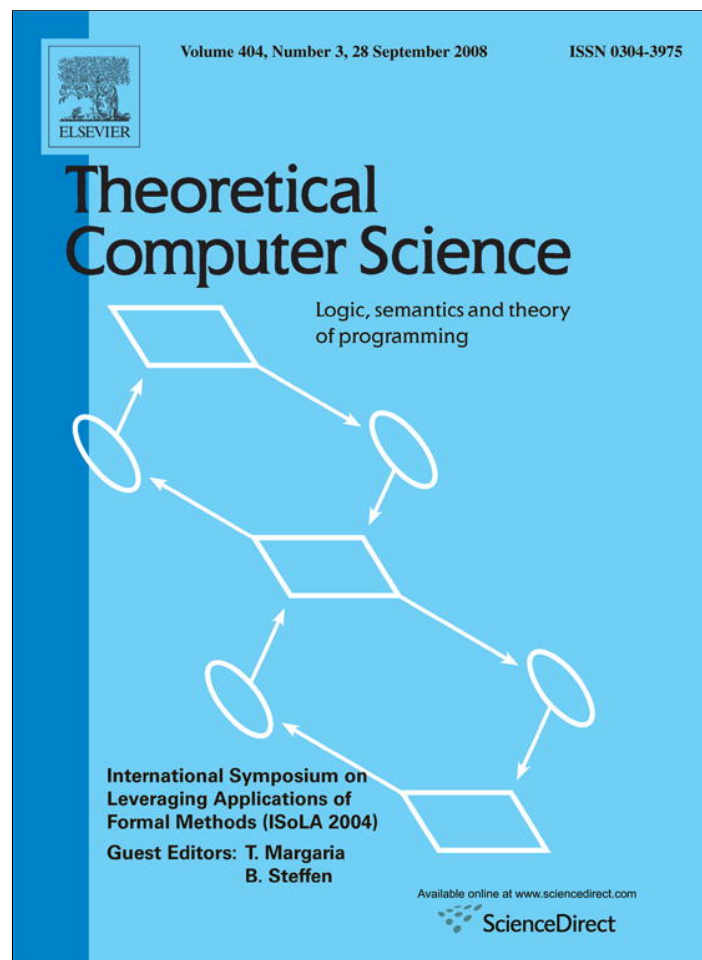


Provided for non-commercial research and education use.  
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

<http://www.elsevier.com/copyright>



Contents lists available at ScienceDirect

## Theoretical Computer Science

journal homepage: [www.elsevier.com/locate/tcs](http://www.elsevier.com/locate/tcs)Automatic generation of path conditions for concurrent timed systems<sup>☆</sup>Saddek Bensalem<sup>a</sup>, Doron Peled<sup>b,\*</sup>, Hongyang Qu<sup>c</sup>, Stavros Tripakis<sup>a</sup><sup>a</sup> Verimag, 2 Avenue de Vignate, 38610 Gieres, France<sup>b</sup> Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel<sup>c</sup> Department of Computing, Imperial College London, London SW7 2RH, UK

## ARTICLE INFO

## Keywords:

Path condition  
 Test case generation  
 Timed transition systems  
 Extended timed automata  
 Partial order  
 Difference-Bound Matrices

## ABSTRACT

This paper presents an automatic method for calculating the path condition for programs with real time constraints. We model concurrent systems using timed transition systems and translate them into extended timed automata. Then an acyclic extended timed automaton is constructed and the path condition is calculated backwards over it. This method can be used for semiautomatic verification of a unit of code in isolation, i.e., without providing the exact values of parameters with which it is called. It can also be used for test case generation for real-time systems. Such a symbolic model checking algorithm was implemented previous in the PET system [E. Gunter, D. Peled, Unit checking: Symbolic model checking for a unit of code, Verification: Theory and Practice 2003, Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday, Lecture Notes in Computer Science, vol. 2772, Springer, 548–567] for untimed systems. 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 timing constraints of alternative choices in the code.

© 2008 Elsevier B.V. All rights reserved.

## 1. Introduction

Software testing often involves the use of informal intuition and reasoning. Although there are several tools and techniques for mechanizing the testing process, many of the tools focus on bookkeeping and administration of test results. Formal methods such as automatic verification (model checking) and deductive verification are more systematic and comprehensive, but they involve some inherent complexity and decidability difficulties. The testing process, on the other hand, benefits from the ability to exploit human experience and intuition in order to accelerate the validation.

Although adding human intuition to the validation process suggests a manual technique, it is possible to employ some mathematical ideas and provide tools to support it. Such tools can help in translating informal ideas and intuition into a formal specification, assist in searching the code, support the process of inspecting it and help analyzing the results. A tester, who is typically also an experienced programmer, may have a vague idea where problems in the code may occur. For example, it may involve executing a particular loop twice, followed by another segment of code. This, along with some data satisfying some particular conditions exchanged with another process, may cause a certain overflow to occur. The following techniques may help the tester to confirm or refute such a suspicion:

<sup>☆</sup> A preliminary version appeared in the proceedings of the Fifth International Conference on Integrated Formal Methods, 29 Nov–2 Dec 2005.

\* Corresponding author.

E-mail address: [doron.peled@gmail.com](mailto:doron.peled@gmail.com) (D. Peled).

- A search of the code, guided by some formal description (e.g., a temporal formula) of the suspicious case. Such a search, based on model checking [5] techniques, should suggest some sequences of instructions that meet the description of program locations mentioned in the suspicious case.
- The generation of a condition for a generated suspicious sequence. Such a condition describes the allowed values for the program variables at the beginning of the sequence. Starting the execution with values satisfying this condition allows one to recreate the execution.

In the current work, we concentrate on the automatic generation of test cases for concurrent real-time systems. In order to test a particular behavior of the system, we generate path conditions for (concurrent real-time) execution paths. Instantiating such path conditions allows us to test the desired path. We do not assume finite state systems. Hence our modeled systems may reference unbounded variables in tests and assignments (when we ignore the particular word length in a given machine). Such a precondition characterizes all the states from which we *can* execute the path. However, there may be other possible executed paths, due to nondeterministic choice, which can be eliminated by adding further synchronization. The path condition calculation can be used in a model checking search, hunting for a path satisfying a given temporal property. This is done for the untimed case in [10] and was implemented in the PET system. It allows us to verify a procedure or collection of procedures in isolation, without providing initial values. Using the weakest precondition calculation, verification is performed symbolically, or “for all parameters at once”. The temporal property is translated into an automaton and contributes to calculation of the path condition (i.e., it is a condition for executing a path while satisfying the temporal property).

For the real-time case, we need to generalize the calculation of a path condition, taking into account only the essential conditions to follow a particular path in the execution. For example, if the path is *abcd*, we may constrain only *a* to precede *b*, for being on the same process, *c* to precede *d*, again, for being on the other process, and *b* to precede *d*, for referring to the same variable. We start with a given path (in the flow chart, or interleaved from different flow charts for concurrent processes) merely from a practical consideration; it is very simple to specify an interleaved execution sequence. However, we look at the essential partial order, which is consistent with 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. Thus, with the above restrictions, *acbd* is equivalent to *abcd* and represents the same (partial order) execution.

For untimed systems, there is no difference between the condition for partial order execution and the condition for executing any of the sequences (linearizations) consistent with it. Because of commutativity between concurrently executed transitions, we obtain the same path condition either way. However, when taking time constraints into account, the actual time and order between occurrences of transitions does affect the path condition (which now includes time information).

After introduction of the untimed path condition in [6], the weakest precondition for a timed system was studied in [4, 12, 16]. The paper [4] extended the guarded-command language in [6] to involve time. But it only investigated sequential programs with time constraints. The paper [16] gave a definition of the weakest precondition for concurrent programs with time constraints, based on discrete time, rather than dense time. The weakest precondition in [12] 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 [11], which assigns a lower and upper time constraints for a transition that includes both enabling transition and a transformation, detailed enough; this is because alternative choices (which were not taken) 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). The framework of the solution is displayed in Fig. 1.

## 2. Modeling concurrent timed systems

As mentioned in the introduction, we describe concurrent real-time systems using timed transition systems (TTSES). We provide semantics for the latter model in terms of extended timed automata (ETAs). 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 ETAs. This section defines the two models and the translation.

