $s \in S_1 \cup \{done\}$;
- if $e_1 \neq tick$, $((s, s_1), [g_1]e_1\{prog_1\}, (done, s'_1)) \in T$ for all $s \in S_0 \cup \{done\}$;
- if $e_0 = tick$, $((s_0, done), [g_0]tick\{prog_0\}, (s'_0, done)) \in T$;
- if $e_1 = tick$, $((done, s_1), [g_1]tick\{prog_1\}, (done, s'_1)) \in T$;

Initially when the choice is not resolved, if both components take a tick transition, then so does the choice. Only after one of the components takes an action, the choice is resolved and the other component goes to the $done$ state.

Let $(\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the BDD encoding of the choice between $\mathcal{B}_0$ and $\mathcal{B}_1$ such that $\overrightarrow{v} = \overrightarrow{v}_0 \cup \overrightarrow{v}_1 \cup \{choice\}$ where $choice \in \{-1, 0, 1\}$ is a fresh integer variable of value -1 (i.e., the choice is not resolved), 0 (i.e., $\mathcal{M}_0$ has been chosen), or 1 (i.e., $\mathcal{M}_1$ has been chosen); $Init = Init_0 \wedge Init_1 \wedge (choice = -1)$; $Trans, Out, In$ contain the transition $(choice = -1 \vee choice = i) \wedge g_i \wedge e_i \wedge t_i \wedge (choice' = i)$ if $g_i \wedge e_i \wedge t_i$ is a transition in $Trans_i$, Out_i or In_i respectively. Lastly, a transition $(choice = -1) \wedge g_i \wedge g_{1-i} \wedge tick \wedge t_i \wedge t_{1-i} \wedge (choice' = -1)$ is in $Tick$ if $g_i \wedge tick \wedge t_i$ is a transition in $Tick_i$ and $g_{1-i} \wedge tick \wedge t_{1-i}$ is a transition in $Tick_{1-i}$. Moreover $Tick$ also contains tick transitions from $\mathcal{M}_i$ when the choice is already resolved, $(choice = i) \wedge g_i \wedge tick \wedge t_i \wedge (choice' = i)$.

[4] In our encoding, matching synchronous input/output are labeled with the same event.

Timeout A common timed composition function is timeout, i.e., if a system component is not responding within certain time units, then another component takes over control. Given $\mathcal{M}_0$ and $\mathcal{M}_1$ and a constant d, the timeout is a TFSM $\mathcal{M} = (GV, LV, S, init, Act, Ch, T)$ such that $LV = LV_0 \cup LV_1$; $S = S_0 \cup S_1 \cup \{state_i \mid 1 \leq i \leq t\}$; $init = init_0$; $Act = Act_0 \cup Act_1$; $Ch = Ch_0 \cup Ch_1$; and T is the minimum transition relation defined as follows. Notice that we introduce t states to remember the time passage while the $\mathcal{M}_0$ delays its first action. For any $(s_0, [g_0]e_0\{prog_0\}, s_0') \in T_0$; any $(s_1, [g_1]e_1\{prog_1\}, s_1') \in T_1$, T is defined as follow

- $(init_0, tick, state_1) \in T$
- $(state_i, tick, state_{i+1}) \in T$ where $1 \leq i \leq t-1$
- $(state_t, \tau, init_1)$. The timeout occurs and the control is passed to $\mathcal{M}_1$.
- $(s, [g_0]e_0\{prog_0\}, s_0') \in T$ for all $s \in init_0 \cup \{state_1, ..., state_t\}$ where $s_0 = init_0$, e_0 is not a tick. Actions from initial state can happen during the d-unit-long period.
- $(s_0, [g_0]e_0\{prog_0\}, s_0')$ where $s_0 \neq init_0$
- $(s_1, [g_1]e_1\{prog_1\}, s_1') \in T$

The corresponding encoding of $\mathcal{M}$ is built in the standard way. Notice that $timeout$ can be equivalently defined by adopting an integer clock variable clk which is updated by every tick transition and guarding every transition of $\mathcal{M}_0$ with a constraint on clk. The above definition, however, keeps tick transitions simple by avoiding clock variables.

