# Extending Promela and Spin for Real Time

Stavros Tripakis*
Costas Courcoubetis*†

ABSTRACT The efficient representation and manipulation of time information is key to any successful implementation of a verification tool. We extend the syntax and semantics of the higher level specification language Promela to include constructs and statements based on the model of timed Büchi automata [2]. We implement these extensions on top of the verification tool Spin.

## 1 Introduction

*Promela* [8] is a language for the specification of *interactive concurrent* systems. Such systems consist of a finite number of separate *components*, which act independently one from another, and interact through the exchange of *messages* over message *channels*. A large part of these systems, including communication protocols, asynchronous circuits, traffic or flight controllers, and real–time operating systems can be characterized as *real–time* systems. This characterization comes from the following two observations :

1. the correct functioning of those systems depends on the timely coordination of their interacting components ; and,

2. information is available about the *time delays* encountered during the operation of system processes.

The first observation is crucial when trying to ensure that the system meets its requirements. The second one can be used to develop a more efficient system : knowing with certainty some facts about the delays in a system can lead to concluding that a number of behaviors are impossible, and therefore, can be ignored during system design.

Traditional formalisms for temporal reasoning deal only with the *qualitative* aspect of time, that is, the *order* of certain system events [1]. However,

*Department of Computer Science, University of Crete, Heraklion, Greece, and Institute of Computer Science, FORTH

†Partially supported by the BRA ESPRIT project REACT.

[1]An example of a qualitative time property is : "the green light is never switched on after the red one and before the orange one"

real–time systems often demand for a *quantitative* aspect of time, that is, taking into consideration the actual distance in time of certain system events [2]. Hence our motivation to extend Promela for real time. We consider time as *dense*, i.e., an unbounded (although finite) number of events can occur between two successive time moments.

An untimed Promela program consists of a collection of components which interact *asynchronously*. Optionally, a special component can be specified, (called the *never–claim*) which interacts with the rest of the system *synchronously*, and models the complement of the desired system behavior. In the absence of the never–claim, wrong behaviors are coded explicitly into the components in terms of non–progress conditions. In either case, the correctness of the system can be reduced to a language–emptyness problem.

Our verification method consists in considering emptyness of *timed Büchi automata* (TBA) [6, 2] which are Büchi automata extended with a finite number of *clocks*. Based on a timed Promela specification, we construct the equivalent (modulo operational semantics) TBA, and then check if the timed language of the latter is empty.

The work described in this document has been, first of all, to extend the syntax and semantics of untimed Promela for clocks and time information. We call this extended language *Real–Time*–Promela (RT–Promela). Next, we have implemented the TBA verification procedure on top of *Spin* [9], obtaining *RT–Spin*, a tool for the verification of RT–Promela programs. Care has been taken, so that the TBA analysis is absolutely compatible with the existing search algorithms used in untimed Spin. Finally, one of our contributions has been the description of a formal semantics of both untimed and RT–Promela, based on untimed and timed transition systems, respectively.

The rest of this document is organized as follows. Section 2 is a short overview of timed languages and automata. In section 3 we review Promela, give its operational semantics in terms of transition systems, and define the verification problem in the untimed case. RT–Promela is presented in section 5 in the same manner : syntactic extensions, semantics in terms of timed transition systems, verification reduced to language emptyness. In the appendix, we also describe *trace* semantics for individual untimed and RT–Promela processes, and show how one can derive the semantics of the complete specification in a compositional way. Experimental results are presented in section 6.

[2]An example of a quantitative time property is : "the orange light will always be switched on at least 5 time units after the red one, followed in at most 0.5 time units by the green one"

## 2 Timed languages and timed Büchi automata

A Büchi automaton (BA) is a nondeterministic finite-state machine $A = (\Sigma, S, Tr, S_0, F)$. $\Sigma$ is the input alphabet, $S$ is the set of states, $S_0$ the set of initial states, and $F$ the set of *accepting states*. $Tr \in S \times \Sigma \times S$ is the transition relation. If $(s, \sigma, s') \in Tr$ then $A$ can move from $s$ to $s'$ upon reading $\sigma$.

A *trace* or input word is an infinite sequence $\sigma = \sigma_1\sigma_2 ..., \sigma_i \in \Sigma$, while a *run* over $\sigma$ is an infinite sequence $s_0 \overset{\sigma_1}{\mapsto} s_1 \overset{\sigma_2}{\mapsto} ..., s_0 \in S_0, (s_i, \sigma_{i+1}, s_{i+1}) \in Tr, i = 0, 1, ....$ A run $r$ is said to be *accepting* iff there exists a state $f \in F$ such that $f$ appears infinitely often in $r$. The *language* $\mathcal{L}(A)$ of $A$ is the set of all traces $\sigma$ such that $A$ has an accepting run over $\sigma$.

A *timed* trace or word is a pair $(\sigma, \tau)$, where $\sigma$ is a trace and $\tau$ is a *time sequence*, i.e., an infinite sequence $\tau_1, \tau_2, ..., \tau_i \in \mathsf{R}^+$. We only consider strictly increasing, *non-zeno* time sequences, i.e., $\tau_i < \tau_{i+1}$ and $\forall t \in \mathsf{R} \exists i,\ \tau_i > t$. This ensures that time *progresses*, that is, does not converge to a bounded value [3]. A *timed language* is a set of timed traces.

A TBA is a tuple $A = (\Sigma, S, Tr, S_0, F, C)$, where $\Sigma, S, S_0$ and $F$ are as in a BA, and $C$ is a finite set of clocks. A transition in $Tr$ has the form $(s, \sigma, s', R, \mu)$, where $R \subseteq C$ are the clocks to be reset to zero, and $\mu$ is a *clock constraint* (or *guard*), that is, a boolean conjunction of atoms of the form $y \leq k$, $k \leq y$, $x - y \leq k$ and $k \leq x - y$ for two clocks $x, y \in C$, and an integer constant $k \in \mathsf{N}$.

Given a timed word $(\sigma, \tau)$, $A$ starts at a state $s_0 \in S_0$ at time 0. All the clocks of $A$ are active, initialized to zero, and increase at the same rate. At time $\tau_1$ the symbol $\sigma_1$ is read and the automaton takes a transition $tr_0 = (s_0, \sigma_1, s_1, R_1, \mu_1)$, only if the values of the clocks satisfy $\mu_1$. The transition is instantaneous, that is, no clocks change, except from the ones belonging in $R_1$ which are reset to zero. At time $\tau_2$ a new input symbol is read, the next transition is chosen, and so on.

