Modeling Timed Automata Theory in PVS*

Qingguo Xu
School of Computer Engineering and Science,
Shanghai University
No. 149, Yanchang Rd., Shanghai 200072, P. R C

Huaikou Miao
School of Computer Engineering and Science,
Shanghai University
No. 149, Yanchang Rd., Shanghai 200072, P. R C

Abstract - *A mechanized system called FVofTA (Formal Verification of Timed Automata) for specifying and reasoning about real-time systems using TA (Timed Automata) theory in PVS (Prototype Verification System) is proposed in this paper. This system includes two parts: one for modeling real-time system using TA template in PVS and the other for proof intending for formal verification of real-times system. The first part of this system is given via a case study in this paper. The formal verification results for this case study show that our modeling method is effective. The method used in this system reflects the nature of the theory TA modeling and is easily grasped by the general user.*

Keywords: real-time system, timed automata, software verification, PVS

1 Introduction

Timed Automaton (TA) is a natural tool in modeling and verifying real-system. There are many real-time systems which are modeled by TA. Researchers have proposed many innovative formal methods for developing real-time systems using TA model [1, 2]. Such methods can give system developers and customers greater confidence that the real-time systems satisfy their requirements, especially their critical requirements. However, applying formal methods to practical systems raises several challenging problems, e.g., how to make formal descriptions and formal proofs understandable to developers and how to design software tools in support of formal methods that are usable to developers.

To address these challenges, inspired by the works in [1,2,9], we are developing a mechanized system called FVofTA (Formal Verification of Timed Automata) for specifying and reasoning about real-time systems using TA theory in PVS (Prototype Verification System) [3, 4, 5]. The overall structure of FVofTA is shown in Figure 1. Our approach is to build a family of formal theory customized for specifying and verifying systems represented in terms of specific mathematical models.

This paper introduces how we have built upon the mechanical proof system PVS to support formal specification and verification of real-time systems modeled as TA. Then we introduce the application of the methods of mechanical proof [7]. The methods used in this paper are part of FVofTA under development. FVofTA will provide a mechanical assistance that allows humans to specify and reason about real-time systems in a direct manner. We first give the theories about clocks in PVS, and then give some generic timed automaton theories template for modeling real-time system by importing the clocks theories; finally a case study is given.

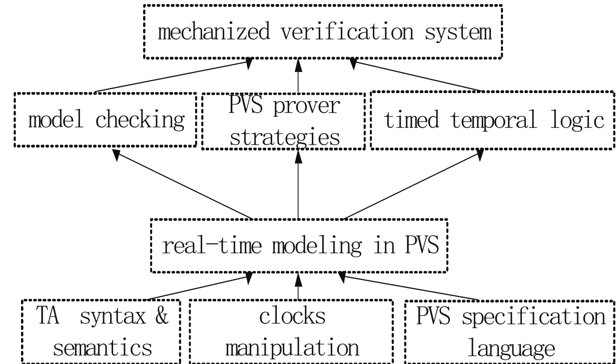

Figure 1. The overall Structure of FVofTA

The rest of this paper is organized as follows. Section 2 introduces the formal syntax and semantics for timed automata. Section 3 presents the timed automaton theory and the necessary clocks theories required by the TA theory in PVS. A case study is investigated in section 4. The related works are discussed in section 5. Finally, the conclusions and the future works under consideration are drawn.

2 Timed Automata Model

The Timed Automata [2,8] for modeling real-time system was invented by Rajeev Alur and David L. Dill. In order to model timed behaviors, the clock constraints and the clock interpretations should be firstly defined.

Definition 1 clock interpretations

A clock interpretation v for a set C of clocks assigns a real value to each clock, that is a mapping from C to the set $\mathbb{R}$ of nonnegative real numbers. ■

Definition 2 clock constraints

For a set of C of clock variables, the set $\Phi(C)$ of clock constraints φ is defined as

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

* This work is supported by the Natural Science Foundation of China (60373072), National 973 Program (2002CB312001) and the Science Development Foundation of Education Committee of Shanghai.

where x is a clock in C and c is a constant in $\mathbb{Q}$, the set of nonnegative rational numbers. ■

We say that a clock interpretation v for C satisfies a clock constraint φ over C if and only if φ evaluates to true according to the values given by v. For $\delta \in \mathbb{R}$, $v+\delta$ denotes the clock interpretation which maps every clock x to the value $v(x)+\delta$. For $Y \subseteq C$, $v[Y:=0]$ denotes the clock interpretation for C that assigns 0 to each $x \in Y$, and agrees with v over the rest of the clocks.