Deadline A timed system requirement may put a bound on the execution time of a component, i.e., a component must terminate before certain time units. A TFSM $\mathcal{M}_0$ with a deadline d is a TFSM $\mathcal{M} = (GV, LV, S, init, Act, Ch, T)$ such that $LV = LV_0$; $S = S_0 \times \{0, 1, \cdots, d\}$ where the numbers represent the number of elapsed time units; $init = (init_0, 0)$; $Act = Act_0$; $Ch = Ch_0$; and T is the minimum transition relation such that:

- for any $(s, [g]e\{prog\}, s') \in T_0$ and $e \neq tick$, $((s, d_0), [g]e\{prog\}, (s', d_0)) \in T$ for all $d_0 \in \{0, 1, \cdots, d\}$.
- for any $(s, [g]tick\{prog\}, s') \in T_0$, $((s, d_0), [g]tick\{prog\}, (s', d_0 + 1)) \in T$ for all $d_0 \in \{0, 1, \cdots, d-1\}$.

Similarly, the corresponding BDD encoding of $\mathcal{M}$ is built in the standard way.
Through literature survey and case studies, we collected and defined more than twenty composition functions. Other functions like time/event interrupt, sequential composition, conditional choice, repetition, etc., are similarly defined. Interested readers can refer to [11] for the complete list. We remark that the compositions remain as TFSM and therefore not only system components can be composed repeatedly but also the library of system composition functions is extensible.

3 Case Studies

In this section, we show how to support model checking of two fairly different languages, i.e., closed timed automata and Stateful Timed CSP, using our library.

3.1 Closed Timed Automata

Given a set of clocks C, the set $\Phi(C)$ of closed clock constraints δ is defined inductively by: $\delta := x \sim n \mid \neg\delta \mid \delta \wedge \delta$ where $\sim \in \{=, \leq, \geq\}$; x is a clock in C and $n \in \mathbb{R}^+$ is a constant. Without loss of generality (Lemma 4.1 of [13]), we assume that n is an integer constant. The set of downward closed constraints obtained with $\sim = \leq$ is denoted as $\Phi_{\leq}(C)$. A clock valuation v for a set of clocks C is a function which assigns a real value to each clock. A clock valuation v satisfies a clock constraint δ, written as $v \models \delta$, if and only if δ evaluates to true using the clock values given by v. For $d \in \mathbb{R}_+$, let $v+d$ denote the clock valuation v' such that $v'(c) = v(c) + d$ for all $c \in C$. For $X \subseteq C$, let clock resetting notion $[X \mapsto 0]v$ denote the valuation v' such that $v'(c) = v(c)$ for all $c \in C \setminus X$ and $v'(x) = 0$ for all $x \in X$.

Definition 2. *A closed timed automaton $\mathcal{A}$ is a tuple $(S, init, \Sigma, C, L, \rightarrow)$ where S is a finite set of states; $init \in S$ is an initial state; Σ is an alphabet; C is a finite set of clocks; $L : S \rightarrow \Phi_{\leq}(C)$ is a function which associates an invariant with each state; $\rightarrow: S \times \Sigma \times \Phi(C) \times 2^C \times S$ is a labeled transition relation.*

A transition $(s, e, \delta, X, s') \in \rightarrow$ is fired only if δ and $L(s)$ are satisfied by the current clock valuation v and $[X \mapsto 0]v$ satisfies $L(s')$. After event e occurs, clocks in X are set to zero. Given any clock c in C, the upper-bound of time constraints associated with a clock c, denoted as $\lceil c \rceil$, is called its ceiling. A closed timed automaton can be digitized [4] and interpreted as a TFSM $\mathcal{M} = (\varnothing, \varnothing, St, x_{init}, Act, \varnothing, T)$ which is defined as follows. A state in St is a pair (s, v) where $s \in S$ and v is the valuation of all the clocks in C such that for every clock $c \in C$, $v(c)$ is a number in $\{0, \cdots, \lceil c \rceil\}$; $x_{init} = (init, v_0)$ where v_0 is a clock valuation which assigns every clock value 0; and T contains two kinds of transitions.

- event-transitions: for any $(s, e, \delta, X, s') \in \rightarrow$, $((s, v), e, (s', v')) \in T$ if v satisfies δ and $L(s)$ and $v' = [X \mapsto 0]v$ and v' satisfies $L(s')$.
- time-transitions: for any $(s, v) \in S$, $((s, v), tick, (s, v')) \in T$ such that for any $c \in C$, $v'(c) = v(c) + 1$ if $v(c) < \lceil c \rceil$ or $v'(c) = v(c)$ otherwise; and v' satisfies $L(s)$.