### 2.1. Timed transition systems

We consider a *timed transition system* over a finite set of processes  $P_1 \dots P_n$ . Each process consists of a finite number of transitions. Although the processes are not mentioned explicitly in the transitions, each process  $P_i$  has its own location

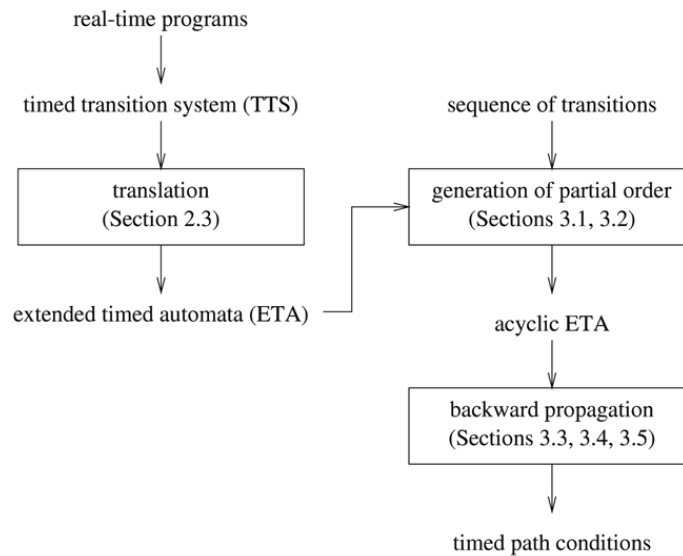


Fig. 1. The framework of the solution.

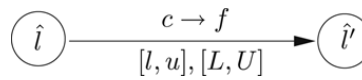


Fig. 2. A transition.

counter  $loc_i$ . The transitions involve checking and updating control variables i.e., location counters, and program variables<sup>1</sup> (over the integers). An enabling condition is an assertion over the variables. 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.

*Writing* (changing the value of) a variable can be done in the transformation part of the transition while *reading* (accessing its value) can be done in either the condition or transformation. We allow shared variables, but make some restrictions on reading and writing by transitions of different processes. In particular, we assume that in our models at most one variable  $v$  can be written by two transitions  $a$  and  $b$  in different processes. Moreover, if both write to  $v$ , then  $v$  is not read in either of their conditions (to achieve this, transitions may need to be broken into several parts).

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. Fig. 2 shows the graphic representation of a transition.

### 2.1.1. Capturing programs as TTS processes

A program is essentially a flow chart. A flow chart can be captured as a TTS process, sometimes in different ways. For instance, an assignment node can be described as a transition with an enabling condition *true* and a transformation that is an assignment (see Fig. 3). A branch node with predicate  $pred$  can be described as shown in Fig. 4. It uses two transitions with a null transformation.  $pred$  and  $\neg pred$  are the enabling conditions of the two transitions respectively, depending on whether the corresponding edge of the diamond is labeled *yes* or *no*. Another way of capturing the branch node is shown in Fig. 5.

<sup>1</sup> We shall use the term “variable” for program variables.

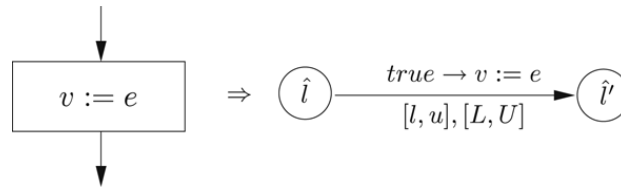


Fig. 3. The description of an assignment node.

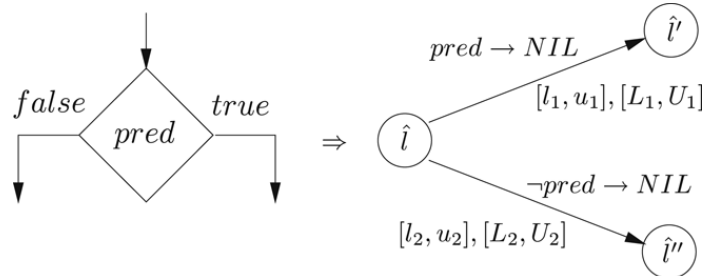


Fig. 4. A possible description of a branch node.

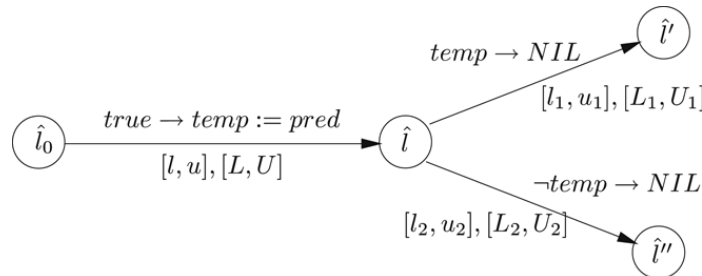


Fig. 5. Another possible description of a branch node.

## 2.2. Extended timed automata

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

- $V$  is a set of variables.
- $X$  is a finite set of assertions over the set of variables  $V$ .
- $Cl$  is a finite set of clocks.
- $B$  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.<sup>2</sup> 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.
- $\Sigma$  is a finite set of labels.
- $E$  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  $e^\Sigma$  from  $\Sigma$  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 clocks 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 edges and states, and transformations on variables on the edges (similar to the difference between finite state machines and extended finite state machines).

<sup>2</sup> We use the term “state” for extended timed automata to distinguish from “location” for timed transition systems.

### 2.2.1. 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  $\delta$ . Then  $V_{i+1} = V_i$  and  $s_{i+1} = s_i$ .

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 the generation of test cases we deal here with finite consecutive segments of executions, which do not have to be initialized.

### 2.2.2. The product of ETA

Let us consider two ETAs,  $ETA_1 = \langle V_1, X_1, Cl_1, B_1, F_1, S_1, S_1^0, \Sigma_1, E_1 \rangle$  and  $ETA_2 = \langle V_2, X_2, Cl_2, B_2, F_2, S_2, S_2^0, \Sigma_2, E_2 \rangle$ . Assume the clock sets  $Cl_1$  and  $Cl_2$  are disjoint.

Then the product, denoted  $ETA_1 \parallel ETA_2$ , is the ETA  $\langle V, X, Cl, B, F, S, S^0, \Sigma, E \rangle$ , where  $V = V_1 \cup V_2$ ,  $X = X_1 \cup X_2$ ,  $Cl = Cl_1 \cup Cl_2$ ,  $B = B_1 \cup B_2$ ,  $F = F_1 \cup F_2$ ,  $S = S_1 \times S_2$ ,  $S^0 = S_1^0 \times S_2^0$ , and  $\Sigma = \Sigma_1 \cup \Sigma_2$ . 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  $\langle (s_1, s_2), e_1^{Cl_1} \cup e_2^{Cl_2}, e_1^{\Sigma_1} \cup e_2^{\Sigma_2}, e_1^{X_1} \wedge e_2^{X_2}, e_1^{B_1} \wedge e_2^{B_2}, e_1^{F_1} \cup e_2^{F_2}, (s'_1, s'_2) \rangle$ .
- 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 timed transition systems into extended timed automata

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 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  $t_1, \dots, t_n$ , let  $c_1, \dots, c_n$  be the enabling conditions of  $n$  transitions respectively. A Boolean combination of these conditions has the form of

$$C_1 \wedge \dots \wedge C_n,$$

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 Boolean combination of enabling condition truth values. For any enabledness states  $s_i$  and  $s_k$ , there is an *internal* edge starting at  $s_i$  and pointing to  $s_k$ . Let  $C_i$  and  $C_k$  be the combinations for  $s_i$  and  $s_k$ , respectively. The edge from  $s_i$  to  $s_k$  is associated with  $C_k$  as the assertion over variables. For any condition  $C_j$  which appears negative ( $\neg c_j$ ) in  $C_i$  and positive ( $c_j$ ) in  $C_k$ , 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. We add a self loop to each state in order to generate the product of automata. The self loop is labeled with the same combination as the state has, but it does not reset any clocks.
2. We construct a *target* state per each transition in the neighborhood to represent its target location.
3. 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  $C$  in which the condition corresponding to the transition  $t_j$  is  $c_j$ , let  $s'_j$  be the intermediate state for  $t_j$  and do the following:
  - 3a. We have the conjunct  $x_j < u_j$  as part of  $s^X$ , the assertion over the clock of  $t_j$ , disallowing  $t_j$  to be enabled in  $s$  more than its upper limit  $u_j$ .

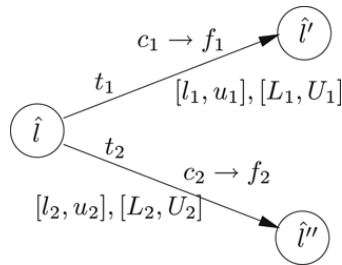


Fig. 6. A neighborhood of two TTS transitions.

