Generating Path Conditions for Timed Systems

Saddek Bensalem[1], Doron Peled[2,⋆], Hongyang Qu[2], and Stavros Tripakis[1]

[1] Verimag, 2 Avenue de Vignate, 38610 Gieres, France
[2] Department of Computer Science, University of Warwick, Coventry, CV4 7AL United Kingdom

Abstract. We provide an automatic method for calculating the path condition for programs with real time constraints. This method can be used for the semiautomatic verification of a unit of code in isolation, i.e., without providing the exact values of parameters with which it is called. Our method can also be used for the automatic generation of test cases for unit testing. The current generalization of the calculation of path condition for the timed case turns out to be quite tricky, since not only the selected path contributes to the path condition, but also the timing constraints of alternative choices in the code.

1 Introduction

Software testing often involves the use of informal intuition and reasoning. But it is possible to employ some formal methods techniques and provide tools to support it. Such tools can help in translating the informal ideas and intuition into formal specification, assist in searching the code, support the process of inspecting it and help analyzing the results. A tester may have a vague idea where problems in the code may occur. The generation of a condition for a generated suspicious sequence may help the tester to confirm or refute such a suspicion. Such a condition relates the variables at the beginning of the sequence. Starting the execution with values satisfying this condition is necessary to recreate the execution.

We generalize the calculation of a path condition, taking into account only the essential conditions to follow a particular path in the execution. We start with a given path merely from practical consideration; it is simpler to choose a sequence of program statements to execute. However, we look at the essential partial order, which is consistent with the real-time constraints, rather than at the total order. We cannot assume that transitions must follow each other, unless this order stems from some sequentiality constraints such as transitions belonging to the same process or using the same variable or from timing constraints.

For untimed systems, there is no difference between the condition for the partial order execution and the condition to execute any of the sequences (linearizations) consistent with it. Because of commutativity between concurrently

⋆ This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin's prime National Science Foundation Grant #CCR-0205483.

J. Romijn, G. Smith, and J. van de Pol (Eds.): IFM 2005, LNCS 3771, pp. 5–19, 2005.

executed transitions, we obtain the same path condition either way. However, when taking the time constraints into account, the actual time and order between occurrences of transitions does affect the path condition (which now includes time information).

After the introduction of the untimed path condition in [3], weakest precondition for timed system was studied in [2, 8, 9]. The paper [2] extended the guarded-command language in [3] to involve time. But it only investigated sequential programs with time constraints. The paper [9] gave a definition of the weakest precondition for concurrent program with time constraints, based on discrete time, rather than dense time. The weakest precondition in [8] is defined for timed guarded-command programs or, alternatively, timed safety automata.

We model concurrent systems using timed transition systems. Our model is quite detailed in the sense that it separates the decision to take a transition (the enabling condition) from performing the transformation associated with it. We allow separate timing constraints (lower and upper bounds) for both parts. Thus, we do not find the model proposed in [7], which assigns a lower and upper time constraints for a transition that includes both enabling transition and a transformation, detailed enough. Alternative choices in the code may compete with each other, and their time constraints may affect each other in quite an intricate way. Although we do not suggest that our model provides the only way for describing a particular real-time system, it is detailed enough to demonstrate how to automatically generate test cases for realistic concurrent real-time systems.

In our solution, we translate the timed transition system into a collection of extended timed automata, which is then synchronized with constraints stemming from the given execution sequence. We then obtain a directed acyclic graph of executed transitions. We apply to it a weakest precondition construction, enriched with time analysis based on time zone analysis (using difference bound matrices).

2 Modeling Concurrent Timed Systems

As mentioned in the introduction, we describe concurrent real-time systems using timed transition systems (TTS). The latter model is given a semantics in terms of extended timed automata (ETA). This is done by defining a modular translation where each process in the TTS model is translated into an ETA. Thus the entire TTS model is translated into a network of synchronizing ETA. This section defines the two models and the translation.

2.1 Timed Transition Systems

We consider *timed transition systems* over a finite set of processes $P_1 \ldots P_n$. Each process consists of a finite number of transitions. The transitions involve checking and updating control variables and program variables (over the integers). An enabling condition is an assertion over the program variables. Although the

processes are not mentioned explicitly in the transitions, each process P_i has its own location counter loc_i. It is possible that a transition is jointly performed by two processes, e.g., a synchronous communication transition. We leave out the details for various modes of concurrency, and use as an example a model that has only shared variables.

A transition t includes (1) an enabling condition c, (2) an assertion over the current process P_j location, of the form $loc_j = \hat{l}$, (3) a transformation f of the variables, and (4) a new value $\hat{l}'$ for the location of process P_j. For example, a test (e.g., `while` loop or `if` condition) from a control value $\hat{l}$ of process P_j to a control value $\hat{l}'$, can be executed when $(loc_j = \hat{l}) \wedge c$, and result in the transformation f being performed on the variables, and $loc_j = \hat{l}'$.