Timed Automaton (TA) is finite state systems with some clocks, and the formal definition for TA is as follows:

Definition 3 Timed Automata

A timed automaton is a tuple $<L, L^0, \Sigma, E, C, Inv, Gad, Rst>$ with

- L, a finite set of locations with initial locations set $L^0 \subseteq L$
- Σ, a finite set of labels
- $E \subseteq L \times \Sigma \times L$, a set of edges (also called switches)
- C, a finite set of clocks
- $Inv: L \to \Phi(C)$, a function that assigns to each locations $l \in L$ an invariant $Inv(l)$
- $Gad: E \to \Phi(C)$, a function that labels each edge $e \in E$ with a clock constraint Gad(e) (called *guard*) in $\Phi(C)$ over C, and
- $Rst: E \to 2^C$, a function that assigns to each edge $e \in E$ a set of clocks $Rst(e)$.

$L \times \Sigma \times 2^C \times \Phi(C) \times L$ is a set of switches. A switch $< s, a, \varphi, \lambda, s' >$ represents a transition from location s to location s' on symbol a, if $e=(s, a, s')$ is in E, $\varphi = Inv(s) \wedge Gad(e)$ is enabled, and $\lambda = Rst(e)$. ■

The semantics of a TA A is defined by associating a transition system S_A with it.

Definition 4 Transition System

Transition System is a tuple $< Q, Q^0, \Sigma, \to >$, where

- Q is a set of states,
- $Q^0 \subseteq Q$ is a set of initial states,
- Σ is a state of labels (or events), and
- $\to \subseteq Q \times \Sigma \times Q$ is a set of transitions.

For a transition $<q, a, q'>$ in $\to$, we write $q \xrightarrow{a} q'$. The system starts in an initial state, and if $q \xrightarrow{a} q'$ then the system can change its state form q to q'. We write $q \to q'$ if $q \xrightarrow{a} q'$ for some label a. $q \to^* q'$ denotes that state q' is reachable from the state q. ■

Thus, a state of S_A is a pair (s,v) such that s is a location of TA A and v is a clock interpretation for clock set C such that v satisfies the invariant $I(s)$. The set of all states of A is denoted Q_A. A state (s,v) is an initial state if s is an initial location of A and $v(x)=0$ for all clocks x. There are two types of transitions in S_A :

- State can change due to elapse of time: for a state (s,v) and a real-valued time increment $\delta \geq 0$, $(s,v) \xrightarrow{\delta} (s, v+\delta)$ if $\forall \delta', 0 \leq \delta' \leq \delta, v+\delta'$ satisfies the invariant $I(s)$,
- State can change due to a locations-switch: for a state (s,v) and a switch $< s, a, \varphi, \lambda, s' >$ such that v satisfies φ, $(s,v) \xrightarrow{a} (s, v[\lambda := 0])$.

Based on this point, a run for A is defined by a state sequences starting from an initial state and triggered by an action (time-delay or locations-switch).

To investigate the entire behavior for a real-time system modeled by TA, the product TA is defined as follows.

Definition 5 Product Timed Automata

Let $A_1 = <L_1, L_1^0, S_1, \Sigma_1, C_1, Inv_1, Gad_1, Rst_1>$ and $A_2 = <L_2, L_2^0, \Sigma_2, E_2, C_2, Inv_2, Gad_2, Rst_2>$ be two TAs, assuming the clock sets C_1 and C_2 are disjoint. Then the product, denoted $A_1 \| A_2$, is the TA $< L_1 \times L_2, L_1^0 \times L_2^0, \Sigma_1 \cup \Sigma_2, E, C_1 \cup C_2, Inv, Gad, Rst >$, where $Inv(s_1, s_2) = Inv(s_1) \wedge Inv(s_2)$, the edges, guards and Resetting functions are defined by:

- For $a \in \Sigma_1 \cap \Sigma_2$, for every $e_1 = < s_1, a, s_1' >$ in E_1 and $e_2 = < s_2, a, s_2' >$ in E_2, E contains $e = < (s_1, s_2), a, (s_1', s_2') >$, $Gad(e) = Gad(e_1) \wedge Gad(e_2)$ and $Rst(e) = Rst(e_1) \cup Rst(e_2)$.
- For $a \in \Sigma_1 \setminus \Sigma_2$, for every $e_1 = < s, a, s' >$ in E_1 and every location t in L_2, E contains $e = < (s,t), a, (s',t) >$, $Gad(e) = Gad(e_1)$ and $Rst(e) = Rst(e_1)$.
- For $a \in \Sigma_2 \setminus \Sigma_1$, for every $e_2 = < s, a, s' >$ in E_2 and every location t in L_1, E contains $e = < (t,s), a, (t,s') >$, $Gad(e) = Gad(e_2)$ and $Rst(e) = Rst(e_2)$. ■

3 TA Theory Built on PVS

3.1 PVS

PVS [3] is a specification and verification environment developed by SRI International's Computer Science Laboratory. It provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing, modifying, managing, and documenting theories and proofs. In distinction to other widely used proof systems, such as HOL and the Boyer-Moore theorem prover, PVS supports both a highly expressive specification language and an interactive theorem prover in which most low-level proof steps are automated. The system consists of a specification language [4], a parser, a type checker, and an interactive proof checker [5]. The PVS specification language is based on a richly typed higher-order logic that permits a type checker to catch a number of semantic errors in specifications. The PVS prover consists of a set of inference steps that can be used to reduce a proof goal to simpler sub goals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps incorporate arithmetic and equality decision

procedures, automatic rewriting, and BDD-based boolean simplification.

Specifications in PVS consist of one or more theories. Each theory may be parameterized and may import other theories. In proving theorems in PVS, users can apply a sequence of primitive proof steps. In addition to primitive proof steps, PVS supports more complex proof steps called strategies, which can be invoked just like any other proof step. Strategies may be defined using primitive proof steps, applicative Lisp code, and other strategies, and may be built-in or user-defined. It is very convenient for us to formalize a system and prove the corresponding theorems.

3.2 The Formal Theory about Clocks in PVS

Because the clock is interpreted over nonnegative real numbers (called *Time*), we first give the theory for *Time* in Figure 2.

```
time: THEORY BEGIN
    Time: TYPE = nonneg_real
    PosTime: TYPE = posreal
END time
```

Figure 2. The Theory time

Where *nonneg_real* and *posreal* in the above theory is the type defined in the PVS prelude file library. Some syntax constructs' interpretation in the following PVS specification can be acquired in [4, 11]. The only role for this theory is simple for expressing the operation over clocks.

Next, we model the clock interpretations and clock constraints according to the Definition 1 and Definition 2 in section 2.1 using PVS specification language [4]. These two theories are shown in Figure 3 and 4. The codes that lie between the symbol "%" and the end of a line are comments in PVS.

```
clkinterp [ N: nat ] : THEORY  BEGIN
  IMPORTING time
  clock: TYPE = {j: upto(N) | j≠0}  %has N clocks, 1...N
  %clock interpretations and upper bound
  clkTime: TYPE = [clock → Time]
  clock_bound: TYPE = [clock → nat]
  v : VAR clkTime  x : VAR clock
  +(v, (t : Time)):clkTime=λx: v(x) + t %time-elapse
  reset(v)(R: pred[clock])(x): Time = IF R(x) THEN 0 ELSE
          v(x) ENDIF  %resetting clocks set
END clkinterp
```

Figure 3. The Theory clock interpretation

clock_bound in theory *clkinterp* shown in Figure 3 is a function that assigns to each clock $x \in C$ a natural number c such that x is compared with c in some clock constraint appearing in an invariant or a guard. It is necessary for establishing clock region equivalence theory, DBM (difference-bounded matrix) theory, etc.