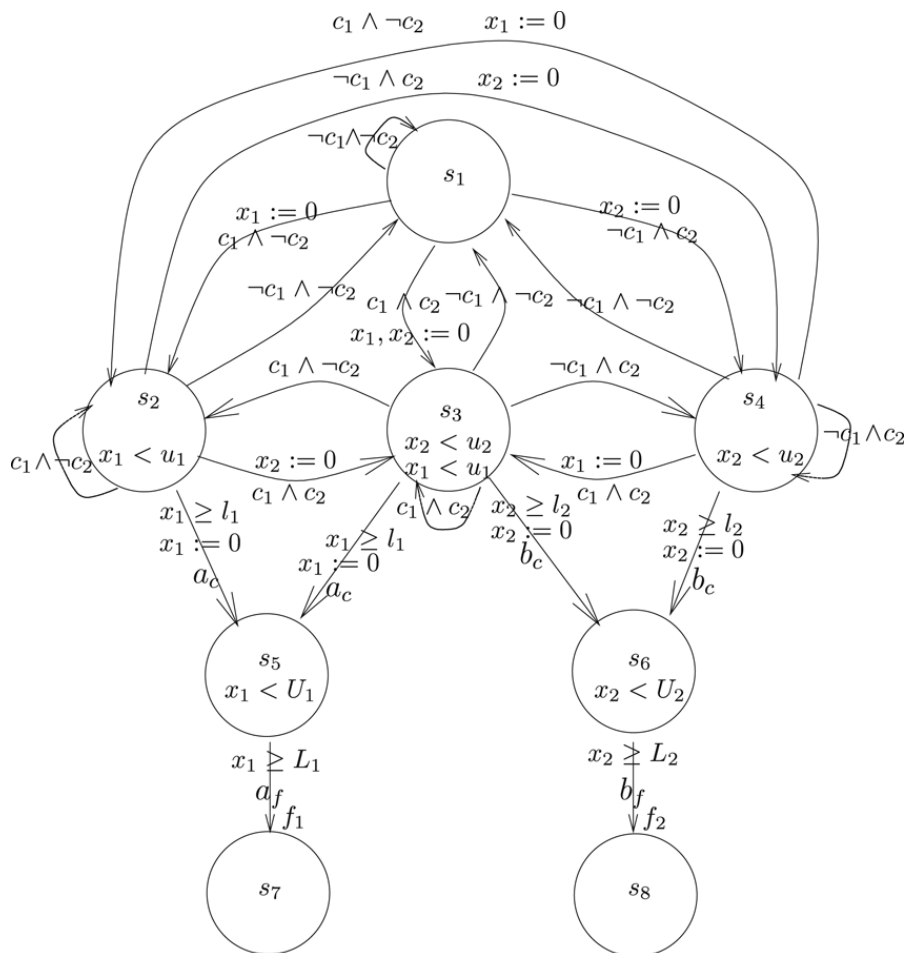


Fig. 7. The ETA for the neighborhood of two TTS transitions.

- 3b. 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.
- 3c. We put the assertion  $x_j < U_j$  into  $s'_j$ , not allowing the transformation to be delayed more than  $U_j$  time.
- 3d. We add a *transformation* edge from  $s'_j$  to the target state of  $t_j$  with the enabling condition  $x_j \geq L_j$  and the transformation of  $t_j$ .

The connection of translations of locations to generate an ETA is done by merging target states with enabledness states that are translated from the same location. Since target states may correspond, by our transformation, to multiple enabledness states of a new location, each edge pointing to the target state is replicated to have a copy pointing to every enabledness state. Moreover, each duplicated edge uses the corresponding combination of conditions of its target state as its enabling condition, and needs to reset clocks that appear in the invariant assertion of its target state.

Fig. 6 illustrates a neighborhood with two transitions and Fig. 7 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 states  $s_5$  and  $s_6$  are intermediate states. 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)

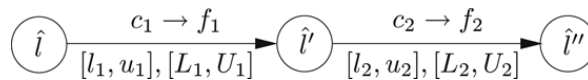


Fig. 8. Two sequential TTS transitions.

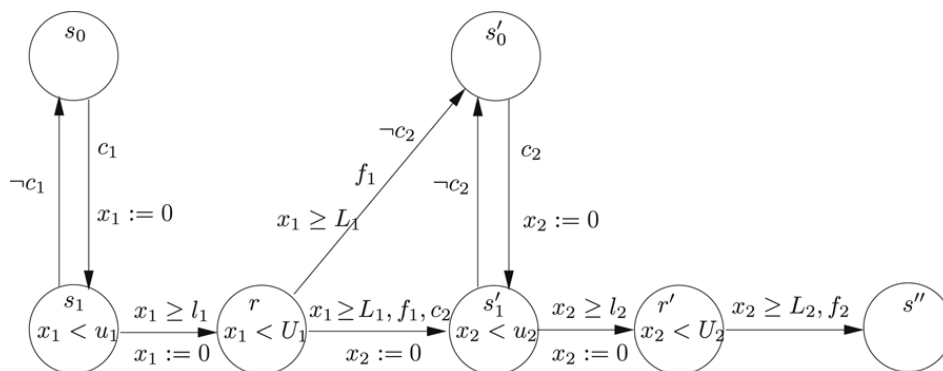


Fig. 9. The ETA for the two sequential TTS transitions.

allows us to wait no longer than  $U_1$  ( $U_2$ , resp.) before we perform  $t_1$  ( $t_2$ ). The states  $s_7$  and  $s_8$  are target states. The edge from  $s_5$  ( $s_6$ ) to  $s_7$  ( $s_8$ ) allows a 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 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.

Fig. 8 shows two consecutive transitions and Fig. 9 provides the ETA construction for these transitions. For simplicity, the self loops are omitted. Location  $\hat{l}$  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  $r$  and  $r'$  are intermediate states.

### 3. Calculating 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 the set of ETAs corresponding to the TTS in question and the TTS path (i.e., program transition sequence) provided by the user.

#### 3.1. The partial order of a TTS path

Given a selected sequence  $\sigma$  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 transformations. Thus, a transition  $a$  is split into two components,  $a_c \in A_c$  and  $a_f \in A_f$ . For two transitions  $a$  with  $a_c$  and  $a_f$  and  $b$  with  $b_c$  and  $b_f$ , and  $a$  occurs earlier in  $\sigma$  than  $b$  does, the ordering relation  $<$  over  $\{a_c, a_f, b_c, b_f\}$  is defined as follows.

1.  $a_c < a_f$  and  $b_c < b_f$ .
2. If  $a$  and  $b$  belong to the same process, then  $a_f < b_c$ .
3. If  $b_f$  writes to  $v$  and  $a_c$  ( $a_f$ , respectively) reads (or writes to)  $v$ , then  $a_c < b_f$  ( $a_f < b_f$ , respectively).
4. If  $a_f$  writes to  $v$  and  $b_c$  ( $b_f$ , respectively) reads  $v$ , then  $a_f < b_c$  ( $a_f < b_f$ , respectively).

By applying the above definition to every pair of transitions in the sequence, we obtain a partial order  $< \subset Act \times Act$ . The transitive relations in the partial order can be removed to form the essential partial order whose transitive closure is the partial order. For example, if actions  $\alpha, \beta, \gamma \in Act$  and  $(\alpha < \beta) \wedge (\beta < \gamma) \wedge (\alpha < \gamma)$ ,  $\alpha < \gamma$  can be removed since it can be deduced from  $(\alpha < \beta) \wedge (\beta < \gamma)$ . This simplification is done by applying the Floyd–Warshall algorithm [8,17]. From here on, we use the term “partial order” for “essential partial order”.

The partial order can be illustrated as a directed graph, where a node represents an action and an edge represents a  $<$  relation. For example, we assume that transitions  $a$  and  $b$  belong to different processes. Let  $a_c$  and  $a_f$  be the enabling condition and the transformation of  $a$  respectively, and  $b_c$  and  $b_f$  the enabling condition and the transformation of  $b$  respectively. Moreover,  $a$  and  $b$  have a shared variable  $v$  that is read by the enabledness condition of transition  $a$  and updated by the transformation of transition  $b$ . The corresponding partial order requires then  $a_c < a_f$ ,  $b_c < b_f$  and  $a_c < b_f$ . The partial order is shown in Fig. 10.

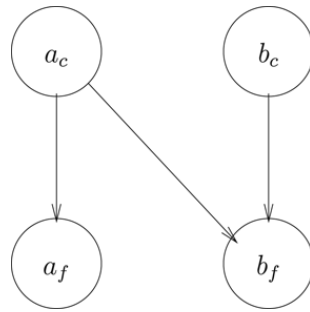


Fig. 10. A partial order.