More formally, a run $(\overline{s}, \overline{\nu})$ of a TBA over a timed word $(\sigma, \tau)$ is an infinite sequence $(s_0, \nu_0) \overset{\sigma_1, \tau_1}{\longmapsto} (s_1, \nu_1) \overset{\sigma_2, \tau_2}{\longmapsto} ..., \nu_i \in \mathsf{R}^{|C|}$ such that $s_0 \in S_0, \forall x \in C,\ \nu_0(x) = 0$, and $\forall i = 1, 2, ..., (s_{i-1}, \sigma_i, s_i, R_i, \mu_i) \in Tr, (\nu_{i-1} + \tau_i - \tau_{i-1}) \in \mu_i$, and $\nu_i = \nu_{i-1}[R_i := 0]$ [4]. If such a run exists, then $(\sigma, \tau)$ is *timing consistent*. $(\sigma, \tau)$ is accepting iff it is timing consistent and there exists a state $f \in F$ such that $f$ appears infinitely often in $\overline{s}$. The *timed language* $\mathcal{L}(A)$ of $A$ is the set of all timed traces $(\sigma, \tau)$ such that $A$ has an accepting run over $(\sigma, \tau)$. Languages accepted by TBA are called timed *regular*. It is shown that they are closed under union and intersection, but not under complement [1].

[3] An example of a *zeno* time sequence is $0, 1/2, 3/4, 7/8, ....$

[4] The vector $\nu_i$ is called a clock *valuation*. $\nu[R := 0]$ is the valuation $\nu'$ such that $\forall x \in R, \nu'(x) = 0$ and $\forall x \in C \setminus R, \nu'(x) = \nu(x)$. For $t \in \mathsf{R}$, $\nu + t$ is $\nu'$ such that $\forall x \in C, \nu'(x) = \nu(x) + t$. Finally, we write $\nu \in \mu$ if $\nu$ satisfies $\mu$.

The *synchronous product* of two TBA $A_1$ and $A_2$ is a TBA $A = A_1 \otimes_s A_2$ such that $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

## 3 Untimed Promela

### 3.1 Language summary

Promela [5] programs consist of *processes*, message *channels*, and *variables*. A specification in Promela consists in two parts : the *system specification part* (system, for short) and the *property specification part* (the never–claim, which is optional). In the first, a number of process *types* are declared which are then instantiated into *real* processes at run time. A process (usually init) can create other processes (of a certain type) by means of the run statement. Processes execute their statements asynchronously, except in the case of atomic statements, or *rendez–vous* handshakes.

The syntax of the never–claim is just like any other process. However, at most one never claim can be present in the specification. Moreover, it should not *participate* in the execution of the system, but rather *monitor* it. By this we mean that every statement inside a claim is interpreted as a condition, and should not have side effects (i.e., send or receive messages, set global variables, execute run statements etc.). Since the system and the claim operate synchronously, the latter can observe the system's behavior step by step, and catch errors.

### 3.2 The Promela semantics

The operational semantics of an untimed Promela program $\mathcal{P}$ will be specified in terms of a *transition system* (TS), i.e., a (possibly infinite) graph $T = (Q, \rightarrow)$, where $Q$ is the set of nodes, and $\rightarrow \subseteq Q \times Q$ the set of edges. For matters of simplicity, we consider a known number of processes $P_0, P_1, ..., P_m$ which are active right from the start. By convention, $P_0$ will be the never claim, if specified, otherwise, $P_0 \overset{\text{def}}{=} \{$ do:: skip od $\}$.

The state of the system is completely described by the contents of channels and memory (global and local variables), as well as the *control location* of each active process $P_i$. Let $gv$ (resp. $lv_i$) be the vector representing the current values of global variables (resp. local variables of $P_i$), and $l_i$ be the location of $P_i$. The location just after the opening bracket { (resp. just before the closing bracket }) is $start_i$ (resp. $end_i$). We write $l_i \xrightarrow{st} l'_i$ iff there is a statement $st$ from $l_i$ to $l'_i$. The variable vectors after executing $st$ are $st[gv]$ and $st[lv_i]$.

[5]The reader can refer to [8, 9, 10] for a complete presentation of untimed Promela.

So, states of $Q$ are of the form $q = (l_0, l_1, ..., l_m, lv_0, lv_1, ..., lv_m, gv)$. A statement $l'_i \xrightarrow{st_i} l''_i$ of $P_i$ is *enabled* at $q$ iff $l'_i = l_i$ and :

1. either $st_i$ is an assignment, skip, or conditional statement satisfied at $q$ ;

2. or $st_i$ is an asynchronous send (resp. receive) to a non-full (resp. from a non-empty) channel ;

3. or $st_i$ is a rendez-vous send and there exists a rendez-vous receive $l'_j \xrightarrow{st_j} l''_j$ such that $l'_j = l_j$.

Then, $tr = (q, q') \in \rightarrow$ iff there exists a statement $l_0 \xrightarrow{st_0} l'_0$ of $P_0$ enabled at $q$, and :

1. either there exists a statement $l_i \xrightarrow{st_i} l'_i$ of $P_i$ enabled at $q$ such that $q' = (l'_0, ..., l'_i, ..., l_m, st_0[lv_0], ..., st_i[lv_i], ..., lv_m, st_i[st_0[gv]])$ ;

2. or there exists a rendez-vous pair $l_i \xrightarrow{st_i} l'_i$, $l_j \xrightarrow{st_j} l'_j$ enabled at $q$ such that $q' = (l'_0, ..., l'_i, ..., l'_j, ..., l_m, st_0[lv_0], ..., st_i[lv_i], ..., st_j[lv_j], ..., lv_m, gv')$, where $gv' = st_i[st_j[st_0[gv]]]$) ;

3. or no statement of any process $P_i, i > 0$ is enabled at $q$ (such a state is called a *deadlock* one) and $q' = (l'_0, ...l_m, st_0[lv_0], ...lv_m, st_0[gv])$ [6].