The clock constraints (shown in Figure 4) set is defined as an abstract data type [10] in PVS. Its expressing power is equal to Definition 2. The semantics model "|=" for a clock constraint is recursively interpreted over a clock interpretation, where "<<" is a defined function reflecting the sub-term relation of that clock interpretation, and generated automatically by PVS.

```
clkp[N: nat, (importing clkinterp[N])
      cb: clock_bound]: THEORY   BEGIN
      le_ub(x: clock): TYPE = {i: nat | i ≤cb(x)}
      %======Syntax construct============
      clkp: DATATYPE    BEGIN
         ≤ (x:clock, c: le_ub(x)): le?
         ≥ (x:clock, c: le_ub(x)): ge?
         < (x:clock, nzc: {i:le_ub(x)|i≠0}: lt?
         > (x:clock, c: le_ub(x)): gt?
         = (x:clock, c: le_ub(x)): eq?
         TRUE              : true?
         AND (f1, f2: clkp)     : and?
         END clkp
      %=====Semantics interpretation========
       |=(v, (f:clkp)): RECURSIVE bool = CASES f OF
           ≤ (x, c): v(x) ≤ c,
           ≥ (x, c): v(x) ≥ c,
          < (x, c): v(x) < c,
           > (x, c): v(x) > c,
            = (x, c): v(x) = c,
            TRUE: TRUE,
          AND(f1, f2): (v |= f1) AND (v |= f2)
        ENDCASES  MEASURE (λv, f: f) BY <<;
   END clkp
```

Figure 4. The Theory clock constraints

```
clk_cons [M, N: nat]: THEORY  BEGIN
    IMPORTING clkinterp
   vx: VAR clkTime[M]
    vy: VAR clkTime[N]
    ^(vx, vy)(z: clock[M + N]): Time= IF z ≤ M THEN vx(z)
            ELSE vy(z - M) ENDIF;
   union(px: set[clock[M]], py: set[clock[N]]): set[clock[M +
       N]] = λ (z: clock[M + N]) : IF z ≤ M THEN px(z) ELSE
       py(z - M) ENDIF
END clk_cons
```

Figure 5. The Theory concatenation of clock interpretations

Let A_1 and A_2 be two TAs with clocks set C_1 and C_2, which are numbered 1, …, M and 1,…, N respectively, and C_1 and C_2 are disjoint. According to the Definition 5, If we want to get the product construct $A = A_1||A_2$, the new clocks set $C= C_1 \cup C_2$ for A should be acquired, meanwhile the clocks in C be numbered 1,…, $M+N$, moreover, the clocks number 1,…, M have the same behavior with the corresponding one in C_1, and that numbered M+1,…, $M+N$ have the same behavior as that numbered 1,…, N in C_2. This process can be implemented in theory *clk_cons* and *clkp_cons* shown in Figure 5 and 6.

```
clkp_cons[M, N: nat, (IMPORTING clkinterp)
 c1: clock_bound[M], c2: clock_bound[N]]: THEORY
   BEGIN
     IMPORTING clk_cons[M, N]
     IMPORTING clkp[M, c1] AS P1, clkp[N, c2] AS P2,
                   clkp[M + N, c1 ^ c2] AS P12
     f1: VAR P1.clkp
     f2: VAR P2.clkp
     extend(f1): RECURSIVE P12.clkp = CASES f1 OF
         ≤ (x, c): P12. ≤(x, c),   % type conversions
         ≥(x, c): P12.≥ (x, c),
         <(x, c): P12.<(x, c),
         >(x, c): P12.>(x, c),
         =(x, c): P12.=(x, c),
         TRUE: TRUE,
         AND(f_1, f_2): extend(f_1) AND extend(f_2)
       ENDCASES MEASURE id BY <<;
     extend(f2): RECURSIVE P12.clkp =  CASES f2
       OF ≤(x, c): P12.≤ (x + M, c),
         ≥ (x, c): P12.≥ (x + M, c),
         <(x, c): P12.<(x + M, c),
         >(x, c): P12.>(x + M, c),
         =(x, c): P12.=(x + M, c),
         TRUE: TRUE,
         AND(f_1, f_2): extend(f_1) AND extend(f_2)
       ENDCASES MEASURE id BY <<;
       ^(f1, f2): P12.clkp = extend(f1) AND extend(f2)
 END clkp_cons
```

Figure 6. The Theory concatenation of clock constraints

Here the overloaded infix operator "^" can combine two clock constraints expressions (or two clock interpretation variable) of different type into one new type of clock constraints expression (or clock interpretation variable). It is noted that "^" is associative but not commutative.

All the proof obligations, i.e., TCCs (Type Correctness Condition) judged and generated by PVS automatically, for the above several theories, can be successfully proved via the prover invocation command TCP (Type-Check Prove) [3] by PVS itself.