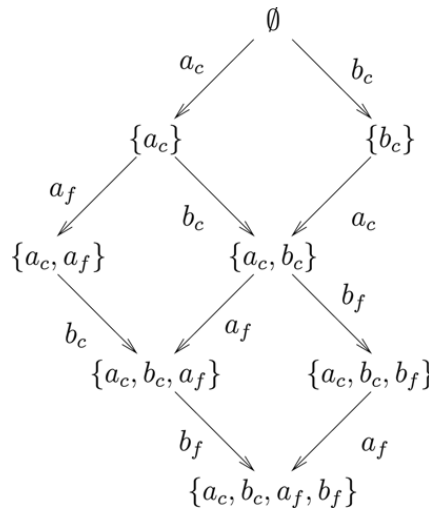


Fig. 11. A partial order automaton.

### 3.2. Generation of an acyclic ETA from a partial order

After we generate the set of the ETAs for different processes, we label each transition in the ETAs with respect to  $Act$ . First, we label the edges from enabledness to intermediate states with corresponding actions from  $A_c$  and the edges from intermediate to target states with actions from  $A_f$ . For example, in Fig. 7, 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  $s_6 \rightarrow s_8$  by  $b_f$ .

Now, the transformation of transitions in one process may change the enabledness of other transitions that read these values in other processes. Thus, we must synchronize such potential change of enabledness with these transformations as follows. Let  $a$  be a transition of some process  $P_i$  that changes some variable  $v$ . Then  $a_f$  appears as a label on all the edges between enabledness states of other processes that have a condition which reads the value of  $v$  (obviously,  $v$  is not a program counter, as  $a$  cannot change the value of program counters other than that of its own process,  $P_i$ ). For the graph in Fig. 7, the edges between nodes  $s_1, s_2, s_3$  and  $s_4$ , including self loops, are labeled with  $d_f$  for each transition  $d$  that is not included in the same process, and has a transformation that can change a variable appearing in the conditions  $c_1$  or  $c_2$ . (These labels do not appear in Fig. 9.)

Recall that edges between enabledness nodes can be labeled by several actions. Formally, that means that such an edge is copied into several copies that are identical except for the label. Moreover, each edge of the ETA generated needs to be labeled by at least some action in order to be executable.

Let  $\prec$  be a finite partial order among occurrences  $St$  of  $Act$ . Note that an action from  $Act$  can occur multiple times. An occurrence is then a pair from  $Act \times \mathcal{N}$ , where  $\mathcal{N}$  are the natural numbers. We generate an automaton  $Lin_{\prec}$  with edges labeled with actions of  $Act$ . The automaton  $Lin_{\prec}$  accepts all linearizations of  $\prec$ . Hence, it also necessarily accepts the original sequence from which we generated  $\prec$ .

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  $\prec$ , 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 of the automaton  $Lin_{\prec}$  is of the form  $\mathcal{S} \xrightarrow{\alpha} \mathcal{S} \cup \{\alpha\}$ , where  $\alpha$  is an occurrence of an action. The empty set is the initial state and the set  $St$  is the accepting state. Fig. 11 shows the automaton for the partial order in Fig. 10.

The product of ETAs for several processes with an automaton  $Lin_{\prec}$  is the standard one. That is, we start with a common initial state of all ETAs, which will be in some enabledness state of their initial label. The automaton  $Lin_{\prec}$  will be in a state corresponding to the empty set of actions taken so far. Then we progress by taking an action  $\mu$  from  $Act$  in *all* automata that is labeled by  $\mu$ .

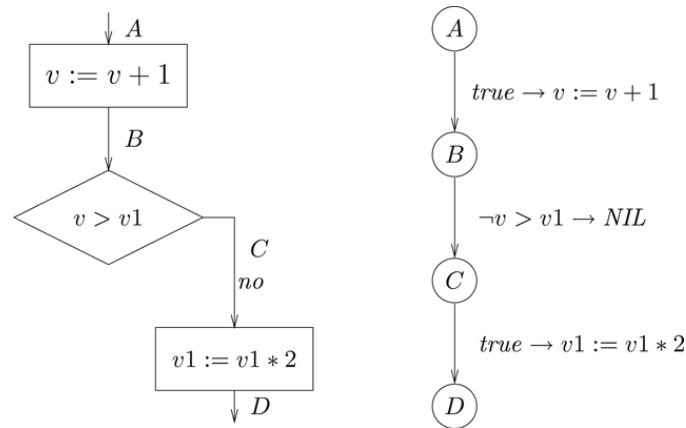


Fig. 12. A flow chart path (left) and its transitions (right).

### 3.3. Calculating untimed path conditions

A *path* of a program is a consecutive sequence of nodes in the flow chart. The projection of an execution sequence on the program counter values is a path through the nodes labeled with these program counter values in the corresponding flow chart. (Not every node has to have an explicit program counter value labeling it, but our implementation automatically provides such a value when translating code to a flow chart.) Thus, in general, a path may correspond to multiple executions. A *path condition* is a first order predicate that expresses the condition to execute the path, starting from a given node. In deterministic code, when we start to execute the code from the first node in the path in a state that satisfies the path condition, we are guaranteed to follow that path. In case of nondeterminism, this is the condition on assignments at the beginning of the path that *can* execute the path (if favorable nondeterministic choices are made). Or stated differently, one can avoid executing the path when starting at its beginning exactly with assignments *not* satisfying the path condition.

We first translate the flow chart nodes into (untimed) transitions. The translation is the same as the way described in Figs. 3 and 4 except that we do not consider time here. We obtain a graph with nodes representing locations and edges representing transitions. When we translate the path at the left of Fig. 12, we obtain the graph on the right of that figure (for simplicity, we do not use the program counter in the translation). To avoid confusion, we use the term “node” for flow chart nodes in this section, while use “point” for nodes in the translated graph.

An *accumulated path condition* represents the condition to move from the current point in the calculation to the *end of the path*. The path condition is the accumulated path condition at the first point of the path. The current point moves at each step in the calculation of the path condition backwards, over one node to the previous point. We start with the condition *true*, at the end of the path (i.e., after the last node). Going backwards from a given point over an edge marked with a transition with condition *c* and transformation *f* and into another point, we perform the following transformations to the accumulated path condition  $\varphi$  to obtain a new path condition  $\varphi^R$ :

- We “relativize” the  $\varphi$  with respect to the assignment representing the transformation; if the assignment is of the form  $v := expr$ , where  $v$  is a variable and  $expr$  is an expression, we substitute  $expr$  for each free occurrence of  $v$  in the path condition. This is denoted by  $\varphi[expr/v]$ , and is generalized to multiple assignment<sup>3</sup> as  $\varphi[expr_1/v_1, \dots, expr_m/v_m]$ .
- Conjoin the condition *c*. We simplify the new accumulated path condition obtained using various first order logic equivalences.

Thus,  $\varphi^R$  is defined as follows:

$$\varphi^R = \varphi[expr_1/v_1, \dots, expr_m/v_m] \wedge c. \tag{1}$$

Calculating the path condition for the example in Fig. 12 backwards, we start at the end of the path, i.e., point **D**, with an accumulated path condition *true*. Moving backwards through the assignment  $v1 := v1 * 2$  to point **C**, we substitute every occurrence of  $v1$  with  $v1 * 2$ . However, there are no such occurrences in *true*, so the accumulated path condition remains *true*. Conjoining *true* with the transition condition *true* maintains *true*. Progressing backwards to point **B**, we now conjoin the accumulated path condition with  $\neg v > v1$ , obtaining (after simplification, which gets rid of the conjunct *true*)  $\neg(v > v1)$ . This is now the condition to execute the path from **B** to **D**. Passing further back to point **A**, we have to relativize the accumulated path condition  $\neg(v > v1)$  with respect to the assignment  $v := v + 1$ , which means replacing the occurrence of  $v$  with  $v + 1$ , obtaining  $\neg(v + 1 > v1)$ . Again, conjoining that with *true* does not change the accumulated path condition. The accumulated path condition at point **A** is the path condition for the path in Fig. 12.

<sup>3</sup> Calculating *first* the expressions  $expr_1$  to  $expr_m$ , then assigning the calculated value to variables  $v_1$  to  $v_m$ , respectively.