The initial state of $T$ is $q_0 = (start_0, ..., start_m, lv_0^{init}, ..., lv_m^{init}, gv^{init})$. A *path* of $T$ is a sequence $q_0, q_1, ...$ such that $(q_i, q_{i+1}) \in \rightarrow$.

## 3.3 Verification in untimed Promela

The *correctness criteria* of $\mathcal{P}$ are implied by the various types of analysis performed using the tool Spin. Locations can be optionally labeled as end, accept, or progress. For a state $q = (l_0, l_1, ..., l_m, ...)$, we define $end(q) = \{l_i \mid l_i = end_i$ or $l_i$ is labeled as end $\}$, and $accept(q) = \{l_i \mid l_i$ is labeled as accept $\}$. A deadlock state $q$ is called *valid* iff $\forall i = 1, .., m,\ l_i \in end(q)$. We say that $\mathcal{P}$ is :

1. *deadlock free* iff all deadlock states are valid ;

2. $\omega$*-correct* iff for each infinite path $p$, if $q_1, ...q_k$ is the set of states appearing infinitely many times in $p$, then $\forall i = 1, ..., k,\ accept(q_i) = \emptyset$ (i.e., there is no cycle passing by a location labeled as accept).

[6]This case corresponds to Promela's *claim-stuttering* semantics.

# 4 Time extensions

## 4.1 Syntax

First of all, we add the type clock to the declarations of Promela variables. Clock variables can be scalar or arrays, and are declared globally [7]. Here is an example of the declaration of clocks :

```
clock x, y, z[5];
```

Next, each statement is expanded with an optional time part, according to the following grammar rules :

```
     stmnt ::=  untimed_stmnt   |   timed_stmnt
timed_stmnt ::=  'when' '{' μ '}' untimed_stmnt
              |  'reset' '{' R '}' untimed_stmnt
              |  'when' '{' μ '}' 'reset' '{' R '}' untimed_stmnt
         R ::=  clock ',' R
         μ ::=  ineq ',' μ
      ineq ::=  clock op int    |   clock op clock '+' int
     clock ::=  x, y, z ∈ C    |   x '[' expr ']'
        op ::=  '<'  |  '>'  |  '<='  |  '>='  |  '=='
```

Here are some examples of timed statements :

| | |
|---|---|
| when$\{x < 4, x \geq 2\}$ reset$\{x\}$ | B!mymesg ; |
| when$\{z < 1, y \geq 1\}$ reset$\{x, z\}$ | a = a*b ; |
| when$\{x[i] == 1\}$ | goto error ; |

The guard $\mu$ is interpreted as the *conjunction* of the inequalities it consists of, e.g., "when $\{x < 4, x \geq 2\}$" stands for "when $\{x < 4 \wedge x \geq 2\}$". There is no way to express *disjunctions* using a single statement. Instead, one should use a branching nondeterministic statement, like :

```
if
:: when{x < 4} reset{x} stmnt_part
:: when{x ≥ 2} reset{x} stmnt_part
fi
```

The reason for the above restriction will be clear in section 5, where we discuss our verification methodology.

## 4.2 The RT–Promela Semantics

The semantics of a RT–Promela program $\mathcal{P}$ is a *timed* TS (TTS) $(Q^{\tau}, \rightarrow^{\tau})$. States of $Q^{\tau}$ are of the form $(q, \nu)$ where $q$ is as in section 3.2 and $\nu$ is a clock valuation. A timed statement $(st, R, \mu)$ is enabled at $(q, \nu)$ if $st$ is

[7]The reason for this is that the clock–space dimension cannot change at run time.

enabled at $q$ and $\nu \in \mu$. The transition relation $\rightarrow^{\tau}$ contains two types of transitions :

1. Action transitions, $((q,\nu),(q',\nu'))$ (defined as in section 3.2). Each such transition is associated with a pair (resp. triple) of timed statements $(st_0, R_0, \mu_0)$, $(st_i, R_i, \mu_i)$ (resp. and $(st_j, R_j, \mu_j)$) which are enabled at $(q,\nu)$ (this implies $\nu \in \mu_0 \wedge \mu_i \wedge \mu_j$). Let $R$ be $R_0 \cup R_i$ (resp. $R_0 \cup R_i \cup R_j$). Then, $\nu' = \nu[R := 0]$.

2. Time transitions, $((q,\nu),(q,\nu+\delta))$, for $\delta \in \mathbb{R}_+$.

The initial state of $T$ is $(q_0, \mathbf{0})$, $\mathbf{0} = (0,...,0) \in \mathbb{R}_+^{|C|}$. The correctness criteria of a RT–Promela program are identical to those defined in section 3.2 except that instead of $T$ we consider $(Q^{\tau}, \rightarrow^{\tau})$.

## 5 Verification using RT–Promela

Our aim is to reduce the verification of the correctness criteria of RT–Promela programs to verification of TBA emptyness, following the approach of [5]. For a RT–Promela program $\mathcal{P}$, we define two TBA $A^{\omega}_{\mathcal{P}}$ and $A^{dlock}_{\mathcal{P}}$, one for each correctness criterion.

### *5.1 The TBA defined from a RT–Promela program*

Then, $A^{\omega}_{\mathcal{P}} \overset{\text{def}}{=} (Q, Q, Tr, \{q_0\}, F, C)$, where $Q, q_0$ are as in section 3.2, $C$ is the set of declared clocks, $F = \{q \mid accept(q) \neq \emptyset\}$, and $(q, \sigma, q', R, \mu) \in Tr$ iff $\sigma = s$ and $\mu = \mu_0 \wedge \mu_i$, $R = R_0 \cup R_i$ (or $\mu_0 \wedge \mu_i \wedge \mu_j$, $R_0 \cup R_i \cup R_j$, respectively, in the case of a rendez-vous handshake).

Similarly, $A^{dlock}_{\mathcal{P}} \overset{\text{def}}{=} (\Sigma', Q \cup \{end\}, Tr', \{q_0\}, F', C)$, where $\Sigma' = Q \cup \{\sigma_{end}\}$, $F' = \{end\}$, and $end \notin Q$. $Tr'$ is obtained by adding to $Tr$ a a transition $(q, \sigma_{end}, end, \emptyset, true)$ for each invalid deadlock state $q$, plus the loop $(end, \sigma_{end}, end, \emptyset, true)$.