3.3 TA Theory in PVS

According to the formal definition A=<L, L^0, Σ, E, C, Inv, Gad, Rst> of TA in Definition 3 and the corresponding transition system in Definition 4, we model *TimedAutomaton* theory in PVS. This theory shown in Figure 7 has nine parameters, where *Locations, Actions, Init, Inv, Edge, Guard,* and *ResetC* correspond with the components L, Σ, L^0, Inv, E, C, Gad and Rst of A respectively. The other two parameters N and c represent the clocks number and the function that map each clock to the maximum *Time* value occurred in *Inv* or *Gad* respectively. It is worthy noted that some functions about these arguments are implemented by importing the theories defined in section 3.2. The operator "`", which occurs followed by a variable or a constant in PVS specification, is the field accessor of a tuple or a record type.

```
 TimedAutomaton[Locations, Actions: TYPE, N: nat,
        (IMPORTING clkinterp[N]) c: clock_bound,
         (IMPORTING clkp[N, c]) Init: pred[Locations],
         Inv: [Locations → clkp],
         Edge: pred[[Locations, Actions, Locations]],
         Guard: [(Edge) → clkp],
          ResetC: [(Edge) → pred[clock]]]: THEORY BEGIN
    States: TYPE=[# loc: Locations, v: clkTime, now: Time #];
    PreRuns: TYPE = [# states: sequence[States],
                     events: sequence [Actions] #];
     pr: VAR PreRuns
     t: VAR Time
     d: VAR PosTime
     s, s0, s1: VAR States
     i, j: VAR nat
     A: VAR Actions
     +(s, t): States = s WITH [v := s`v + t, now := s`now + t]
     delta(s0, d, s1): bool=    s1 = s0 + d &
            (s0`v |= Inv(s0`loc)) & (s1`v |= Inv(s1`loc))
     locswitch(s0, A, s1): bool=LET sw =(s0`loc, A, s1`loc) IN
       Edge(sw) & (s0`v |= (Inv(s0`loc) AND Guard(sw))) &
          s1 = s0 WITH [v := reset(s0`v)(ResetC(sw))] &
            (s1`v |= Inv(s1`loc))
     Step(s0, s1): bool = (∃d: delta(s0, d, s1)) OR
                             (∃A: locswitch(s0, A, s1))
     Init(s): bool = Init(s`loc) AND s`v = (λx: 0) & s`now = 0
     NoTimeDecrease(pr): bool = ∀i:
                     now(states(pr)(i)) ≤ now(states(pr)(i + 1))
     NonZeno(pr): bool = ∀ t: ∃ i: t < now(states(pr)(i))
      run?(pr): bool =  Init(pr`states(0)) &
             (∀i: Step( pr`states(i), pr`states(i + 1))) &
            NoTimeDecrease(pr) & NonZeno(pr)
     Runs: TYPE=(run?)
 END TimedAutomaton
```

Figure 7. The Theory TimedAutomaton

The *States, delta, locswitch*, and *Runs* Defined in theory T*imedAutomaton* intend to define the semantics for TA over the corresponding transition system. The definition for type *Runs*, which is the form of $s_0 \rightarrow a_0(\delta) \rightarrow s_1 \rightarrow a_1(\delta) \rightarrow \ldots$, is more complex than that of others, and based upon the following conditions:

- It starts from the initial state.
- Every state in the run is triggered by some action.
- It never decreases the global time.
- It runs forever.

The runs defined here exclude all the pathological behaviors of TA.

Based on this theory, we can further define the properties, such as some invariant properties and deadlock freedom, over *States* or *Runs*. These properties, i.e., some lemmas (or theorems), which assert some behaviors of a real-time system, will form the infrastructure for the formal verification for our system FVofTA.

Using the theory *TimedAutomaton*, we can easily give the PVS specification for a real-time system modeled by TA. The template shown in Figure 8 can help to complete this process to some extent. The template is instantiated by filling in the missing parts and adding any desired auxiliary declarations and definitions. The missing parts are represented in Figure 8 by the symbol "<…>".

```
<timed-automaton name>: THEORY
BEGIN
N: nat = <…>
Locations: TYPE =<…>
Actions : TYPE =<…>
IMPORTING clkinterp[N]
xx: VAR clock
c: clock_bound =λxx: <..>
IMPORTING clkp[N, c]
l,l1: var Locations
A: VAR Actions
Init(l:Locations): bool = <…>
Inv(l:Locations): clkp = <…>
Edge(l, A, l1): bool =<…>
sw: VAR (Edge)
Guard(sw): clkp = <…>
ResetC(sw): pred[clock] =<…>
IMPORTING TimedAutomaton [Locations, Actions, N, c,
                Init, Inv, Edge, Guard, ResetC]
END <timed-automaton name>
```