### 3.4. Path condition for a DAG

Without timing constraints, the condition to perform at least one path in the DAG can be calculated as follows:

1. Mark all the states of the DAG as *new*.
2. Attach the assertion *true* for each leaf state, which does not have successors, and mark these states as *old*.
3. While there are states marked with *new* do
  - 3a. Pick up a state  $z$  that is marked *new* such that all its successors  $Y = \{y_1, \dots, y_k\}$  are marked *old*.
  - 3b. For each state  $y_i$ , calculate  $\varphi_i^R$  from the assertion  $\varphi_i$  already attached to the state  $y_i \in Y$  and the transition  $c \rightarrow (v_1, \dots, v_m) := (expr_1, \dots, expr_m)$  on the edge between  $z$  and  $y_i$ , according to the formula (1).
  - 3c. Attach  $\varphi_1^R \vee \dots \vee \varphi_k^R$  to state  $z$ . Mark  $z$  as *old*.
4. Suppose an initial state of the DAG is reached during the backward calculation. By the product of automata, such a node can consist of several components from different ETAs. For each such a component that is an enabledness node, there is some combination of conditions associated with it (this condition appears on any incoming edge for such a component; refer to Section 2.3 for details). These combinations, one per each such component node, must be conjuncted with the accumulated precondition. For non initial states this is not needed, as these combinations are processed during the backward calculation.
5. Finally, all initial preconditions of initial states are disjuncted together to form the initial precondition of the DAG.

### 3.5. Adding time constraints

We describe now how to add the time constraints for the DAG conditions. Time constraints are a set of relations among local clocks. We also use a global clock to count the system execution time from its initial state to its last state which, unlike local clocks, is not reset during execution. Time constraint can be obtained from reachability analysis of clock zones. Difference-Bound Matrix (DBM) [7] is a data structure for representing clock zones.

#### 3.5.1. The data structure

A DBM is a  $(m + 2) \times (m + 2)$  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 use  $x_1$  to represent the global clock and  $x_2, \dots, x_{m+1}$  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  $\infty$ . Addition  $+$  is defined over  $F, F' \in \{\leq, <\}$  as follows:

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

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

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

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

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  $0 \leq i, j, k \leq (m + 2)$ ,  $D_{i,k} \leq D_{i,j} + D_{j,k}$ . A DBM  $D$  is *satisfiable* iff there is no such a sequence of indices  $0 \leq i_1, \dots, i_k \leq (m + 2)$  that  $D_{i_1, i_2} + D_{i_2, i_3} + \dots + D_{i_k, i_1} < (0, \leq)$ . An unsatisfiable DBM  $D$  represents an empty clock zone.

Calculating time constraints following an edge  $\tau$  backwards from its target state  $s$  to its source state  $s'$  has been explained in [18]. Let  $I(s')^\tau$  be the assertion on clocks in a state invariant of  $s'$ , and  $\psi^\tau$  be the assertion on clocks on the edge  $\tau$ . The DBM  $D$  represents the time constraints at  $s$ . Assertions  $I(s')^\tau$  and  $\psi^\tau$  are represented by DBMs too. The time constraint  $D'$  at  $s'$  is defined as follows:

$$D' = ((([\lambda := 0]D) \wedge I(s')^\tau \wedge \psi^\tau) \Downarrow) \wedge I(s')^\tau. \quad (2)$$

The operators appearing in formula (2) are calculated as follows:

“ $\wedge$ ” is the conjunction of two clock zones. Let  $D^1$  and  $D^2$  be two DBMs. 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 that 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 constraints 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  $x_i$  that is not reset, update its upper and lower bounds as follows:
  - 1a.  $D'_{i,0} = \min\{D_{i,k} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$ .
  - 1b.  $D'_{0,i} = \min\{D_{k,i} | x_k \in \lambda \cup \{x_0\} \text{ for every } k\}$ .
2. On the other hand, for a clock  $x_k$  that is reset, its value before resetting can be any non-negative real number. Thus its lower bound is 0 and upper bound is  $\infty$ , 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  $x_i$  that is not reset and a clock  $x_k$  that 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 intermediate DBMs in the calculation of formula (2) need to be changed to canonical form after each operation. This is done using the Floyd–Warshall algorithm to find the all-pairs shortest paths.

Reset operation in backward DBM calculation needs special treatment, which is not explained in [18]. Consider the example in Fig. 13. We start the computation at state  $s_3$  with the following DBM  $D$  (for the sake of simplicity, we neglect the global clock here).

$$\begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) \\ (\infty, <) & (0, \leq) & (\infty, <) \\ (\infty, <) & (\infty, <) & (0, \leq) \end{pmatrix}.$$

The clock  $x_1$  is encoded in the second row and  $x_2$  in third. After backward calculation to  $s_2$ , we obtain a new DBM  $D'$ .

$$\begin{pmatrix} (0, \leq) & (-2, <) & (0, \leq) \\ (20, <) & (0, \leq) & (20, <) \\ (8, <) & (-2, <) & (0, \leq) \end{pmatrix}.$$

The DBM calculated backwards at  $s_1$  appears below.

$$\begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) \\ (\infty, <) & (0, \leq) & (\infty, <) \\ (\infty, <) & (\infty, <) & (0, \leq) \end{pmatrix}.$$

That the last DBM is satisfiable means that the path from  $s_1$  to  $s_3$  is possible, while in fact, it is not. This situation implies that backward reset operation loses some useful information which can tell whether the path is possible or not. The element  $D'_{2,1}$  represents  $x_2 - x_1 < -2$ , which means that  $x_1$  goes longer than  $x_2$ . However, the fact that  $x_1$  and  $x_2$  are reset at same time requires that their readings are also the same after being reset. This contradiction reveals that this path is impossible. Therefore, we add an extra operation before the reset operation: If more than one clock is reset at the same time, check whether an upper bound of the differences among them is smaller than 0. If the answer is yes, the DBM cannot be satisfiable.

### 3.5.2. The algorithm

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

1. Mark all the states as *new*.
2. Attach the assertion on variables  $\varphi = \text{true}$  and the assertion on clocks represented by DBM  $\mathcal{D}_0$  to each leaf state, noted by  $\varphi \wedge \mathcal{D}_0$ . The DBM  $\mathcal{D}_0$  is defined below.

$$\mathcal{D}_0 = \begin{pmatrix} (0, \leq) & (-d, \leq) & (0, \leq) & \cdots & (0, \leq) \\ (d, \leq) & (0, \leq) & (d, \leq) & \cdots & (d, \leq) \\ (\infty, <) & (\infty, <) & (0, \leq) & \cdots & (\infty, <) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\infty, <) & (\infty, <) & (\infty, <) & \cdots & (0, \leq) \end{pmatrix} \quad (3)$$

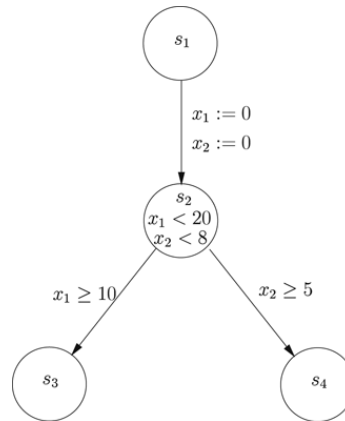


Fig. 13. An example.

<pre> Program 1 begin   wait(v &gt; 0, l, f1);   if (f1 = 0) then     f2 := 0   else     f2 := 1;   end. </pre>	<pre> Program 2 begin   v := 1 end. </pre>
---	--

Fig. 14. An example.

In  $\mathcal{D}_0$ , the upper bound and the lower bound of the global clock are both  $d$ . This is because when we start at a leaf state to calculate time constraints backwards, we do not know the exact value of the global clock when the system enters the leaf state, and therefore assume its value is  $d$ . We need not assume a value for any local clock. Thus their values ranges from 0 to  $\infty$ . Their exact value scopes can be computed during backward calculation. Mark these states as *old*.

3. While there are states marked with *new* do
  - 3a. Pick up a state  $z$  that is marked *new* such that all its successors  $Y = \{y_1, \dots, y_k\}$  are marked *old*.
  - 3b. For each  $y_i \in Y$  there is an assertion over variables  $\varphi_i$  and over clocks  $\mathcal{D}_i$  already attached. We obtain  $\varphi_i^R$  from  $\varphi_i$  according to the formula (1) and  $\mathcal{D}_i^R$  from  $\mathcal{D}_i$  according to the formula (2).
  - 3c. Attach

$$\bigvee_{y_i \in Y} (\varphi_i^R \wedge \mathcal{D}_i^R) \tag{4}$$

to the state  $z$ . Mark  $z$  as *old*.

4. As in the untimed case, as the construction propagates backwards, we need to consider each initial state of the DAG. For each such node, the combination of conditions associated with its components that are enabledness states of ETAs are conjuncted with its accumulated precondition.
5. All initial preconditions of initial states are disjuncted together to form the initial precondition of the DAG.