The following follows directly from the above definitions.

**Theorem 5.1** • $\mathcal{P}$ is deadlock free iff $\mathcal{L}(A^{dlock}_{\mathcal{P}}) = \emptyset$.

• $\mathcal{P}$ is $\omega$–correct iff $\mathcal{L}(A^{\omega}_{\mathcal{P}}) = \emptyset$.

Let $\mathcal{L}$ be a timed language $\mathcal{L}$. Its *untimed projection* is defined as $unt(\mathcal{L}) = \{\sigma \mid \exists \tau \; s.t. \; (\sigma,\tau) \in \mathcal{L}\}$. Then, $unt(\mathcal{L}) = \emptyset$ iff $\mathcal{L} = \emptyset$ [1]. Thus, it suffices to check the emptyness of $unt(\mathcal{L}(A^{dlock}_{\mathcal{P}}))$ and $unt(\mathcal{L}(A^{\omega}_{\mathcal{P}}))$.

**Theorem 5.2** [[1]] For each TBA $A$ there exists a BA $U$ accepting $unt(\mathcal{L}(A))$.

Intuitively, $U$ will also have an extended state space, each state $(q, \alpha)$ containing, apart from the state $q$ of $A$, the set $\alpha$ of all possible clock valuations.

The latter is generally infinite, due to dense time. To represent such a set, the valuation space $\mathsf{R}^{|C|}$ is partitioned into a finite number of *equivalence classes*. Two members $\nu$ and $\nu'$ of a class $\alpha$ are equivalent in the sense that, if $\nu$ belongs to an accepting run $(s_0, \nu_0) \overset{\sigma_1,\tau_1}{\longmapsto} (s_1, \nu_1) \overset{\sigma_2,\tau_2}{\longmapsto} \ldots \overset{\sigma_i,\tau_i}{\longmapsto}$ $(s_i, \nu) \overset{\sigma_{i+1},\tau_{i+1}}{\longmapsto} (s_{i+1}, \nu_{i+1})\ldots$, then it can be substituted by $\nu'$, which gives another accepting run $(s_0, \nu_0) \overset{\sigma_1,\tau_1}{\longmapsto} (s_1, \nu_1) \overset{\sigma_2,\tau_2}{\longmapsto} \ldots \overset{\sigma_i,\tau_i}{\longmapsto} (s_i, \nu') \overset{\sigma_{i+1},\tau_{i+1}}{\longmapsto}$ $(s_{i+1}, \nu'_{i+1})\ldots$, so that the untimed projections of the two runs are the same.

## 5.2 *Checking emptyness efficiently*

Checking whether $\mathcal{L}(U) = \emptyset$ is reduced to a reachability analysis (depth-first search) seeking loops which pass by accepting states. The state space of $U$ is constructed *on-the-fly*, i.e., during the dfs. Notice that if $Q$ is finite, then the state space of $U$ is also finite, since the set of equivalence classes $C_\alpha$ is finite. Therefore, termination is ensured.

Each control state $q$ can be visited up to $|C_\alpha|$ times. This number depends on the number of clocks and the constraints, and can be quite large. Thus, we would like to be able to do the analysis in terms of *unions* of equivalence classes. Such a union is called a *clock region*, noted $CR$. Our goal is to find an efficient way to represent clock regions.

A very popular representation uses *difference bounds matrices* (DBMs) [6]. DBMs are inexpensive as far as storage is concerned. Moreover, they are simple and require low–cost operations. Briefly, a DBM is a square matrix which describes a very simple system of linear inequalities, of the same form as time constraints, that is, $x$ op $k$, or $x$ op $y + k$, where op $\in \{<, >, \leq, \geq, =\}$, and $k$ is a positive integer constant. Assuming the dimension of a matrix $D$ to be $n \times n$, the set of vectors $\nu \in \mathsf{R}^n$ which satisfy the corresponding inequalities will be denoted $\nu(D)$. This set is convex. Then, the idea is to represent a clock region $CR$ by a DBM $D$, so that $\nu(D) = CR$. At each step during the reachability analysis, a new DBM is computed by transforming the old one. For this, we use a small number of low–cost operations described briefly below. The reader can refer to appendix 1.3 for the precise definitions.

The *intersection* of two DBMs corresponds to the intersection of their regions. The *time–elapse* transformation yields a new DBM which contains all time–successor valuations of the old one, by letting an arbitrary amount of time (possibly zero) elapse. The *clocks–reset* transformation yields a new DBM where some clocks are reset to zero.

In general, more than one DBMs can be used to represent the same set of clock valuations. This is due to the fact that the bounds found in certain inequalities are not "strict" enough. Nevertheless, it is possible to obtain the *canonical* form of a DBM, which is its unique, "minimal" representative. Let $\mathrm{cf}(D)$ denote the canonical form of a DBM $D$, and $D_1, D_2$ be two different matrices. Then : $\nu(D_1) = \nu(D_2) \Leftrightarrow \mathrm{cf}(D_1) = \mathrm{cf}(D_2)$.

The use of canonical form reduces the test for equality of two matrices to a test for the equality of their canonical forms. This is in turn reduced, at the implementation level, to a test for pointer equality, since all DBMs are usually stored in a hashing table. The rest of the DBM operations are also simplified by the use of canonical forms.

During the series of transformations, it is possible that the resulting DBM does not "cover" exactly a clock region. Indeed, a clock region is a union of equivalence classes, which is not always convex, while the region represented by a matrix always is. In that case the matrix can be enlarged to include as many points of the region as possible, resulting in a canonical representation [8]. This process is called *maximization*. For a DBM $D$, and the maximized one, $max(D)$ the following property holds for all other DBM $D'$ :

$$(\forall \alpha \ (\alpha \cap \nu(D) \neq \emptyset \Leftrightarrow \alpha \cap \nu(D') \neq \emptyset)) \Rightarrow \nu(D') \subseteq \nu(max(D)),$$

where $\alpha$, as usual, denotes an equivalence class.

To prove the correctness of our approach, let us define another automaton, called the *DBM automaton*, denoted $A_{DBM}$. This plays the same role as $U$, following exactly the same runs as $A$ does, and keeping track of the possible clock positions at each step.