Figure 8. The Template for Theory TimedAutomaton

After several components have created by the above template, the product construct can be easily acquired using the flowing template derived from Definition 5. The main works for product construction include:

- constructing the tuple for locations,
- forming the new predicate *Init* which expresses the initial locations using the conjunctions of the original ones,
- combining the edges,
- forming the new invariants and guards using the conjunctions of the original ones,
- acquiring the clocks union of original ones, and
- importing the Theory *TimedAutomaton*.

The component *Actions* in the product is shared in the several components. If one action *a* doesn't occur in any edge for one of component, it is regard as a stuttering action, i.e., (*l, a, l*) is an edge of that TA for any location *l*. All the above theories and the templates form the basis (modeling part, i.e., the bottom four rectangles in Figure 1) of our mechanized system FVofTA. The works for the proof part (the top four rectangles in Figure 1) are under development.

```
<Prod name>: THEORY
 BEGIN
  IMPORTING < timed-automaton name1 > AS TA1
   …
  IMPORTING < timed-automaton namen > AS TAn
   Locations: TYPE = [TA1.Locations, …, TAn.Locations]
  A: VAR Actions
  ProdN: posnat = TA1.N + … + TAn.N
  l, l0: VAR Locations
  IMPORTING clk_cons
  cb: clock_bound[ProdN] = TA1.c ^ TA2.c ^…^ TAn.c
  Init(l): bool = TA1.Init(l`1) & …TAn.Init(l`n)
  IMPORTING clkp_cons
  Inv(l): clkp[ProdN, cb] = TA1.Inv(l`1) ^ … ^ TAn.Inv(l`3)
  Edge(l0, A, l): bool =  Edge(l0`1, A, l`1)  & …
                            & Edge(l0`n, A, l`n)
  sw: VAR (Edge)
  Guard(sw): clkp[ProdN, cb] = LET (l0, A, l) = sw IN
      (TA1.Guard(l0`1, A, l`1) ^…TAn.Guard(l0`n, A, l`n)
  ResetC(sw): set[clock[ProdN]] = LET (l0, A, l) = sw IN
    union(…(union(TA1.ResetC(l0`1, A, l`1),
      TA2.ResetC(l0`2, A, l`2)),…TAn.ResetC(l0`n, A, l`n)))
   IMPORTING TimedAutomaton [Locations, Actions, ProdN,
                   cb, Init, Inv, Edge, Guard, ResetC]
 END <Prod name>
```

Figure 9. The Template for product timed automata

4 Case study: Train-Gate-Controller

4.1 TGC introduction

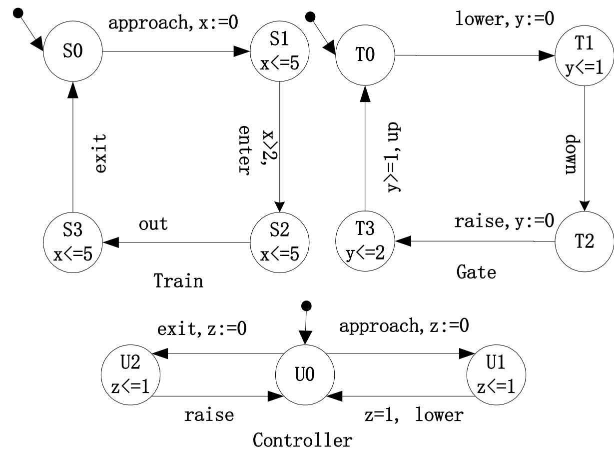

Figure 10. Train-gate controller

A well-known TA modeling example is an automatic controller [8] that opens and closes a gate at a railroad crossing. The system is composed of three components: TRAIN, GATE and CONTROLLER as shown in Figure 10. The train is required to send the signal *approach* to the controller at least 2 minutes before it enters the crossing

(expressed by the guard x>2 associated with the event *enter*). Furthermore, the maximum delay between the signal *approach* and *exit* is 5 minutes (expressed with the clock constraints in locations *S*1, *S*2 and *S*3). When the train leaves out the crossing, it will send the signal *exit* to the controller. Whenever the controller receives the signal *approach,* it responds by sending the signal *lower* to the gate. Whenever it receives the signal *exit,* it responds by sending the signal *raise* to the gate. The response time to signal *approach* is 1 minute, and to the signal *exit* is at most 1 minute (this is expressed with the clock constraint for clock *z*). Correspondingly, the gate responds to the signal *lower* by closing within 1 minute, and response to the signal *raise* within 1 to 2 minutes. The system abbreviates TGC system.