Notice that timing constraints are captured using tick transitions and therefore in the event-transitions above, the transitions are *not* guarded. It is obvious that our library can be used to support verification of closed timed automata as well as many additional features introduced in UPPAAL. For instance, interleaving of multiple closed timed automata can be encoded using the parallel composition function; pair-wise synchronous channel communications can be captured using channels supported in the library; etc. Furthermore, it is straightforward to support hierarchical timed automata [14, 15] using our library (by applying the corresponding composition functions) as long as all clock constraints are closed.

It is worth mentioning that a clock which is shared by multiple timed automata is modeled as a shared variable (ranging from 0 to $\lceil c \rceil$) in GV rather than resolved using tick transitions, due to arbitrary clock resetting. A tick transition in the composition is associated with a program which increases every shared clock except those which have reached their ceilings. This encoding complicates the encoding of tick transitions. Nonetheless, we observe that many real-world timed systems use local clocks only.

3.2 Stateful Timed CSP

Stateful Timed CSP (STCSP) [10] extends Timed CSP to capture hierarchical timed systems with non-trivial data structures and operations. Different from timed automata, STCSP relies on implicit clocks to capture timed aspects of system behaviors. It has been shown that STCSP, like Timed CSP, is equivalent to closed timed automata with τ-transitions [10], and thus can be potentially supported by our library.

A STCSP model is a tuple $\mathcal{S} = (Var, init_G, P)$ where Var is a finite set of *finite-domain* global variables, $init_G$ is the initial valuation of the variables and P is a timed process. Process P can be defined using a rich set of process constructs. The following shows a core subset of them.

$$\begin{aligned} P = \; & Stop \mid Skip \mid e \rightarrow P \mid a\{program\} \rightarrow P \mid if\ (b)\ \{P\}\ else\ \{Q\} \mid P;\ Q \\ & \mid\ P \setminus X \mid (P \mid Q) \mid P\,|[\,X\,]|\,Q \mid Wait[d] \mid P\ timeout[d]\ Q \\ & \mid\ P\ interrupt[d]\ Q \mid P\ within[d] \mid P\ deadline[d] \mid Q \end{aligned}$$

The un-timed process operators are either borrowed from CSP [16] or self-explanatory. We thus focus on the timed operators. Assume that d is a positive integer constant. Process $Wait[d]$ idles for exactly d time units (and becomes *Skip* afterwards). Process $P\ timeout[d]\ Q$ behaves exactly as P if the first observable event of P occurs before d time units (since process $P\ timeout[d]\ Q$ is activated). Otherwise, Q takes over control after exactly d time units. In process $P\ interrupt[d]\ Q$, if P terminates before d time units, $P\ interrupt[d]\ Q$ behaves exactly as P. Otherwise, $P\ interrupt[d]\ Q$ behaves as P until d time units and then Q takes over. In contrast to $P\ timeout[d]\ Q$, P may engage in multiple observable events before it is *interrupted*. Process $P\ within[d]$ must react within d time units, i.e., an observable event must be engaged by process P within d time units. In process $P\ deadline[d]$, P must terminate within d time units, possibly after engaging in multiple observable events.

Example 2. Fischer's mutual exclusion algorithm can be modeled as a STCSP model $(V, v_i, Protocol)$ where V contains two variables id and $counter$, which play the same roles as in Example 1.

$$\begin{aligned} Proc(pid) \;&\widehat{=}\; \textbf{if}\ (id = 0)\ \{ \\ &\qquad Started(pid) \\ &\quad \} \\ Started(pid) \;&\widehat{=}\; (set\{id := pid\} \rightarrow Wait[\alpha + 1])\ within[\alpha]; \\ &\quad \textbf{if}\ (id = pid)\ \{ \\ &\qquad enter\{counter := counter + 1\} \rightarrow \\ &\qquad exit\{counter := counter - 1;\ id := 0\} \rightarrow Proc(pid) \\ &\quad \}\ \textbf{else}\ \{ \\ &\qquad reset \rightarrow Started(pid) \\ &\quad \} \end{aligned}$$

Process $Protocol$ is the parallel composition of the process, i.e., $Proc(1)\ \|\ \cdots\ \|\ Proc(n)$ where n is a constant representing the number of processes. Process $Proc(pid)$ models a process with a unique integer identifier pid. If id is 0 (i.e., no other process

is attempting), id is set to be pid by action set. Note that set must occur within α time units (captured by $within[\alpha]$). Next, the process idles for $\alpha + 1$ time units (captured by $Wait[\alpha + 1]$). It then checks whether id is still pid. If so, it enters the critical section and leaves later. Otherwise, it restarts from the beginning via $reset$ action.