For a TBA $A = (\Sigma, Q, Tr, Q_0, F, C)$, we define $A_{DBM} \overset{\text{def}}{=} (\Sigma, Q', Tr', Q'_0)$ such that :

- the states of $A_{DBM}$ are of the form $(q, D)$, where $q \in Q$ and $D$ is a DBM ;
- the initial states of $A_{DBM}$ are of the form $(q_0, D_0)$, where $q_0 \in Q_0$ and $D_0$ is the DBM such that $\nu(D_0) = \{\mathbf{0}\}$ ;
- $A_{DBM}$ has a transition $((q, D), \sigma, (q', D'))$ iff $(q, \sigma, q', R, \mu) \in Tr$, and $D'$ is obtained by $D$ by the following sequence of DBM transformations :

$$D \xmapsto{\delta} D^{\delta} \xmapsto{\mu} D^{\mu} \xmapsto{\text{cf}} D^{cf} \xmapsto{[R:=0]} D^0 \xmapsto{max} D^{max} = D'.$$

In the above sequence, $\xmapsto{\delta}$ represents the time–elapse transformation, that is, $\forall \nu \in \nu(D), \delta \geq 0, \ \nu + \delta \in \nu(D^{\delta})$. $\xmapsto{\mu}$ represents the intersection with the constraint $\mu$, that is, $\forall \nu \in \nu(D^{\mu}), \ \nu$ satisfies $\mu$. $\xmapsto{[R:=0]}$ represents the clock resets, that is, $\forall \nu \in \nu(D^0), x \in R, \ \nu(x) = 0$. Finally, $\xmapsto{max}$ represents the maximization process. Intuitively, the whole series of transformations corresponds to the fact that, being in a state, the system lets the time pass first (this can be zero time) and then executes a statement instantaneously,

[8]It is not wrong to add these extra points, since each one of them is equivalent with at least one point in the matrix, thus satisfies exactly the same properties regarding the evolution of the system in time.

moving to another state. In order for the transition to be taken, the time constraints must be satisfied. At the same moment, a number (possibly zero) of clocks are reset to zero.

Not all paths which are discovered during the reachability analysis are valid. Indeed, the presence of time gives meaning only to those infinite executions for which time progresses without bound (recall non–zeno timed traces, defined in section 2). A run of $A_{DBM}$, $r = (s_0, D_0) \xmapsto{\sigma_1} (s_1, D_1) \xmapsto{\sigma_2} \ldots$ over a trace $q$, is ***progressive*** iff for each clock $x \in C$ :

1. there are infinitely many $i$'s such that $D_i$ satisfies $(x = 0) \vee (x > c_x)$, where $c_x$ is the maximum constant that appears in an inequality of the form $x$ op $c_x$ in the specification ;

2. there are infinitely many $j$'s such that $D_j$ satisfies $x > 0$.

## 6 Examples

We have implemented the method described above on top of the tool Spin, developed for the validation of concurrent systems [8], by G. J. Holzmann. We have extended Spin to RT–Spin, which performs reachability analysis on the DBM automaton, using as input an RT–Promela program.

We have tested our implementation using a number of examples. We now present three of them. The first one models a simple system of three processes representing a train, a gate, and a controller. The second is a real–time mutual–exclusion protocol, due to Fischer [15]. Both these examples have been taken from [3]. The third has to do with a general–purpose ATM switch [17, 12]. It has been taken from [14], where it has been treated using the selection/resolution model [13] and the tool RT–Cospan [18].

The systems consist of a number of components, modeled as TBA. RT–Promela offers the possibility to use local and global variables, as well as channels. We take advantage of this, and we end up with less components than those described in the original models. For example, we do not need a special automaton to model the global variable in the mutual–exclusion protocol.

The alphabet of each TBA is a set of events. Automata synchronize their actions through shared events. Such an event can occur provided it is enabled in every automaton whose alphabet includes the event. Whenever necessary, synchronization in RT–Promela is done using rendez–vous.

### *6.1 Modeling the systems using RT–Promela*

Train, Gate, Controller (TGC) :

This example deals with an automatic controller that opens and closes a gate at a railway track intersection (see figure 1). Whenever the train

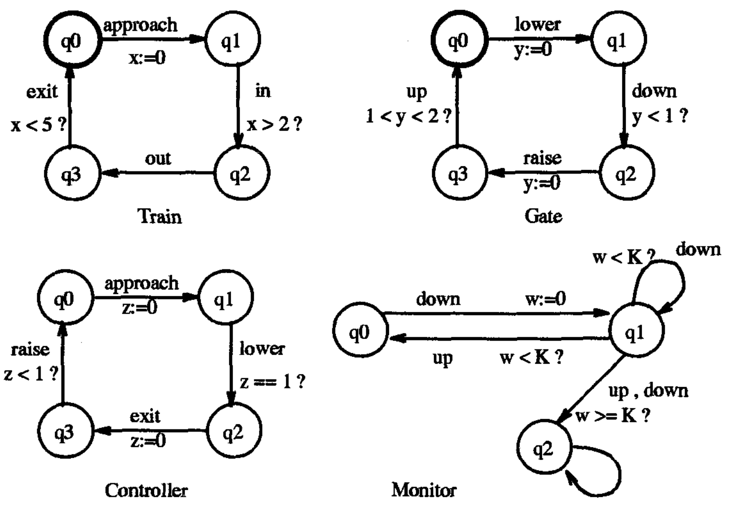

FIGURE 1. Train, Gate, Controller

enters the intersection it sends an approach signal at least two minutes in advance to the controller. The controller also detects the train leaving the intersection and this event occurs within five minutes after it started its approach. The gate responds to lower and raise commands by moving down and up respectively within certain time bounds. The controller sends a lower command to the gate exactly one minute after receiving an approach signal from the train. It commands the gate to raise within one minute of the train's exit from the intersection.

The purpose of the verification is to ensure the following safety property [9] : whenever the gate goes down, it is moved back up within a certain upper time bound $K$. Notice that this implies that the gate *will* eventually come up again. Although this is not immediate from the above property, *liveness* conditions that are associated with each automaton ensure that in every infinite trace, process Gate passes infinitely often from state q0, therefore executing infinitely often the transition $q3 \to q0$ which sets the gate up. Returning to the safety property, the automaton Monitor models precisely the negation of it, as was explained in section 2. The property is satisfied iff the integer constant $K$ is greater than 6.