4.2 Modeling TGC system in PVS

Now we model TGC system using templates given in Figure 8 and 9. First, we give the theory for the three components using the TimedAutomaton template in Figure 8, and then acquire the product construction via the template in Figure 9.

```
TGCActions: theory   BEGIN
TGCActions: TYPE={approach, up, enter, out, exit, lower, down}
END TGCActions
```

Figure 11. The Theory TGCActions

This system has eight actions that can be defined as a separated single theory in order to be shared in the three components.

```
Train: THEORY BEGIN
  N: nat = 1
  IMPORTING clkinterp[N], TGCActions
  Locations: TYPE = {S0, S1, S2, S3}
  Actions: TYPE = TGCActions
  l, l1: VAR Locations
  A: VAR Actions
  xx: VAR clock
  c: clock_bound = λ xx: 5
  IMPORTING clkp[N, c]
  x: MACRO clock = 1
  Init(l): bool = l = S0
  Inv(l): clkp = IF l ≠ S0 THEN x≤ 5 ELSE TRUE ENDIF;
  Edge(l, A, l1): bool = CASES A
     OF approach: l = S0 AND l1 = S1,
       enter: l = S1 AND l1 = S2,
       out: l = S2 AND l1 = S3,
       exit: l = S3 AND l1 = S0
       ELSE l = l1     ENDCASES
 sw: VAR (Edge)
Guard(sw): clkp = IF sw`2 = enter THEN x > 2 ELSE TRUE
                  ENDIF;
ResetC(sw): pred[clock] =   IF sw`2 = approach THEN {xx |
     xx = x} ELSE emptyset[clock] ENDIF
 IMPORTING  TimedAutomaton[Locations,  Actions,  N,  c,
     Init, Inv, Edge, Guard,   ResetC]
 END Train
```

Figure 12. The Theory Train

The theory *Train* shown in Figure 12 is the model of the component TRAIN. The underlined codes, which are very trivial information, are written by the modeler, while the rest are same as the TA template shown in Figure 8.

In the same way, the component GATE and CONTROLLER can also be acquired.

The theory for modeling TGC system can be easily created using the template for product construction. By comparing the codes in Figure 13 with those are in Figure 9, it is shown that, except the theory name, the differences between this theory and the template product timed automata are only the front sections.

Until now, the family of the theories in PVS for modeling the real-time system TGC is modeled.

No unproved proof obligations are generated for these theories.

```
TGC: THEORY   BEGIN
   IMPORTING Train AS T, Gate AS G, Controller AS C
   Locations: TYPE =[T.Locations, G.Locations, C.Locations]
  A: VAR Actions
  ProdN: posnat = T.N + G.N + C.N
  l, l0: VAR Locations
  IMPORTING clk_cons
  x: VAR clock[ProdN]
  cb: clock_bound[ProdN] = (T.c ^ G.c) ^ C.c
  Init(l): bool = T.Init(l`1) AND G.Init(l`2) AND C.Init(l`3)
  IMPORTING clkp_cons
  Inv(l): clkp[ProdN, cb]=(T.Inv(l`1) ^ G.Inv(l`2)) ^ C.Inv(l`3)
  Edge(l0, A, l): bool= Edge(l0`1, A, l`1) &
                    Edge(l0`2, A, l`2) & Edge(l0`3, A, l`3)
  sw: VAR (Edge)
  Guard(sw): clkp[ProdN, cb] = LET (l0, A, l) = sw IN
      (T.Guard(l0`1, A, l`1) ^ G.Guard(l0`2, A, l`2)) ^
       C.Guard(l0`3, A, l`3)
  ResetC(sw): set[clock[ProdN]] = LET (l0, A, l) = sw IN
     union(union(T.ResetC(l0`1, A, l`1),
        G.ResetC(l0`2, A, l`2)),      C.ResetC(l0`3, A, l`3))
  IMPORTING TimedAutomaton [Locations, Actions, ProdN,
                        cb, Init, Inv, Edge, Guard, ResetC]
 END TGC
```

Figure 13. The Theory TGC System

4.3 Formal verification of TGC

The main correctness requirements for the TGC system are [2]:

(1) *Safety*: Whenever the train is inside the gate, the gate should be closed.