Note that we must record in each step the status of each shared variable in order not to evaluate a condition which contains a non-accessible shared variable. The precondition calculated by the algorithm is a set of conditions, each of which contains a Boolean expression on variables and a DBM. We allow some time bounds to be parameters. In this case, the Boolean expression may contain a subexpression on the parameters.

#### 4. An example

We give an example to show the whole process of how to obtain a DAG from a timed transition system and a given partial order, and then the precondition of the partial order. A system consists of two concurrent programs. The variable  $v$  is a shared variable and variables  $f1$  and  $f2$  are local variables. The code is given in Fig. 14.

The semantics of `wait` statement is described as follows: It has three parameters. The first one is the condition it waits for to become true. The second is the time limit and the third is a variable. A timer is started when the statement is executed. If the time limit is reached before the condition becomes true, a timeout is triggered and the variable is set to 1. If the condition becomes true before timeout, the variable is set to 0 and the timer is canceled. It is not appropriate to detect whether the `wait` statement timeouts or not by testing the condition because the condition may not be accessed after the `wait` statement. That the time limit is -1 means the process can wait for the condition forever without timeout. In this example,  $l$  is a parameter which is the time limit of a timer.<sup>4</sup> (Note that  $l$  can be substituted to a constant as well.)

<sup>4</sup> Here we use  $l$  to emphasize that the parameter will be translated to time bounds in the TTS.

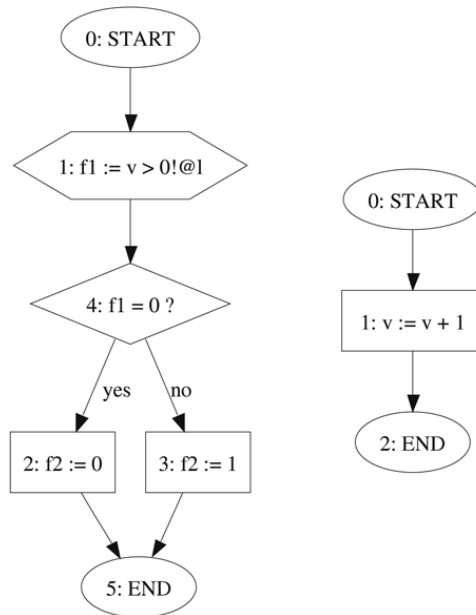


Fig. 15. The flow charts.

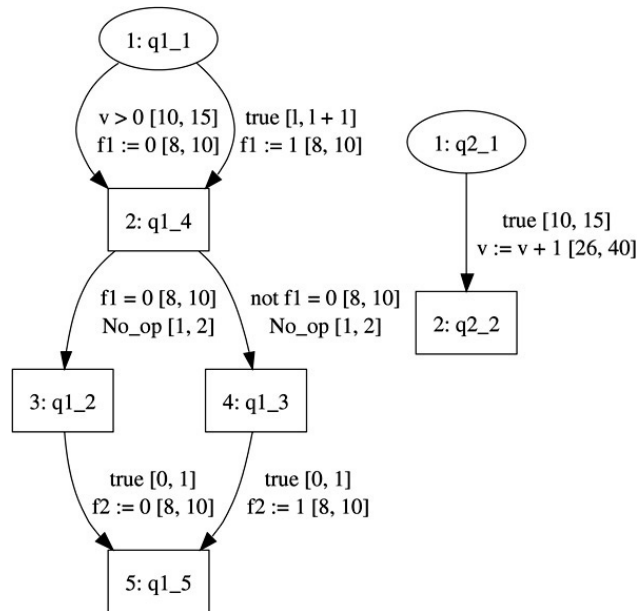


Fig. 16. The transition system.

If the condition  $v > 0$  is not detected before the time limit is reached, a timeout would be triggered. The value range of  $l$  is computed automatically during precondition calculation and given by a predicate in the precondition. The ranges for variables are given in the precondition as well.

Flow charts for the programs are shown in Fig. 15. Program 1 is modeled by the left part of the figure and Program 2 by the right part. There are 4 flow chart nodes in Program 1: the wait statement, the condition of the if statement, the statement when the condition is satisfied and the one when it is not satisfied. Program 2 contains only one flow chart node, which sets  $v$  to 1.

The time transition system is shown in Fig. 16. Program 1 appears on the left and Program 2 appears on the right. Assignments are translated according to Fig. 3 and branch structure according to Fig. 4. The wait statement has two neighbor transitions, one for testing the condition  $v > 0$  and the other for timeout. The node in ellipse shapes are initial locations. The bounds for enabling conditions and transformations are shown beside them respectively.

Time bounds are chosen as follows: the bound for condition true in timeout transition is  $[l, l + 1]$  and the bound for condition true is  $[0, 1]$  for other transitions that do not reference a shared variable or  $[10, 15]$  for those writing a shared variable. The bounds for evaluating the nontautological enabling condition of a transition which does not access any shared variable are  $[8, 10]$ . The bounds for an enabling condition are  $[10, 15]$  if the transition accesses a shared variable. Assigning

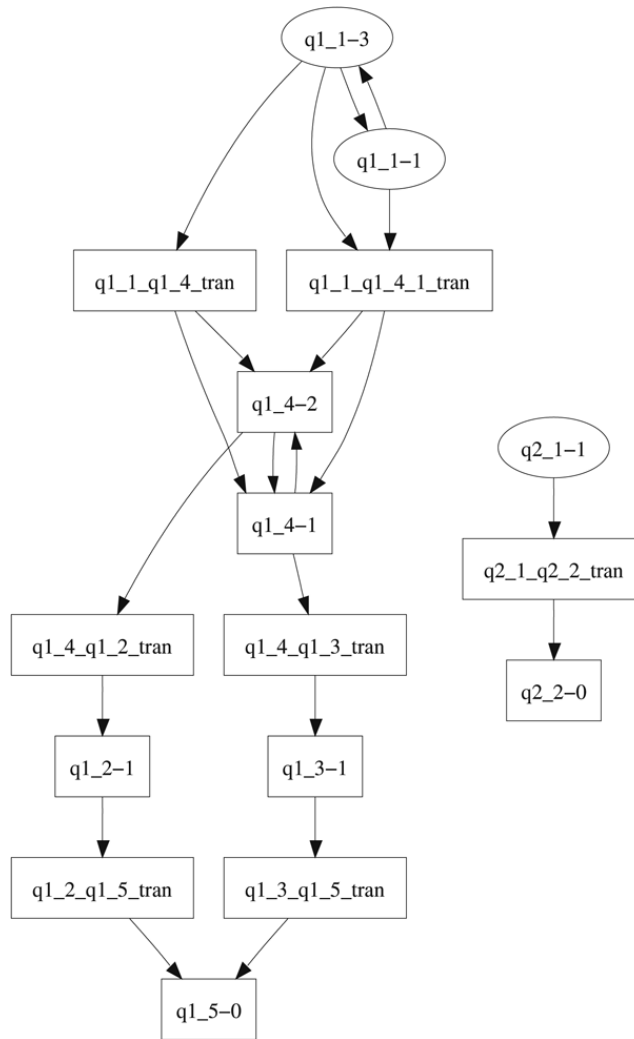


Fig. 17. The extended time automata.

a value to a variable has bounds [8, 10]. The addition operation has bounds [10, 20] and the No\_op has bounds [1, 2]. The transformation  $v := v + 1$  has bounds [26, 40] because it has one addition and accesses the variable twice.

The extended timed automata are shown in Fig. 17. Program 1 is on the left and Program 2 is on the right as well. Initial states are displayed in ellipse. Program 1 has two initial states, one representing the condition  $v > 0$  and the other representing the condition  $\neg(v > 0)$ . Program 2 has one initial condition *true* since it has only one neighbor transition. Each location in the time transition system is translated into a set of states which are named by the location name with different suffix, such “-0” and “-1”. The states whose names are ended in “-tran” are intermediate states. In order to give a succinct demonstration, only the states’ names are shown in the figure. Other information, such as assertions for states and edges, can be deduced from the timed transition system according to translation rules in Section 2.3. For the same reason, self loops are omitted in the figure as well.

The product extended timed automaton is shown in Fig. 18. Only states and edges are displayed. Every state in the product automaton contains one state from Program 1 and one from Program 2. The name of the latter is shown above the name of the former. The product automaton has two initial states because Program 1 has two initial states and Program 2 has one. Note that a product does not contain self-loops.