[9]A *safety* property can be formulated as "never will...". For example "never will processes 1 and 2 be found at their critical sections at the same time"

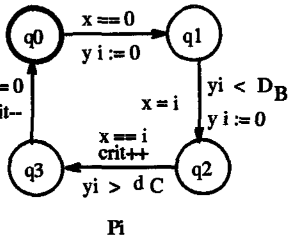

FIGURE 2. Timed mutual exclusion

Timed Mutual Exclusion :

In this protocol, there exist $n$ processes $P_1, ..., P_n$, as shown in figure 2. A process $P_i$ is initially idle, but at any time may begin executing the protocol provided the value of a global variable $x$ is 0. $P_i$ then delays for up to $\Delta_B$ time units before assigning the value $i$ to $x$. It may enter its critical section within $\delta_c$ time units provided the value of $x$ is still $i$. Upon leaving its critical section, it reinitializes $x$ to zero. Global variable crit is used to keep count of the number of processes in the critical section. The auto–increment (auto–decrement) of the variable is done simultaneously with the test (reset of x to zero). This is modeled in RT–Promela using atomic sequences. We need to verify that no two processes are ever in their critical sections at the same time. The property is satisfied iff $\Delta_B > \delta_c$.

A remark needs to be made concerning synchronization between more than two processes. In this case, two or more channels are necessary. The trick is to build a *chain reaction* of receive/send atomic moves in order to propagate the system event to all processes. The method is presented in appendix 1.2 through an example.

Verifying the round–trip delay of an ATM switch :

An ATM switch is a chip used as part of the *Asynchronous Transfer Mode* network protocol for *Broadband Interactive Services Data Networks* (B–ISDN). It consists of four input and four output links, each one of 400 Mbits/sec bandwidth. In ATM, information is transferred in *cells* of fixed length (53 bytes). These cells are routed using *virtual circuits* [10] (VCs), which have different priorities. The *flow–control* mechanism uses a special packet called *token*, which signals to the sender that the receiver is ready to accept a new high–priority cell. Each chip has a flow–control buffer storing the incoming tokens, as well as a cell buffer, used to store the incoming high–priority cells.

[10]There exist also *virtual paths*, which are collections of VCs, but will be ignored, since the chip itself cannot distinguish them from VCs

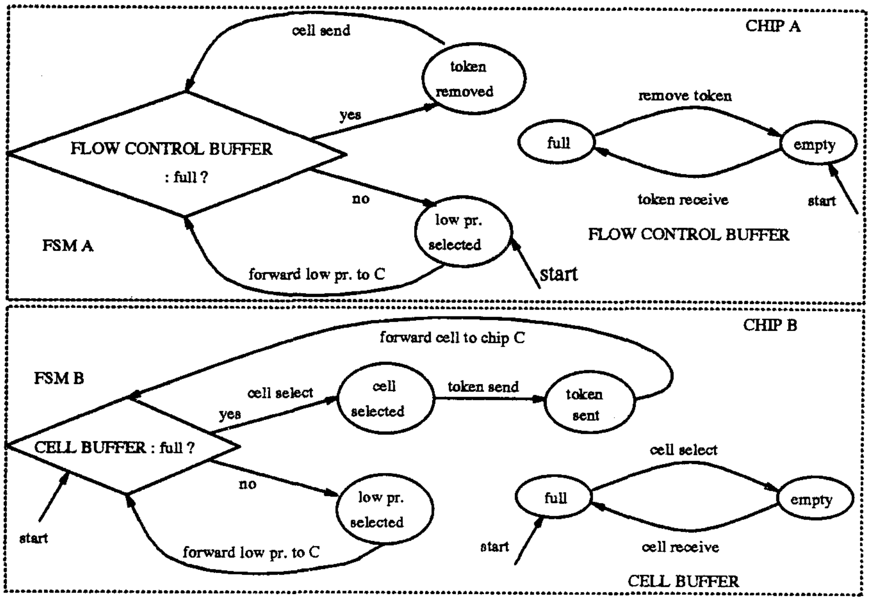

FIGURE 3. Two ATM switches

In our simplified example, we assume two adjucent chips, A and B (figure 3). We are interested in computing the *round–trip* delay. This is defined in [17, 12] as "the delay between the start of two consecutive transmissions of cells of the highest priority", since the chips deal with VCs of different priorities. We make two assumptions :

1. Chip A has always a high priority cell waiting to be transmitted to B.
2. A high priority cell which is sent from B to C is not flow–controlled by C, that is, chip C is always ready to receive it from B.

The first hypothesis allows us to ignore the cell buffer of chip A, while the second allows us to ignore the flow–control buffer of B. The timing assumptions of the system are the following :

1. each chip operates on a cycle of 54 clock *ticks*, that is, between any two packet transmissions, there is a delay of exactly 54 ticks ;
2. the delay between the selection of the next packet to be send and its transmission is between 4 and 10 ticks ;
3. the transmission of a token takes between 11 and 33 ticks ;
4. the transmission of a high priority cell takes exactly 3 ticks ;

| K | states | transitions | DBMs | time | memory |
|---|---|---|---|---|---|
| 5 (erroneous) | 26 | 33 | 21 | 0.2 | 0.8 |
| 7 (correct) | 32 | 34 | 23 | 0.1 | 0.9 |

TABLE 1. Results of train, gate, controller

Based on the above assumptions, we prove that the round–trip delay is never greater than 108 clock ticks, while it cannot be less than 54 ticks. In other words, at most one low priority packet gets transmitted between any two successive transmissions of high priority cells. The specification in RT–Promela can be found in appendix 1.1.

## *6.2 Proving safety properties*

We used three different methods to verify the properties of the systems above. In the case of TGC, the monitor process moves to an error state marked with an accept label, where it stays forever. We ran the validator with the "–a" option to search for acceptance cycles. Notice that in this simple case, maximization wasn't necessary. Performance results are presented in table 2.