We equip each transition with two pairs of time constraints $[l, u], [L, U]$ such that:

- l is a lower bound on the time a transition needs to be *continuously* enabled until it is selected.
- u is an upper bound on the time the transition can be *continuously* enabled without being selected.
- L is a lower bound on the time it takes to perform the transformation of a transition, after it was selected.
- U is the upper bound on the time it takes to perform the transformation of a transition, after it was selected.

We allow shared variables, but make the restriction that each transition may change or use at most a single shared variable.

Every process can be illustrated as a directed graph G. A location is represented by a node and a transition is represented by an edge. Figure 1 shows the graphic representation of a transition.

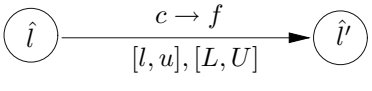

Fig. 1. The edge

2.2 Extended Timed Automata

An *extended timed automaton* is a tuple $\langle V, X, B, F, S, S^0, \Sigma, Cl, E\rangle$ where

- V is a set of variables.
- X is a finite set of assertions over the set of variables V.
- B is a set of Boolean combinations of assertions over clocks of the form $x \# \hat{c}$, where x is a clock, $\#$ is a relation from $\{<, >, \geq, \leq, =\}$ and $\hat{c}$ is a constant (not necessarily a value, as our timed automaton can be parameterized).
- F is a set of transformations for the variables. Each component of F can be represented e.g., as a multiple assignment to some of the variables in V.

- S is a finite set of states.[1] A state $s \in S$ is labeled with an assertion s^X from X and an assertion s^B on B that need to hold invariantly when we are at the state.
- $S^0 \subseteq S$ are the initial states.
- Σ is a finite set of labels.
- Cl is a finite set of clocks.
- E the set of edges over $S \times 2^{Cl} \times \Sigma \times X \times B \times F \times S$. The first component of an edge $e \in E$ is the source state. The second component e^{Cl} is the set of clocks that reset to 0 upon firing this edge. A label e^{Σ} from Σ allows synchronizing edges from different automata, when defining the product. We allow multiple labels on edges, as a terse way of denoting multiple edges. An edge e also includes an assertion e^X over the variables, an assertion e^B over the time variables that has to hold for the edge to fire, a transformation e^F over the variables and a target state.

The above definition extends timed automata [1] by allowing conditions over variables to be associated with the edges and states, and transformations on the variables on the edges (similar to the difference between finite state machines and extended finite state machines).

Semantics. The semantics of extended timed automata is defined as a set of executions. An *execution* is a (finite or infinite) sequence of triples of the form $\langle s_i, V_i, T_i \rangle$, where

1. s_i is a state from S,
2. V_i is an assignment for the variables V over some given domain(s), such that $V_i \models s_i^X$ and
3. T_i is an assignment of (real) time values to the clocks in Cl such that $T_i \models s_i^B$.

In addition, for each adjacent pair $\langle s_i, V_i, T_i \rangle$ $\langle s_{i+1}, V_{i+1}, T_{i+1} \rangle$ one of the following holds:

An edge is fired. There is an edge e from source s_i to target s_{i+1}, where $T_i \models e^B$, $V_i \models e^X$, T_{i+1} agrees with T_i except for the clocks in e^{Cl}, which are set to zero, and $V_{i+1} = e^F(V_i)$, where $e^F(V_i)$ represents performing the transformation over V_i.

Passage of time. $T_{i+1} = T_i + \delta$, i.e., each clock in Cl is incremented by some real value δ. Then $V_{i+1} = V_i$.

An infinite execution must have an infinite progress of time. An initialized execution must start with $s \in S^0$ and with all clocks set to zero. However for generation of test cases we deal here with finite consecutive segments of executions, which do not have to be initialized.

[1] We use the term "state" for extended timed automata to distinguish from "location" for timed transition systems.

The Product of ETA. Let $ETA_1 = \langle V_1, X_1, Cl_1, B_1, F_1, S_1, S_1^0, \Sigma_1, E_1\rangle$ and $ETA_2 = \langle V_2, X_2, Cl_2, B_2, F_2, S_2, S_2^0, \Sigma_2, E_2\rangle$ be two ETAs. Assume the clock sets Cl_1 and Cl_2 are disjoint. Then the product, denoted $ETA_1 \parallel ETA_2$, is the ETA $\langle V_1 \cup V_2, X_1 \cup X_2, Cl_1 \cup Cl_2, B_1 \cup B_2, F_1 \cup F_2, S_1 \times S_2, S_1^0 \times S_2^0, \Sigma_1 \cup \Sigma_2, E\rangle$. For a compound state $s = (s_1, s_2)$ where $s_1 \in S_1$ with $s_1^{X_1} \in X_1$ and $s_1^{B_1} \in B_1$ and $s_2 \in S_2$ with $s_2^{X_2} \in X_2$ and $s_2^{B_2} \in B_2$, $s^{X_1 \cup X_2} = s_1^{X_1} \wedge s_2^{X_2}$ and $s^{B_1 \cup B_2} = s_1^{B_1} \wedge s_2^{B_2}$. The set E of edges are defined as follows. For every edge $e_1 = \langle s_1, e_1^{Cl_1}, e_1^{\Sigma_1}, e_1^{X_1}, e_1^{B_1}, e_1^{F_1}, s_1'\rangle$ in E_1 and $e_2 = \langle s_2, e_2^{Cl_2}, e_2^{\Sigma_2}, e_2^{X_2}, e_2^{B_2}, e_2^{F_2}, s_2'\rangle$ in E_2,