Given a STCSP model $\mathcal{S} = (Var, init_G, P)$, its discrete operational semantics are defined through a set of firing rules. Elapsed time is defined explicitly through transitions labeled with tick. Interested readers are referred to [10]. Supporting STCSP with our library is not trivial due to two reasons. Firstly, STCSP is capable of specifying irregular or even non-context-free languages (due to unbounded recursion, refer to [16] for concrete examples), which are beyond the expressiveness of our library. We thus focus on a subset of STCSP models which are $finite\text{-}state$, as defined in [10]. Secondly, it is not clear what are primitive system components given a STCSP model. Notice that auxiliary variables are sometimes introduced in the BDD composition, which may result in a *non-optimal* encoding. Given a simple system with 1000 simple choices, ideally, 10 Boolean variables are sufficient to capture all outcomes. If the choice pattern is applied each time instead, then 999 Boolean variables are added. This example may suggest that the composition functions should be avoided, whereas we argue that the composition functions may be inevitable as knowing the exact number of states in the composition is as hard as reachability analysis. In order to minimize the overall time, one thus has to find a balance between quick encoding (which may imply more verification time) and fast verification (which may be implied by an optimal encoding).

In this work, given a STCSP model, static system analysis is firstly performed so as to identify maximum sub-systems which do not contain a parallel composition. For instance, in Example 2, the identified maximum sub-system is $Proc(pid)$ where $pid \in \{1, \cdots, n\}$. Next, one TFSM is generated systematically from the maximum sub-systems, according to the firing rules, and then encoded using BDD. Finally, the BDD encodings are composed using the respective composition functions so as to generate the BDD encoding of the model. Notice that extending our library with functions to support process constructs in STCSP is straightforward based on its formal operational semantics.

4 Evaluation

The BDD library [11] has been implemented as part of the PAT framework [17, 18]. It is based on the CUDD package, with about thirty classes and thousands of lines of C# code. A range of properties are supported, e.g., reachability analysis or LTL with or without non-Zenoness assumptions or fairness assumptions, etc. Verification of LTL with non-Zenoness assumption is based on a symbolic implementation of the automata-based approach [19], with an additional checking for non-Zenoness (i.e., a strongly connected component is accepting if it is not only Büchi fair but also contains at least one tick transition).

In the following, we evaluate the model checker for closed timed automata developed based on the library, by comparing its performance with existing timed automata model checkers. An automatic translator is developed to translate timed automata into

bound		32	64	128	256	512	1024	2048	3096
time	PAT	0.5	1.4	5	17	68	293	1297	3018
	Rabbit	5.5	44	570	×	×	×	×	×
memory	PAT	16	21	41	49	104	298	494	519

Table 2. Fischer's protocol with 4 processes

bound		20	40	80	160	320	640	1280	2560
time	PAT	0.5	1.3	4	9	29	105	428	1853
	Rabbit	2.6	5.3	13.4	54.4	256	1510	×	×
memory	PAT	17	24	31	35	62	122	303	446

Table 3. Railway control system with 4 stations

TFSM using the approach documented in Section 3.1. Notice that there is limited tool support (other than our own) for STCSP. Three benchmark systems are used: Fischer's protocol, a railway control system and the CSMA/CD protocol. All models are available online [11]. The test bed is a PC with Intel Core 2 Duo E6550 CPU at 2.33GHz and 2GB RAM. Because the maximal memory for Rabbit is 800MB, in the first two experiments, PAT and Rabbit are both allocated 800MB memory. For other cases, tools are set to run until the memory exhausts.

The first question is how well the library scales with the number of clock ticks. In the first experiment, we exponentially increase the upper bound of the timing constraints while keeping the number of processes constant. Table 2, 3 and 4 summarize the verification time, which includes both system encoding time and searching time (in seconds), and peak memory usage (in Megabytes). $\times$ means either out of memory or running more than 2 hours.