The specification of fischer's mutual–exclusion protocol includes a never claim monitoring the system and announcing an error if it finds out that more than one processes are in the critical section. We verified the correctness of the protocol when $\Delta_B = 1$, $\delta_c = 2$ for up to 4 processes, while in the case of 5 the validator refused to terminate. On the other hand, the erroneous case ($\Delta_B = 2$, $\delta_c = 1$) is very little affected by the size of the problem, since the error is found and announced early on. We managed to ran the wrong case for more than 23 processes. Performance results are shown in table 2[11].

Finally, for the ATM switch, we make use of a clock RT which keeps count of the round–trip delay, and test the value of the clock each time a new high–priority cell is sent. If RT is between 54 and 108, it is reset to zero and the system continues normally. If the clock is strictly less than 54 or greater then 108, the error is announced. Performance results are shown in table 2.

Time is measured in seconds, and memory in megabytes.

[11]The erroneous version is noted with $\star$. Non-termination is noted with $\perp$. The numbers in parentheses are those obtained up to the point where the program has been stopped.

| N | version | states | transitions | DBMs | time | mem. |
|---|---|---|---|---|---|---|
| 23 | ⋆ | 327 | 352 | 172 | 10.1 | 1.7 |
| 5 | ⋆ | 93 | 100 | 46 | 0.1 | 0.9 |
| | | ⊥ (825,610) | ⊥ (1,269,690) | ⊥ (32,265) | ⊥ (1,724) | ⊥ (6) |
| 4 | ⋆ | 80 | 86 | 39 | 0.1 | 0.8 |
| | | 28,254 | 43,490 | 1,869 | 37.5 | 2.2 |
| 3 | ⋆ | 67 | 72 | 32 | 0.1 | 0.8 |
| | | 974 | 1,385 | 127 | 0.4 | 1.2 |
| 2 | ⋆ | 54 | 58 | 17 | 0.1 | 0.8 |
| | | 54 | 70 | 13 | 0.1 | 0.8 |

TABLE 2. Results of Fischer's mutual–exclusion protocol

| states | transitions | DBMs | time | memory |
|---|---|---|---|---|
| 435 | 619 | 510 | 2.6 | 1.5 |

TABLE 3. Results of round–trip delay verification

## 7 Conclusions

We have presented the theory and practice of the extensions made to Promela to include real–time semantics. The extended language, RT–Promela, allows for a special kind of global variables, which represent the clocks of the system. The statements of the language can contain simple linear constraints which restrict the possible values of a clock, or the relative values of two clocks. This changes the executability semantics of a statement, which can which can be executed only if, in addition to the restrictions imposed by standard Promela, the constraints do not come against the current state of the clocks. The execution of a statement can affect a clock by resetting it to zero.

The semantics of a specification in RT–Promela are given in terms of timed transition systems. The problem of verification is reduced to checking if the set of all possible valid paths (that is, the language of the system) is empty.

This time model permits the specification of a large class of real–time systems. We illustrate its power by three examples, which have been already considered in the bibliography, thus, allow for comparisons.

Putting clocks in an untimed specification usually increases the size of the

| model | states | transitions | DBMs | time | memory |
|---|---|---|---|---|---|
| untimed | 1,298 | 3,314 | — | 0.2 | 1 |
| timed | 19,197 | 47,180 | 1,225 | 14 | 2.3 |

TABLE 4. Results of leader–election protocol

state space. There are cases where timing constraints restrict the number of possible behaviors of the system, thus creating less states than the untimed model. However, most of the times, the size is significantly increased, up to one or two orders of magnitude, as it is shown in table 2, where we compare the results obtained by an exhaustive dfs performed on an untimed and a timed model of the leader–election protocol [7].

Consequently, future work mainly concerns the research for methods of *reduction*. The size of larger examples makes their analysis prohibitive. The new version of untimed Spin implements the *partial–order* method presented in [16, 11]. It would be interesting to see whether this reduction preserves time properties, and under which conditions. Apart from the above, older methods for on–the–fly *minimization* of the state space exist [4] and should be also tried out.

## 8 References

[1] R. Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, June 1990.

[3] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *RTSS 1992, proceedings*, 1992.

[4] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. Minimization of timed transition systems. In *CONCUR 1992, proceedings.* Lecture Notes in Computer Science, Springer-Verlag, 1992.

[5] C. Courcoubetis, D. Dill, M. Chatzaki, and P. Tzounakis. Verification with real-time COSPAN. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[6] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. Workshop on Computer Aided Verification, CAV89*, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[7] Dolev, Klawe, and Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *J. of Algs*, 3:245–260, 1982.

[8] G.J. Holzmann. *Design and Validation of Protocols.* Prentice-Hall, 1990.

[9] G.J. Holzmann. Basic spin manual. Technical report, AT&T, Bell Laboratories, 1994.

[10] G.J. Holzmann. What's new in spin. Technical report, AT&T, Bell Laboratories, 1995.

[11] G.J. Holzmann and Doron A. Peled. An improvement in formal verification. In *Proceedings of the 7th International Conference on Formal Description Techniques, FORTE94*, Berne, Switcherland, october 1994.

[12] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general purpose ATM switch chip. *IEEE JSAC*, 9(8):1265–1279, 1991.

[13] J. Katzenelson and B. Kurshan. S/r: A language for specifying protocols and other coordinating processes. In *Proc. 5th Ann. Int'l Phoenix Conf. Comput. Commun., IEEE*, 1986.

[14] N. Lambrogeorgos. Verification of real-time systems: a case study of discrete and dense time models, 1993. Available only in greek.

[15] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[16] Doron A. Peled. Combining partial order reductions with on–the–fly model checking. In *Proceedings of the 6th International Conference on Comptuter Aided Verification, CAV94*, Stanford, California, june 1994.

[17] S. Sidiropoulos. A general purpose ATM switch. architecture and feasibility study, 1991.

[18] P. Tzounakis. Verification of real time systems: The extension of COSPAN in dense time, 1992.

## Acknowledgments

We wish to thank Gerard Holzmann for providing numerous answers regarding the implementation, as well as for attentively reading the paper and making useful remarks.