- joint edges: if $e_1^{\Sigma_1} \cap e_2^{\Sigma_2} \neq \emptyset$, E contains $\langle (s_1, s_2), e_1^{Cl_1} \cup e_2^{Cl_2}, e_1^{\Sigma_1} \cup e_2^{\Sigma_2}, e_1^{X_1} \wedge e_2^{X_2}, e_1^{B_1} \wedge e_2^{B_2}, e_1^{F_1} \cup e_2^{F_2}, (s_1', s_2')\rangle$. Any variable is allowed to be assigned to a new value by either e_1 or e_2, not both.
- edges only in ETA_1 or ETA_2: if $e_1^{\Sigma_1} \cap e_2^{\Sigma_2} = \emptyset$, E contains $\langle (s_1, s''), e_1^{Cl_1}, e_1^{\Sigma_1}, e_1^{X_1}, e_1^{B_1}, e_1^{F_1}, (s_1', s'')\rangle$ for every state $s'' \in S_2$ and $\langle (s', s_2), e_2^{Cl_2}, e_2^{\Sigma_2}, e_2^{X_2}, e_2^{B_2}, e_2^{F_2}, (s', s_2')\rangle$ for every state $s' \in S_1$.

2.3 Translating a TTS into ETAs

We describe the construction of a set of extended timed automata from a timed transition system. We should emphasize that this construction *defines* the semantics of a timed transition system as the semantics of the corresponding set of extended timed automata.

We first show how to construct states and edges for one particular location. An ETA is generated after all locations in a TTS process are translated. Any location in a process is said to be the *neighborhood* of the transitions that must start at that location. The enabledness of each transition depends on the location counter, as well as an enabling condition over the variables. Location counters are translated in an implicit way such that each different location is translated into a different set of states. For a neighborhood with n transitions $t_1, \ldots, t_n$, let $c_1, \ldots, c_n$ be the enabling conditions of n transitions respectively. The combination of these conditions has the form of

$$C_1 \wedge \ldots \wedge C_n,$$

where C_i is c_i or $\neg c_i$. Each transition t_j in the neighborhood has its own local clock x_j. Different transitions may have the same local clocks, if they do not participate in the same process or the same neighborhood.

1. we construct 2^n *enabledness* states, one for each Bool-ean combination of enabling condition truth values. For any enabledness states s_i and s_k (note that s_i and s_k can be the same state), there is an *internal* edge starting at s_i and pointing to s_k. Let $\mathcal{C}_i$ and $\mathcal{C}_k$ be the combinations for s_i and s_k, respectively. The edge is associated with $\mathcal{C}_k$ as the assertion over the variables. For any condition C_j which is $\neg c_j$ in $\mathcal{C}_i$ and c_j in $\mathcal{C}_k$, the clock x_j is reset ($x_j := 0$) upon the edge, for measuring the amount of time that the corresponding transition is enabled. We do not reset x_j in other cases.

2. We also have an additional *intermediate* state per each transition in the neighborhood, from which the transformation associated with the selected transition is performed. For any enabledness state s with the combination $\mathcal{C}$ in which the condition C_j corresponding to the transition t_j is c_j, let s'_j be the intermediate state for t_j and do the following:
   (a) We have the conjunct $x_j < u_j$ as part of s^X, the assertion over the variable of s, disallowing t_j to be enabled in s more than its upper limit u_j.
   (b) We add a *decision* edge with the assertion $x_j \geq l_j$ from s, allowing the selection of t_j only after t_j has been enabled at least l_j time continuously since it became enabled. On the decision edge, we also reset the clock x_j to measure now the time it takes to execute the transformation.
   (c) We put the assertion $x_j < U_j$ into s'_j, not allowing the transformation to be delayed more than U_j time.
   (d) We add an additional *transformation* edge labeled with $x_j \geq L_j$ and the transformation of t_j to from s' any of the enabledness states representing the target location of t_j. Again, this is done according to the above construction. There can be multiple such states, for the successor neighborhood, and we need to reset the appropriate clocks. We add an assertion over variables to the transformation edge. The assertion is the combination of enabling conditions which is associated to the target state of the transformation edge.