For a given sequence “ $q1\_1 \rightarrow q1\_4, q1\_4 \rightarrow q1\_2, q1\_2 \rightarrow q1\_5, q2\_1 \rightarrow q2\_2$ ” of transitions in Fig. 16, the calculated partial order is shown in Fig. 19. The left part of the figure is the partial order composed of flow chart nodes in order to give users some intuition about the partial order relation. To generate the DAG, it is decomposed into the one on the right part of the figure, which is composed of ETA edges. The node “ $q1\_1 \Rightarrow q1\_1\_q1\_4\text{-tran}$ ” represents the edge from state “ $q1\_1$ ” to state “ $q1\_1\_q1\_4\text{-tran}$ ” in Program 1. Other nodes have similar meaning. The `wait` statement in Program 1 behaves as no timeout occurs since testing the condition  $f1 = 0$  succeeds (and thus  $f2$  is set to 0). Therefore, the `wait` statement in Program 1 and the assignment in Program 2 compete each other for accessing the shared variable  $v$ . In this partial order, the `wait` statement gains access first and the assignment acquires the shared variable after the `wait` statement releases it. Therefore, there is an edge in the partial order starting at the `wait` statement and pointing to the assignment, representing that the `wait` statement must be executed earlier than the assignment. However, only the enabling condition  $en$  of the transition in the `wait` statement references the shared variable, while the transformation does not. That is, the `wait` statement *reads* the

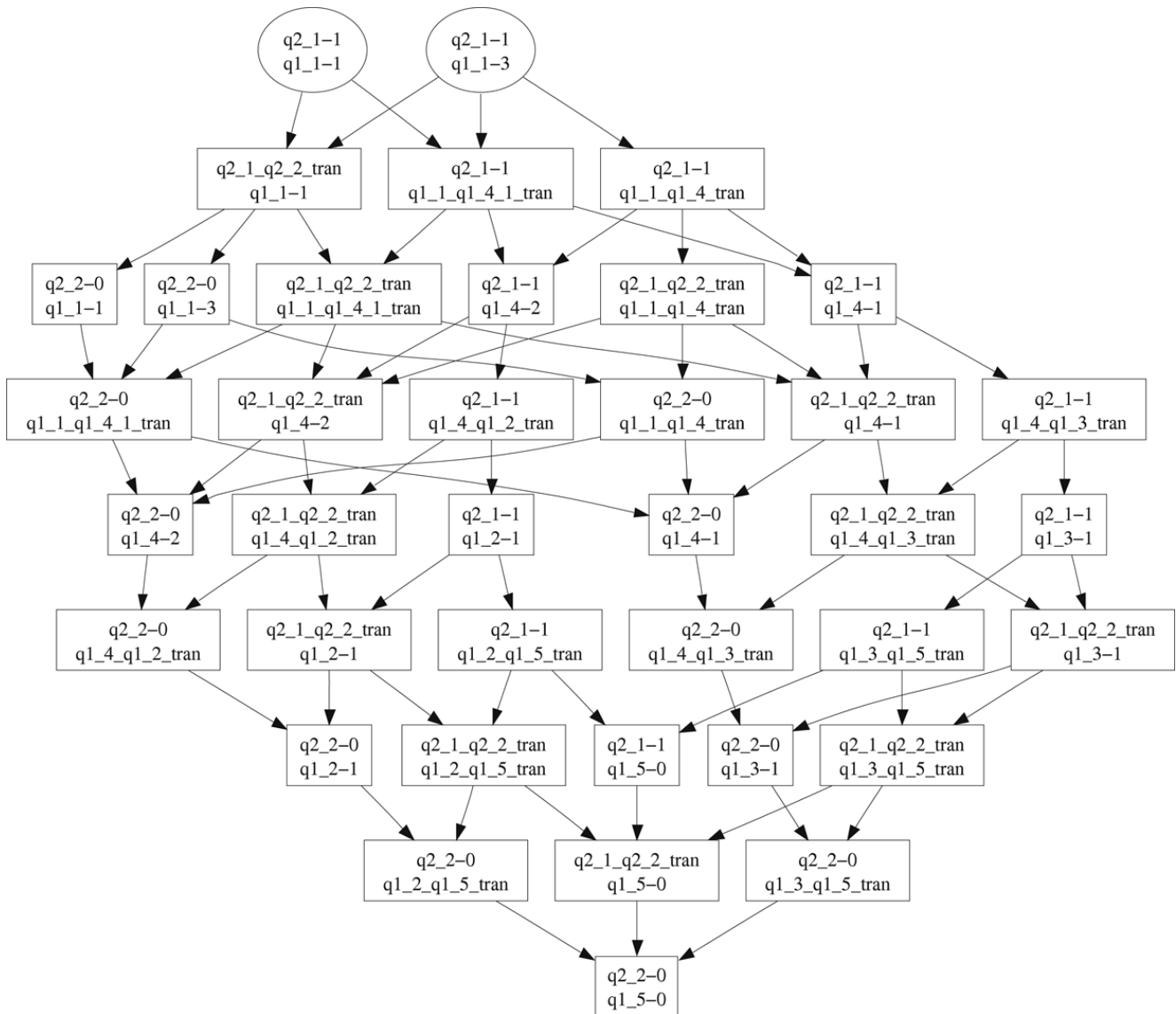


Fig. 18. The product.

shared variable. Thus, the assignment  $v := v + 1$  can be started after the evaluation of  $en$  finishes, which is illustrated as the edge from the node “ $q1_1 \Rightarrow q1_1.q1_4\text{-tran}$ ” to the one “ $q2_1 \Rightarrow q2_1.q2_2\text{-tran}$ ”.

The DAG is shown in Fig. 20. There is only one initial state in this DAG. For other partial orders, there could be multiple initial states. A path from an initial state to a leaf state represents a linearization of the partial order. The precondition of the given partial order is

$$(v \geq 1) \wedge (l \geq 10).$$

The DBMs are omitted from the precondition since they are mainly used for reachability analysis. Not only is there a condition over  $v$  in the precondition, but also a condition over  $l$ . The condition  $l \geq 10$  means that when we use a value to replace  $l$  in Program 1, no linearization of the partial order can be executed if the value is smaller than 10. This example shows that our methodology can be used to calculate the bounds for time parameters, in addition for preconditions over variables. This characteristic is very useful when constructing test cases for real-time systems.

## 5. Implementation

We have implemented the path condition calculation as the real-time extension to the PET system [10], according to the construction described in this paper. The figures in the previous section are based on output generated by PET.

When we translate a timed transition system into extended timed automata, a location can be translated into  $2^n$  states if it has a neighborhood with transitions. In practice, however, the number of states is limited by program structures. For example, the `if` statement has two branches: one satisfies the condition and the other satisfies the negation of the condition. After the branches are translated into transitions, there are less than four combinations of enabling conditions, since the two

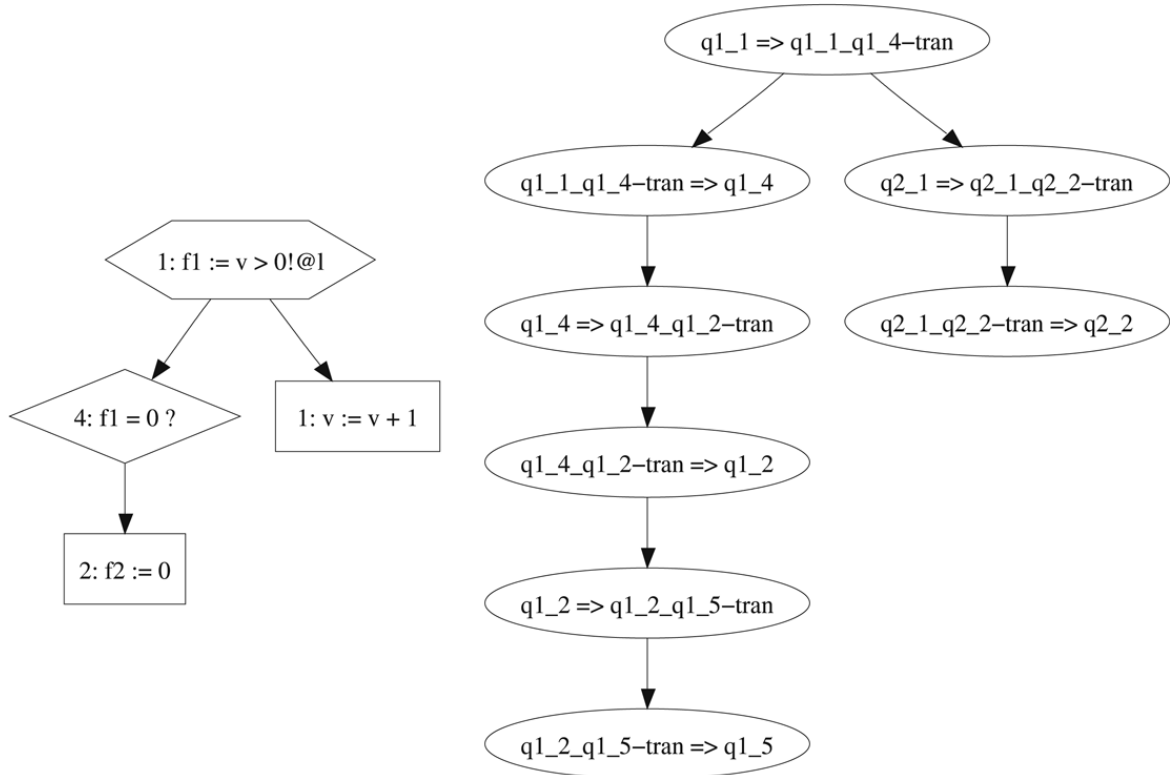


Fig. 19. The partial order.

enabling conditions cannot be both satisfied. Combinations that are equivalent to *false* can be removed, together with the edges attached to them. The `wait` statement in the previous section only generates two combinations because one enabling condition is *true*.