# 1 Appendix

## 1.1 *Mutual exclusion in RT–Promela*

```
#define N 5 /* number of processes */
#define deltaB 1
#define deltaC 2
#define ErRoR assert(0)
clock y[N];
int x, crit;

proctype P ( byte id )
{
      do ::
            reset{y[id]} x==0 ->
            when{y[id]<deltaB} reset{y[id]} x=id+1 ->
            atomic{ when{y[id]>deltaC} x==id+1; crit++; } ->
            atomic{ x=0; crit−−; }
      od
}

never{
      skip -> /* to let the processes be activated */
      do
            :: crit>1 -> ErRoR
            :: else
      od
}

init {
      byte proc;
      atomic {
            crit = 0;
            proc = 1;
            do
            :: proc ≤ N ->
                        run P ( proc%N );
                        proc = proc+1
            :: proc > N -> break
            od
      }
}
```

## *1.2 Multi-way synchronization in RT-Promela*

Assume that four processes wish to synchronize on the signal sync. Then we have to use three rendez-vous channels and the following specification :

```
chan A,B,C = [0] of { byte };
proctype P1() {   ...     q1: A!sync - >   ... }
proctype P2() {   ...     q2: atomic { A?sync - > B!sync }   ... }
proctype P3() {   ...     q3: atomic { B?sync - > C!sync }   ... }
proctype P4() {   ...     q4: C?sync - >   ... }
```

In the case of synchronization of timed statements, the constraints and resets of all of them are grouped together, as if it were a single statement executed instantaneously.

## *1.3 DBMs*

DBMs

We consider the class $\mathcal{CR}$ of convex polyhedra in $\mathsf{R}^{|C|}$ which can be defined by a set of integer constraints on clocks and clock differences. If we identify a new fictitious clock $x_0$ with the constant value 0, the above constraints can be represented as bounds on the difference between two clock values. For instance, $x < 5$ can be expressed as $x - x_0 < 5$, and $x > 5$ as $x_0 - x < -5$. Furthermore, we can introduce $\infty$ as a bounding value, to represent inequalities of the form $x < y$ (we write $x - y < \infty$), and $-\infty$ to express $false$ (we write $x - y < -\infty$). Thus, we can restrict ourselves to upper bounds without loss of generality. More precisely, each inequality can be expressed as : $x_i - x_j < k$ or $x_i - x_j \leq k$, for some integer $k$, or $x_i - x_j < \infty$, or $x_i - x_j < -\infty$.

A DBM is an $(n+1) \times (n+1)$ matrix $D$, whose elements (called ***bounds***) are of the form : $d_{ij} = (r, \#)$, $r \in \mathsf{R} \cup \{\infty\}$, $\# \in \{<, \leq\}$. $D$ represents the polyhedron of $\mathsf{R}^n$ consisting of all points that satisfy the inequality $x_i - x_j \# r$, where $d_{ij} = (r, \#)$.

Canonicalization

There are possibly many DBMs defining the same clock region, because some of the upper bounds need not be tight [12]. For example, $\{x_1 < 2, x_1 \geq 1, x_2 \leq 5\}$ can be represented by any matrix $D$ such that $d_{01} = (-1, \leq), d_{02} = (0, \leq), d_{10} = (2, <), d_{20} = (5, \leq)$, and $d_{12} \in \{(2,<),(2,\leq),(3,<),(3,\leq),...\}, d_{21} \in \{(4,<),(4,\leq),(5,<),(5,\leq),...\}$.

Then, the idea is to represent $D$ in a *canonical* form, where all upper bounds are as "tight" as possible. We denote this canonical matrix cf($D$). Dill [6] showed that cf($D$) can be computed from $D$ by applying an all-pairs

[12]Bounds are ordered lexicographically ($<$ is taken to be strictly less than $\leq$), that is : $(r, \#) < (r', \#')$ iff $(r < r') \vee (r = r' \wedge \# =< \wedge \#' =\leq)$.

shortest-path algorithm. Moreover, canonicalization leads to easy tests for equality and emptiness of clock regions. A matrix $D$ represents an empty region if a negative-cost cycle (i.e., $(-\infty, <)$) appears during the computation of cf($D$).

### Elapse of time

As time elapses, clock differences remain the same, since all clocks increase at the same rate. Lower bounds do not change either since there are no decreasing clocks. Upper bounds have to be canceled, since an arbitrary period of time may pass. Let $CR$ be the clock region represented by DBM $D$, and $CR^\delta$ the one represented by $D^\delta$. Then :

$$\begin{array}{ll} d^\delta_{i0} = (\infty, <) & \text{for all } i = 1, 2, ..., |C| \\ d^\delta_{ij} = d_{ij} & \text{otherwise} \end{array}$$

Then it is easy to see that for each equivalence class $\alpha$ such that $\alpha \cap CR \neq \emptyset$, if $\alpha'$ is a time successor of $\alpha$ then $\alpha' \cap CR^\delta \neq \emptyset$.

### Intersection and union

Let $D$ and $D'$ be DBMs for $CR$ and $CR'$. The intersection $CR \cap CR'$ (resp. union $CR \cup CR'$) is represented by the matrix $D^\cap$ (resp. $D^\cup$) such that $\forall i, j$, $d^\cap_{ij} = min\{d_{ij}, d'_{ij}\}$ (resp. $d^\cup_{ij} = max\{d_{ij}, d'_{ij}\}$).

### Clock resets

Let $CR$ be the clock region represented by DBM $D$, and $R \subseteq C$. Then $CR[R := 0]$ is represented by $D'$, defined as follows :

$$\begin{array}{ll} d'_{i0} = d_{0i} = (0, \leq) & \text{if } i \in R \\ d'_{ij} = d_{ji} = (\infty, <) & \text{if } i \in R \text{ and } j \in C \setminus R \\ d'_{ij} = d_{ji} = (0, \leq) & \text{if } i, j \in R \\ d'_{ij} = d_{ji} & \text{otherwise} \end{array}$$

### Maximization

Let $c^1_{ij}, ..., c^m_{ij}$ be an increasing sequence of bounds, where $c^k_{ij} = (r_k, \#_k)$ corresponds to an atom $x_i - x_j \#_k r_k$ appearing in the program (if no such atom exists, let $m = 1$, $c^1_{ij} = (\infty, <)$). For a DBM $D$, if $max(D) = D'$, then $d'_{ij} = c^k_{ij}$, such that $d_{ij} < c^k_{ij}$, $\forall l < k$, $c^l_{ij} < d_{ij}$, and $\forall l' > k$, $d_{ij} < c^{l'}_{ij}$.