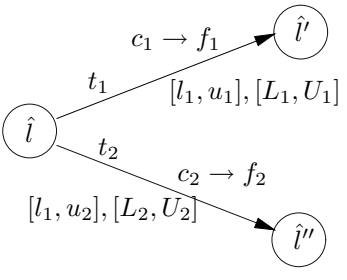

Fig. 2. A neighborhood of two TTS transitions

Figure 2 illustrates a neighborhood with two transitions and Figure 3 provides the ETA construction for this neighborhood. The states s_1, s_2, s_3 and s_4 are enabledness states, corresponding to the subset of enabling conditions of t_1 and t_2 that hold in the current location $\hat{l}$. The edges to s_5 correspond to t_1 being selected, and the edges to s_6 correspond to t_2 being selected. The edges into s_5 also reset the local clock x_1 that times the duration of the transformation f_1 of t_1, while the edges into s_6 zero the clock x_2 that times the duration of f_2. The state s_5 (s_6, respectively) allows us to wait no longer than U_1 (U_2, resp.) before we perform t_1 (t_2). The edge from s_5 (s_6) to s_8 (s_8) allows delay of no less than L_1 (L_2) before completing t_1 (t_2). Note that s_7 (as well as s_8) actually represents

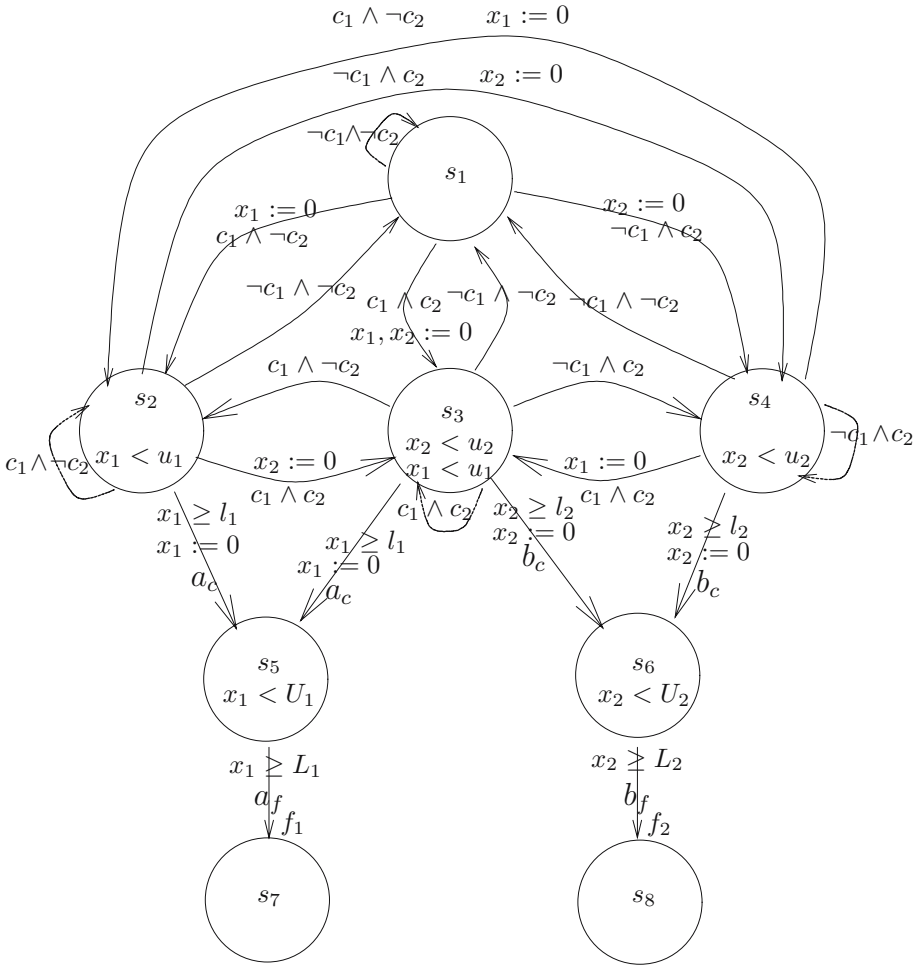

Fig. 3. The ETA for the neighborhood of two TTS transitions

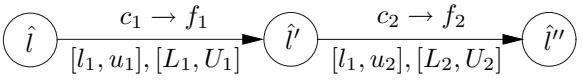

Fig. 4. Two sequential TTS transitions

one of a set of enabledness states, in the pattern of s_1 to s_4, for the location $\hat{l}'$ ($\hat{l}''$, resp), according to the enabledness of transitions in it (depending on the enabledness of the various transitions in the new neighborhood and including the corresponding reset of enabledness measuring clocks).

Figure 4 shows two consecutive transitions and Figure 5 provides the ETA construction for these transitions. For simplicity, the self loops are omitted. Lo-

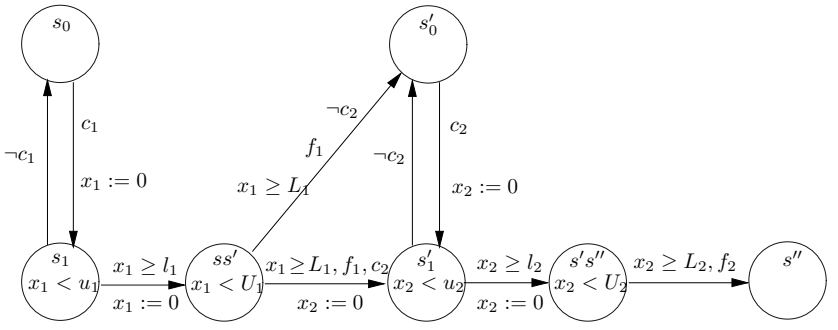

Fig. 5. The ETA for the two sequential TTS transitions

cation $\hat{l}$ is translated into states s_0 and s_1, location $\hat{l}'$ into s'_0 and s'_1, and location $\hat{l}''$ into s''. States ss' and $s's''$ are intermediate states.