All of the properties verified are safety condition which are unreachable from the initial state. The row $bound$ shows the maximum time upper-bound. The $bound$ in CSMA/CD protocol is in the form m/n where m is the time for signal propagation and n is the time for data transmission. The memory consumption of Rabbit is not available from the tool.

The data confirm that time and memory consumptions do increase with the number of tick transitions. Nonetheless, PAT is more robust than Rabbit, e.g., Rabbit exhausts the memory earlier, whereas PAT is able to handle relatively large time upper bounds (e.g., more than one thousand for all three cases). This outperformance can come from of our strategy of Keeping Ticks Simple (section 2.2). Zone-based approaches like the one implemented in UPPAAL are more robust to the increment of the $bound$. UPPAAL's time/memory consumption remains constant (i.e., about one second and 30Mb) as expected. However, notice that UPPAAL's performance could be sensitive to the ratio of time bounds in a model (even if the bounds are small), which is not the case for digitization-based approaches. We refer the readers to [5] for details.

In the second experiment, we increase the number of processes (while keeping the time upper bounds constant) and compare the performance of PAT, UPPAAL and Rabbit. The verification results are summarized in Table 5, 6 and 7. It can be seen that both PAT and Rabbit offer significantly better performance than UPPAAL on Fischer's protocol

bound		8/248	12/372	16/497	20/621	26/808	40/1243
time	PAT	5	10	21	35	67	205
	Rabbit	10	32.7	67	90	342	1160
memory	PAT	31	72	126	245	468	518

Table 4. CSMA/CD with 4 processes

	#proc	8	12	16	24	32	40	50
time	PAT	0.4	1.1	4	20	61	195	531
	UPPAAL	1	200	×	×	×	×	×
	Rabbit	1.6	4.4	12	60	180	473	1142
memory	PAT	17	26	47	136	278	386	757
	UPPAAL	29	629	×	×	×	×	×

Table 5. Fischer's protocol with time upper-bound 4

and the CSMA/CD protocol. For railway control system, PAT and Rabbit take more time than UPPAAL for less than 10 processes. It is likely due to the queue data structure in the model, which is costly to support using BDD. Compared with Rabbit, in this experiment, PAT is better than Rabbit in Fischer's protocol and railway control system whereas Rabbit is faster than PAT in CSMA/CD protocol

In addition to reachability analysis, our library offers verification of the full set of LTL formulae, LTL with non-Zenoness assumption, etc. In the following, we compare the performance of verifying liveness properties, with non-Zenoness (row -Zeno) or without non-Zenoness (row +Zeno). Two approaches are compared, i.e., zone-based approach implemented in UPPAAL and the BDD-based approach proposed in this work (i.e., row PAT). The liveness properties are all progress properties which are supported by UPPAAL. Notice that verification with non-Zenoness is not supported in UPPAAL. Furthermore, Rabbit does not support liveness.

As shown in Table 8, BDD-based approach can handle more processes than UPPAAL for Fischer's protocol and CSMA/CD. It is, however, slower than UPPAAL for railway control system. Encoding the queue data structure symbolically, e.g., pushing and popping an element, makes the BDD of the transition function complex. Thus the BDD-based operations over the transition functions are slow. In addition, the experiments suggest that checking non-Zenoness does incur computational overheads. The reason of these overheads is that the additional computation to discard the strongly connected components which do not contain any tick transition.

5 Discussion

The technical contribution is twofold. Firstly, we develop a BDD library which supports verification of timed systems based on digitization. The library is shown to be reasonably robust with a large number of tick transitions and efficient in verifying benchmark systems. Secondly, based on the library, two model checkers are developed to support two different timed modeling languages.