(2) *Real-time Liveness*: The gate is never closed at a stretch for more than 10 minutes.

The first property can be formalized in TGC theory using PVS specifications:

safe(s: states): bool = s`loc`1 = S2 => s`locl`2 = T2

Safety : THEOREM (∀r: Runs): let s = states(r)(i) in safe(s)

Here *Safety* is the Theorem 5 in [12], its proof can also be seen in [12].

As to the formal verification of the second property *Real-time Liveness*, there has an extra step to convert its negation to a tester TA, and then prove that the negation is false. The converting step and the subsequent proof can also be seen in [12].

From the results about the verifications of the two properties, we can conclude that our formalizations about clock and TA syntax and semantics in our framework are solid.

5 Related Works

One of the research work about formal verification for real-time system based-on timed automata model using theorem proving method had been conducted in TAME (Timed Automata Modeling Environment) project [1] in US Naval Research Laboratory. TAME provides mechanical assistance that allows humans to specify and reason about real-time systems in a direct manner. Nevertheless, TAME doesn't supply some operation on clocks, such as clocks concatenations in our theory *clkp_cons*. Therorefore, it is inconvenient and unnatural to use TAME especially in proving some properties about clocks of TA. Another research work, which is conducted by Jozef Hooman [9], gives the TA model theory and some lemmas about the corresponding runs of TA in PVS. The runs' definition in FVofTA has been inspired by this work.

The common shortcoming for both TAME and the works of Jozef Hooman is that they both don't provide the manipulation for the clocks, a very important notion in TA theory. Therefore, some operations on TA, such as clock region-equivalence or time-abstract application in terms of clocks, and the formal verifications of some properties defined over the clock variable become unnatural and difficult.

In fact, the theories they give are modeling the transition system associated with TA rather than TA itself.

6 Conclusions and Future Works

This paper gives a real-time system modeling framework based on TA with PVS and investigates a real-time system. This framework can conveniently model some real-time system in the form of TA model and conduct some formal verification for the safety properties.

The future works about FVofTA are the formal verification about TA. They mainly include the following parts:

(1) We want to establish the timed temporal logic framework based on the TA modeling theory.

(2) Some techniques used in model checking [6], such as region-equivalence and timed-abstract, etc., will be adopted in this system framework.

(3) The automatic translation from timed temporal logical formula into TA theory should be implemented in PVS.

(4) The model checking algorithms will be encoded in PVS, furthermore, the correctness for them will be ensured by the PVS proof system.

7 References

[1] M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. *1996 IEEE Real-Time Systems Symp*., Washington, D.C., 1996,3-6,.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science* 126:183-235, 1994.

[3] Owre S, Shankar N, Rushby J, Stringer-Calvert D. PVS system guide version 3.2. Computer Science Laboratory, SRI International, September, 2004, 1–95.

[4] Owre S, Shankar N, Rushby J, Stringer-Calvert D. PVS language reference version 3.0. Computer Science Laboratory, SRI International, February, 2003, 1–123.

[5] Shankar N, Owre S, Rushby J, Stringer-Calvert D. PVS prover guide version 3.2. Computer Science Laboratory, SRI International, September, 2004, 1–128.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Cambridge: MIT PRESS, 1999.

[7] Archer, M., TAME: Using PVS Strategies for Special-Purpose Theorem Proving, A*nnals of Mathematics and Artificial Intelligence* 29(1-4), 2000, 139-181.

[8] R. Alur. Timed Automata. *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems.*

[9] Jozef Hooman. Timed Automata in PVS. Summerschool Zhengzhou, 2004.

[10] Owre S, Shankar N. Abstract Datatypes in PVS. Computer Science Laboratory, SRI International, 1997,1-52.

[11] Owre S, Shankar N. The PVS Prelude Library.: Computer Science Laboratory, SRI International, 2003,1-31.

[12] Qingguo Xu, Huaikou Miao. Formal Verification Framework for safety of Real-time System Based-on Timed Automata Model in PVS, the IASTED International Conference on Software Engineering, Innsbruck, Austria, February, 2006,107-112.