2.4 Modeling Shared Variables

We present the procedure of modeling shared variables in a mutual exclusion manner. A shared variable needs to be protected by mutual exclusion when two or more transformations attempt to write it concurrently. For each shared variable v we provide a two state automaton, $\{used, unused\}$. We synchronize the decision edge of each transition that references such a variable with an edge from *unused* to *used*, and each transformation edge of such a transition with an edge from *used* to *unused*. When a decision edge acquires v, all other processes accessing v are forced to move to corresponding states by synchronizing the decision edge with proper internal edges in those processes. For the same reason, a transformation releasing v is synchronized with relative edges to enable accessing v.

3 Calculating the Path Conditions

In order to compute the path condition, the first step of our method involves generating an acyclic ETA (which we will call a DAG, or *directed acyclic graph*). Then the path condition is computed by propagating constraints backwards in this DAG. The DAG is generated using, on one hand, the set of ETAs corresponding to the TTS in question, and on the other hand, the TTS path (i.e., program transition sequence) provided by the user.

3.1 The Partial Order in a TTS Path

Given a selected sequence σ of occurrences of transitions, we calculate the *essential* partial order, i.e., a transitive, reflexive and asymmetric order between the execution of the transitions, as described below. This essential partial order is represented as a formula over a finite set of *actions* $Act = A_c \cup A_f$,

where the actions A_c represent the selections of transitions, i.e., waiting for their enabledness, and the actions A_f represent the transformations. Thus, a transition a is split into two components, $a_c \in A_c$ and $a_f \in A_f$. The essential partial order imposes sequencing all the actions in the same process, and pairs of actions that use or set a mutual variable. In the latter case, the enabledness part b_c of the latter transition succeed the transformation part a_f of the earlier transition. However, other transitions can interleave in various ways (e.g., $d_c \prec e_c \prec e_f \prec d_f$). This order relation $\prec$ corresponds to a partial order over Act. The formula is satisfied by all the sequences that satisfy the constraints in $\prec$, i.e., the linearizations (complementation to total orders) over Act. In particular, σ is one (but not necessarily the only) sequence satisfying the constraints in φ.

The partial order can be illustrated as a directed graph, where a node represents an action and an edge represents a $\prec$ relation. For example, we assume that transitions a and b belong to a process and a transition d belongs to another process. A partial order requires $a_c \prec a_f$, $a_f \prec b_c$, $b_c \prec b_f$, $d_c \prec d_f$ and $a_f \prec d_c$. The partial order is shown in Figure 6.

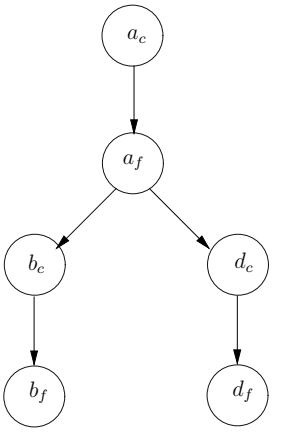

Fig. 6. A partial order

3.2 Generation of an Acyclic ETA from a Partial Order

After we generate the set of the ETAs for the different processes, we label each transition in the ETAs with respect to Act. For example in Figure 3, the edges $s_2 \rightarrow s_5$ and $s_3 \rightarrow s_5$ can be labeled with a_c, the edges $s_3 \rightarrow s_6$ and $s_4 \rightarrow s_6$ can be labeled with b_c. The edge $s_5 \rightarrow s_7$ can be marked by a_f and s_6 to s_8 by b_f.

Let $\mathcal{A}_\prec$ be a finite partial order among occurrences of Act (note that an action from Act can occur multiple times.). We generate an automaton $Lin_\prec$ with edges labeled with actions of Act. The automaton $Lin_\prec$ accepts all the linearizations of $\mathcal{A}_\prec$. Hence, it also necessary accepts the original sequence from which we generated $\mathcal{A}_\prec$.

The algorithm for generating $Lin_{\prec}$ is as follows. The sets of states of $Lin_{\prec}$ are subsets $\mathcal{S} \subseteq St$, the set of occurrences of $\mathcal{A}_{\prec}$, such that for each such subset $\mathcal{S}$, it holds that if $\alpha \prec \beta$ and $\beta \in \mathcal{S}$ then also $\alpha \in \mathcal{S}$. They are the *history closed* subsets of St. A transition is of the form $\mathcal{S} \xrightarrow{\alpha} \mathcal{S} \cup \{\alpha\}$ where α is an occurrence of an action. The empty set is the initial state and the set St is the accepting state. Figure 7 shows the automaton for the partial order in Figure 6.