Calculating a precondition for a partial order runs very fast if all time bounds in the system are constants. However, when we calculate time constraints symbolically in order to handle symbolic parameters appearing on time bounds in timed systems (as was shown in the example), the initial constraints on time parameters, such as  $l_1 \leq u_1$  and  $L_1 \leq U_1$  may remain unresolved in their parametrized form in the resulted precondition.

There are two DBM operations that need to be considered carefully during symbolic calculation. One is canonicalization. The other is to check whether a DBM is satisfiable. These two operations have a higher complexity than other operations. Canonicalization can be computed by the Floyd–Warshall algorithm, which has  $O(n^3)$  complexity. Checking satisfiability can be computed by a Bellman–Ford algorithm [2,9,15]. The Bellman–Ford algorithm checks a single source vertex to all other vertices and runs in  $O(nm)$ , which is actually  $O(n^2)$  due to  $m$  equal to  $n - 1$  in DBM. The Bellman–Ford algorithm has to be applied to every source vertex so that checking satisfiability also runs in  $O(n^3)$ . Therefore, both of them generate  $O(n^3)$  symbolic comparisons. Since each comparison may generate three new assumptions, the worst case complexity of computing one DBM is  $O(3^{n^3})$ . Although a worst case never occurs in practice, we still experience a very long execution time. A similar result was obtained on handling parametric DBM in forward reachability analysis in [13].

There are several ways to accelerate the calculation in term of symbolic parameters. An intuitive way is to use a parallel computer to do the calculation. Since the operations on one DBM does not communicate with operations on other DBMs, the current sequential algorithm can be modified to a parallel or distributed algorithm without theoretical difficulty. A second way is to apply partial order reduction to the current algorithm. Partial order reduction on model checking timed automata has been studied in [3,14]. But their results need modification before being applied. Furthermore, both ways can be combined together.

## 6. 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 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 time constraints as path conditions, including e.g., some equations, whose solutions provide the appropriate required time constants.

In addition to the basic TTS model with shared variables and corresponding translation to ETA described in Section 2, the TTS can be extended to handle shared communication. In this case, we need to label different components, in different

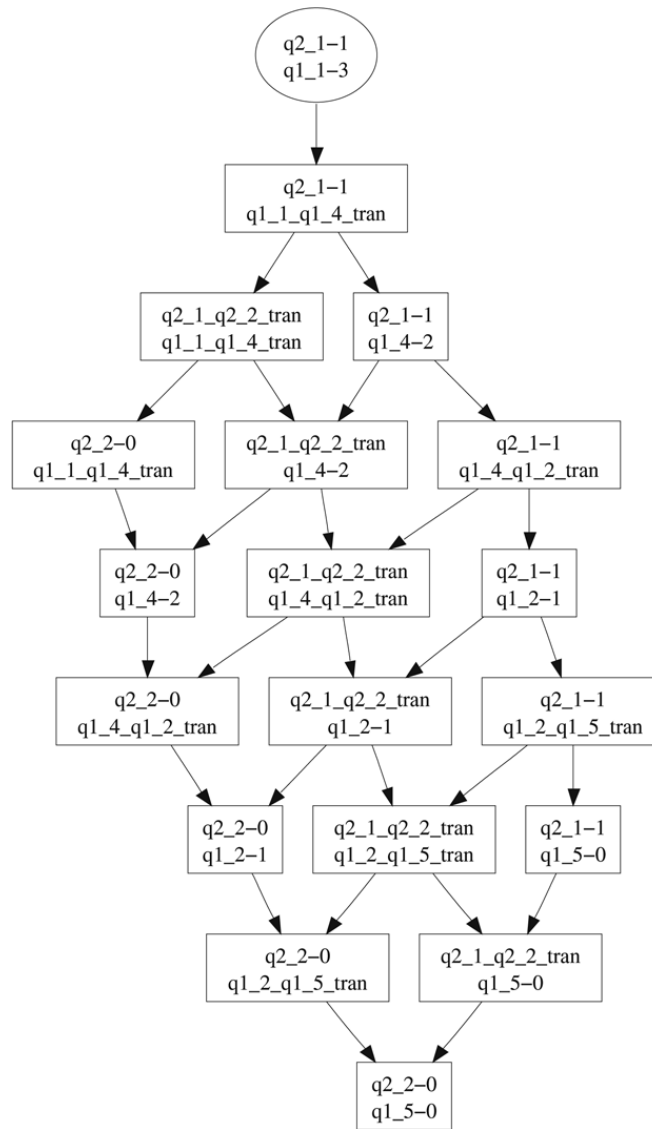


Fig. 20. The DAG.

processes, by the same label. Synchronization is done on both the condition edge (as the edge  $s_2 \rightarrow s_5$  in Fig. 7) and on the transformation edge (as the edge  $s_5 \rightarrow s_7$  in Fig. 7).

We believe that the constructed theory is helpful in automatic generation of test cases. 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 program execution paths. This was done for the untimed case in [10], 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 an arbitrary allowed input. This is done symbolically, rather than state by state.

### Acknowledgement

Second author's 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.

### References

- [1] R. Alur, D.L. Dill, A theory of timed automata, *Theoretical Computer Science* 126 (1994) 183–235.
- [2] R. Bellman, On a routing problem, *Quarterly of Applied Mathematics* 16 (1) (1958) 87–90.
- [3] J. Bengtsson, B. Jonsson, J. Lilius, W. Yi, Partial order reductions for timed systems, in: *The 9th International Conference on Concurrency Theory*, in: *Lecture Notes in Computer Science*, vol. 1466, Springer, 1998, pp. 485–500.
- [4] 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.

- [5] E.M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 2000.
- [6] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, *Communications of the ACM* 18 (1975) 453–457.
- [7] D.L. Dill, Timing assumptions and verification of finite-state concurrent systems, in: *Automatic Verification Methods for Finite State Systems*, in: *Lecture Notes in Computer Science*, vol. 407, Springer, 1989, pp. 197–212.
- [8] R.W. Floyd, Algorithm 97: shortest path, *Communications of the ACM* 5 (6) (1962) 345.
- [9] L.R. Ford, Jr, D.R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.
- [10] E. Gunter, D. Peled, Unit checking: Symbolic model checking for a unit of code, *Verification: Theory and Practice 2003, Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, *Lecture Notes in Computer Science*, vol. 2772, Springer, 548–567.
- [11] T.A. Henzinger, Z. Manna, A. Pnueli, Temporal proof methodologies for timed transition systems, *Information and Computation* 112 (1994) 273–337.
- [12] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, *Information and Computation* 111 (1994) 193–244.
- [13] T.S. Hune, J. Romijn, M. Stoelinga, F.W. Vaandrager, Linear parametric model checking of timed automata, *Journal of Logic and Algebraic Programming* 52–53 (2002) 183–220.
- [14] M. Minea, Partial order reduction for model checking of timed automata, in: *The 10th International Conference on Concurrency Theory*, in: *Lecture Notes in Computer Science*, vol. 1664, Springer, 2002, pp. 431–446.
- [15] E.F. Moore, The shortest path through a maze, in: *The International Symposium on the Theory of Switching*, Harvard University Press, 1959, pp. 285–292.
- [16] D.J. Scholefield, H.S.M. Zedan, Weakest precondition semantics for time and concurrency, *Information Processing Letters* 43 (1992) 301–308.
- [17] S. Warshall, A theorem on boolean matrices, *Journal of the ACM* 9 (1) (1962) 11–12.
- [18] S. Yovine, Model checking timed automata, in: *Lectures on Embedded Systems*, in: *Lecture Notes in Computer Science*, vol. 1494, Springer, 1998, pp. 114–152.