	#proc	6	7	8	9	10
time	PAT	1.8	6	16	58	169
	UPPAAL	0.2	1.1	7.9	83.1	×
	Rabbit	53	805	×	×	×
memory	PAT	33	64	170	460	715
	UPPAAL	26	36	111	835	×

Table 6. Railway control system with time upper-bound 5

	#proc	8	10	12	14	16	32	64	128
time	PAT	0.3	0.3	0.4	0.6	0.8	5	45	593
	UPPAAL	0.4	3.0	22.9	163	×	×	×	×
	Rabbit	1	1	1.3	1.4	1.5	3	16.1	80
memory	PAT	16	17	18	25	28	73	312	661
	UPPAAL	29	51	292	1894	×	×	×	×

Table 7. CSMA/CD with time upper-bound 1/4

This work follows the line of research on using digital clocks for modeling and verifying timed systems. In [4], the usefulness and limitations of digital clocks have been formally established, which forms the theoretical background of this work. In contrast to the approach in [5] where integer clock variables are used, we use tick transitions only and avoid clock variables so as to obtain a smaller BDD encoding of tick transitions. As a result, we are able to verify systems with many more ticks or processes. In the name of improving modularity, Lamport's method is slightly improved by work in [20]. This work continues the line of work by Beyer [9, 8] to cope with large time upper-bounds and supports liveness properties and liveness with fairness/non-Zenoness. This work is remotely related to work on symbolic model checking of timed systems [21]. As for future work, we are constantly optimizing the library so as to encode further state reduction techniques, e.g., symmetry reduction and, more importantly, compositional verification techniques.

Acknowledge

We would like to thank anonymous reviewers for their extremely valuable comments. This research is partially supported by project IDG31100105/IDD11100102 from Singapore University of Technology and Design, and TRF project 'Research and Development in the Formal Verification of System Design and Implementation'.

Model		Fischer						Railway Control				CSMA/CD			
#proc		6	8	10	12	14	16	6	7	8	9	4	6	8	9
+Zeno	PAT	5	39	177	599	1653	4345	14	48	157	887	0.2	3	24	106
	UPPAAL	2.3	6711	×	×	×	×	0.4	2.6	24.1	242	0	0.6	662	×
-Zeno	PAT	9	59	269	980	3014	×	21	66	207	1006	0.4	5	55	368

Table 8. LTL model checking with/without non-Zenoness

References

1. B. Berthomieu and M. Menasche. An Enumerative Approach for Analyzing Time Petri Nets. In *IFIP Congress*, pages 41–46, 1983.
2. D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.
3. 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. In *RTSS*, pages 2–13, 1997.
4. T. A. Henzinger, Z. Manna, and A. Pnueli. What Good Are Digital Clocks? In *ICALP*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.
5. L. Lamport. Real-Time Model Checking Is Really Simple. In *CHARME*, volume 3725 of *LNCS*, pages 162–175. Springer, 2005.
6. S. Tripakis. Verifying Progress in Timed Systems. In *5th International AMAST Workshop ARTS on Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314. Springer, 1999.
7. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient Emptiness Check for Timed Büchi Automata. In *CAV*, volume 6174 of *LNCS*, pages 148–161, 2010.
8. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *CAV*, volume 2725 of *LNCS*, pages 122–125, 2003.
9. D. Beyer and A. Noack. Can Decision Diagrams Overcome State Space Explosion in Real-Time Verification? In *FORTE*, pages 193–208. Springer, 2003.
10. J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and E. André. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 2011. to appear.
11. T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu. BDD-based Discrete Analysis of Timed Systems. `http://www.comp.nus.edu.sg/%7Epat/bddlib/`, 2012.
12. Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic Model Checking with Partitioned Transistion Relations. In *VLSI*, pages 49–58, 1991.
13. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
14. X. L. Jin, H. D. Ma, and Z. H. Gu. Real-Time Component Composition Using Hierarchical Timed Automata. In *QSIC*, pages 90–99. IEEE, 2007.
15. A. David, R. David, and M. O. Möller. From HUPPAAL to UPPAAL - A Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical report, Department of Computer Science, University of Aarhus, 2001.
16. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
17. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, volume 5643 of *LNCS*. Springer, 2009.
18. Yang Liu, Jun Sun, and Jin Song Dong. Developing Model Checkers Using PAT. In *ATVA*, pages 371–377, 2010.
19. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986.
20. H. Wang and W. MacCaull. Verifying Real-Time Systems using Explicit-time Description Methods. In *QFM*, volume 13 of *EPTCS*, pages 67–78, 2009.
21. G. Morbé, F. Pigorsch, and C. Scholl. Fully Symbolic Model Checking for Timed Automata. In *CAV*, volume 6806 of *LNCS*, pages 616–632. Springer, 2011.

View publication stats