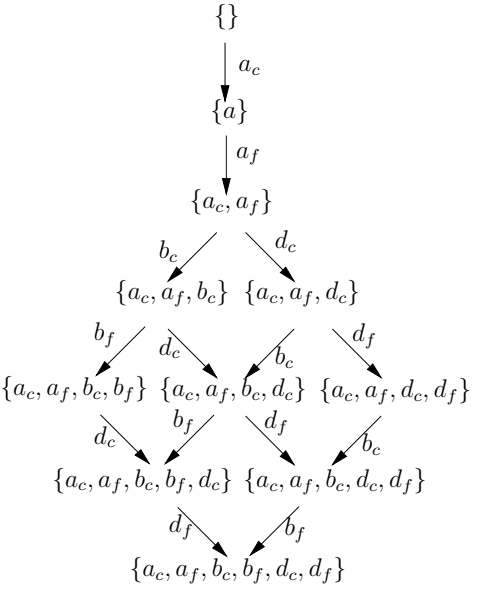

Fig. 7. A partial order automaton

We generate now the acyclic ETA $ETA_{\prec}$, whose executions are linearizations of $Lin_{\prec}$, with a collection of extended timed automata $T_1, \ldots, T_n$. At first, we describe the synchronization of a transition $\bar{\alpha}$ marked with an action $\bar{a} \in Act$ on $Lin_{\prec}$ with the edges in a component T_i:

1. Synchronization of $\bar{\alpha}$ with an edge labeled as $\bar{a}$. This corresponds to the selection or the transformation of a transition in the TTS being taken.
2. Synchronization of $\bar{\alpha}$ with an internal edge τ which references a shared variable v if $\bar{a}$ acquires or releases v. This corresponds to an enabledness condition being changed. If there exists an edge which has the same source state as τ has and is labeled as $\bar{a}$, τ is not allowed to be synchronzed with $\bar{\alpha}$.

Now we generate the initial states of $ETA_{\prec}$. For every participant process T_j, we find the set $\hat{S}_j$ of enabledness states for the first transition occurring in $\mathcal{A}_{\prec}$. At any initial states of $ETA_{\prec}$, T_j stays at one of the states in $\hat{S}_j$. If a process T_k does not have any transitions occurring in $\mathcal{A}_{\prec}$, we asssume it stays at one of its initial states and thus we use the set of initial states as $\hat{S}_k$. An initial

state of $ETA_{\prec}$ is composed of such states that each state belongs to a different set $\hat{S}_j$. Each initial state of $ETA_{\prec}$ is matched to the initial state of $Lin_{\prec}$.

The successive states of $ETA_{\prec}$ is generated in a deductive way from the initial states. For clarity, a state $g = \langle g^1, \ldots, g^n \rangle$ of $ETA_{\prec}$ is denoted by a *global* state and the states $g^1, \ldots, g^n$ composing g are denoted by *component* states. Any global state g has a matched state on $Lin_{\prec}$ as this is guaranted by the deductive generation method. Let g be a global state whose successive global states have not be generated and $\bar{g}$ be the matched state of g on $Lin_{\prec}$. The successive global states of g are generated in the following way:

> We synchronize each transition $\bar{\beta}$ starting at $\bar{g}$ on $Lin_{\prec}$ with ETA edges whose source states are component states of g. For any T_j, let w_j be the set of edges that are synchronized with $\bar{\beta}$ and $|w_j|$ be the number of edges in w_j. A component state g^j in g is the source state of the edges in w_j. A successive state g' with respect to $\bar{\beta}$ is generated by replacing each g^j by the target state of an edge in w_j. If $|w_j| = 0$, T_j does not change its state in g'. If $|w_j| > 1$, the number of successive global states with respect to $\bar{\beta}$ is increased $|w_j|$ times. These successive global states are matched to the target state of $\bar{\beta}$ on $Lin_{\prec}$.

Note that we remove any global states from which there is no path leading to the global states matched to the accepting state on $Lin_{\prec}$. Since there is often a nondeterministic choice for taking such labeled edges, this choice increases the branching degree on top of the branching already allowed by $\mathcal{A}_{\prec}$. The synchronous execution forms a DAG structure, which will be used later calculating the path precondition.

3.3 Data Structure for Backward Reachability Analysis

Time constraints are a set of relations among lock clocks. These contraints can be obtained from reachability analysis of clock zones. Difference-Bound Matrix (DBM) [4] is a data structure for representing clock zones.

A DBM is a $(m+1) \times (m+1)$ matrix where m is the number of local clocks of all processes. Each element $D_{i,j}$ of a DBM D is an upper bound of the difference of two clocks x_i and x_j, i.e., $x_i - x_j \leq D_{i,j}$. We $x_1, \cdots, x_m$ to represent local clocks. The clock x_0 is a special clock whose value is always 0. Therefore, $D_{i,0}$ $(i > 0)$, the upper bound of $x_i - x_0$, is the upper bound of clock x_i; $D_{0,j}$ $(j > 0)$, the lower bound of $x_0 - x_j$, is the negative form of the lower bound of clock x_j. To distinguish non-strict inequality $\leq$ with strict inequality $<$, each element $D_{i,j}$ has the form of (r, F) where $r \in \mathbb{R} \cup \{\infty\}$ and $F \in \{\leq, <\}$ with an exception that F cannot be $\leq$ when r is ∞. Addition $+$ is defined over $F, F' \in \{\leq, <\}$ as follows:

$$F + F' = \begin{cases} F, & \textit{if } F = F' \textit{ and} \\ <, & \textit{if } F \neq F' \end{cases}$$

Now we define addition $+$ and comparison $<$ for two elements (r_1, F_1) and (r_2, F_2).

$$(r_1, F_1) + (r_2, F_2) = (r_1 + r_2, F_1 + F_2).$$

$$(r_1, F_1) < (r_2, F_2) \text{ iff } r_1 < r_2 \text{ or } (r_1 = r_2) \wedge (F_1 =<) \wedge (F_2 =\leq).$$

The minimum of (r_1, F_1) and (r_2, F_2) is defined below:

$$min((r_1, F_1), (r_2, F_2)) = \begin{cases} (r_1, F_1) & \text{if } (r_1, F_1) < (r_2, F_2) \\ (r_2, F_2) & \text{otherwise} \end{cases}$$

A DBM D is *canonical* iff for any $0 \leq i, j, k \leq m$, $D_{i,k} \leq D_{i,j} + D_{j,k}$. A DBM D is *satisfiable* iff there is no such a sequence of indices $0 \leq i_1, \cdots, i_k \leq m$ that $D_{i_1,i_2} + D_{i_2,i_3} + \cdots + D_{i_k,i_1} < (0, \leq)$. An unsatisfiable DBM D represents an empty clock zone.

Calculating time constraints following an edge τ backwards from its target state s to its source state s' has been explained in [11]. Let $I(s')^c$ be the assertion on clocks in state invariant of s', and ψ^c be the assertion on clocks within the edge τ. The DBM D represents the time constraints at s. Assertions $I(s')^c$ and ψ^c are represented by DBMs too. The time constraints D' at s' is defined as follows:

$$D' = ((([\lambda := 0]D) \wedge I(s')^c \wedge \psi^c) \Downarrow) \wedge I(s')^c \tag{1}$$

"$\wedge$" is conjunction of two clock zones. Calculating $D' = D^1 \wedge D^2$ is to set each element $D'_{i,j}$ in D' to be the minimum value of the element $D^1_{i,j}$ in D^1 and the element $D^2_{i,j}$ in D^2, i.e.,

$$D'_{i,j} = min(D^1_{i,j}, D^2_{i,j}).$$

"$\Downarrow$" is time predecessor. Calculating $D' = D \Downarrow$ is to set lower bound of each clock to 0, i.e.,

$$D'_{i,j} = \begin{cases} (0, \leq) & \text{if } i = 0 \\ D_{i,j} & \text{if } i \neq 0 \end{cases}$$

"$[\lambda := 0]D$" is reset predecessor. Calculating $D' = [\lambda := 0]D$ is as follows:

1. Resetting a clock x to 0 can be seen as substituting x by x_0. Let x' be a clock which is not reset. Before resetting, we have constraints $x' - x_0 \leq c_1$ and $x' - x \leq c_2$. After resetting, we obtain constraints $x' - x_0 \leq c_1$ and $x' - x_0 \leq c_2$ by replacing x with x_0. Then these constrains are conjunct into $x' - x_0 \leq min(c_1, c_2)$. Therefore, when we calculate time constraints from after resetting back to before resetting, we substitute $x' - x_0$ by $min(x' - x_0, x' - x)$ and $x_0 - x'$ by $min(x_0 - x', x - x')$. Therefore, for a clock x_i which is not reset, update its upper and lower bounds as follows:
   (a) $D'_{i,0} = min\{D_{i,k} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$.
   (b) $D'_{0,i} = min\{D_{k,i} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$.
2. On the other hand, for a clock x_k which is reset, its value before resetting can be any non-negative real number. Thus its lower bound is 0 and upper bound is ∞, i.e., $D'_{0,k} = (0, \leq)$ and $D'_{k,0} = (\infty, <)$. Furthermore, for any other clock x_j $(j \neq k \wedge j > 0)$, $D'_{k,j} = (\infty, <)$.
3. For a clock x_i which is not reset and a clock x_k which is reset, update $x_i - x_k$ as $D'_{i,k} = D'_{i,0}$. (Note that this step must be done after the upper bound of x_i is updated.)
4. For two clocks x_i and x_j that are not reset, $D'_{i,j} = D_{i,j}$.

Note that D' needs to be changed to canonical form after each operation. This is done using Floyd-Warshall algorithm [5, 10] to find the all-pairs shortest paths.

3.4 Path Condition for a DAG

We can now calculate the condition for that DAG from the leaf states backwards. The condition would use the usual *weakest* precondition for variables, and a similar update for time variables that involve local clocks and time parameters. When a state has several successors, we disjoin the conditions obtained on the different edges.

At first, we give a brief description of updating a condition over variables backwards from a given state to another state over an edge with condition c and transformation of the form $v := expr$, where v is a variable and $expr$ is an expression. Let φ be the condition over variables at the given state. The new condition φ^R is defined as follows:

$$\varphi^R = \varphi[expr/v] \wedge c, \tag{2}$$

where $\varphi[expr/v]$ denotes substituting $expr$ for each free occurrence of v in φ.

The backward calculation of the precondition for a DAG is described as follows:

1. Mark each leaf state as *old* and all other states as *new*. Attach the assertion on variables $\varphi = true$ and the assertion on clocks represented by DBM D_0 to each leaf, noted by $\varphi \wedge D_0$. The DBM D_0 is defined below:

$$\varphi_0 = \begin{pmatrix} (0,\leq) & (0,\leq) & \cdots & (0,\leq) \\ (\infty,<) & (0,\leq) & \cdots & (\infty,<) \\ \vdots & \vdots & \vdots & \vdots \\ (\infty,<) & (\infty,<) & \cdots & (0,\leq) \end{pmatrix} \tag{3}$$

   When we start at a leaf state to calculate time constraints backwards, we do not know the exact value of any local clock when the system enters the leaf state. Thus their values ranges from 0 to ∞. Their exact value scopes can be computed during backward calculation.
2. While there are states marked with *new* do
   (a) Pick up a state z that is marked *new* such that all its successors $Y = \{y_1, \ldots, y_k\}$ are marked *old*.
   (b) Assume each $y_i \in Y$ is attached an assertion over variables and clocks. The assertion has the form of

$$\bigvee_{1 \leq j \leq m_i} (\varphi_{i,j} \wedge \mathcal{D}_{i,j}).$$

   (note that $m_i = 1$ if y_i is a leaf state.) We obtain $\varphi^R_{i,j}$ from $\varphi_{i,j}$ according to the formula (2) and $\mathcal{D}^R_{i,j}$ from $\mathcal{D}_{i,j}$ according to the formula (1).

(c) Attach

$$\bigvee_{\substack{y_i \in Y \\ 1 \leq j \leq m_i}} (\varphi^R_{i,j} \wedge \mathcal{D}^R_{i,j}) \tag{4}$$

to the state z. Mark z as *old*.
Note: when $\varphi^R_{i,j} = false$ or $\mathcal{D}^R_{i,j}$ is not satisfiable, $\varphi^R_{i,j} \wedge \mathcal{D}^R_{i,j}$ is removed from formula (4).

3. When an initial node is reached backwards, the combination of conditions over variables that it represents (refer to Section 2.3 for detail) must be conjuncted with the condition calculated at this state in order to get the initial precondition for this state, because this combination is not processed during the backward calculation. The combinations represented by non-initial nodes are processed through the edges pointing to them. All initial preconditions of initial states are disjuncted together to form the initial precondition of the DAG.

4 Discussion

We described here a method for calculating the path condition for a timed system. The condition is calculated automatically, then simplified using various heuristics. Of course we do not assume that the time constraints are given. The actual time for lower and upper bounds on transitions is given symbolically. Then we can make various assumptions about these values, e.g., the relative magnitude of various time constants. Given that we need to guarantee some particular execution and not the other, we may obtain the time constraints as path conditions, including e.g., some equations, whose solutions provide the appropriate required time constants.

We believe that the constructed theory is helpful in the automatic generation of test cases. The test case construction can also be used to synthesize real time system time. Another way to use this theory is to extend it to encapsulate temporal specification. This allows verifying a unit of code in isolation. Instead of verifying each state in separation, one may verify the code according to the program execution paths. This was done for the untimed case in [6], and we are working on extending this framework for the timed case. Such a verification method allows us to handle infinite state systems (although the problem is inherently undecidable, and hence we are not guaranteed to terminate), and parametric systems e.g., we may verify a procedure with respect to arbitrary allowed input. This is done symbolically, rather than state by state.

References

1. R. Alur, D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 1994, 183–235.
2. N. Budhiraja, K. Marzullo, F. B. Schneider, Derivation of sequential, real-time process-control programs, *Foundations of Real-Time Computing: Formal Specifications and Methods*, 1991, 39-54

3. E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, *Communications of the ACM* 18, 1975, 453–457
4. D. L. Dill, Timing assumptions and verification of finite-state concurrent systems, *Automatic Verification Methods for Finite State Systems*, LNCS 407, 1989, 197–212
5. R. W. Floyd, Algorithm 97: Shortest Path, *Communications of the ACM*, 5(6), 1962, 345
6. E. Gunter, D. Peled, Unit Checking: Symbolic Model Checking for a Unit of Code, *Verification: Theory and Practice 2003*, LNCS 2772, 548–567.
7. T. A. Henzinger, Z. Manna, A. Pnueli, Temporal proof methodologies for timed transition systems, *Information and Computation* 112, 1994, 273–337
8. T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, *Information and Computation* 111, 1994, 193–244
9. D. J. Scholefield, H. S. M. Zedan, Weakest Precondition Semantics for Time and Concurrency, *Information Processing Letters* 43, 1992, 301-308
10. S. Warshall, A theorem on boolean matrices, *Journal of the ACM*, 9(1), 1962, 11–12
11. S.Yovine, Model checking timed automata, *Lectures on Embedded Systems*, LNCS 1494, 1998, 114–152