FORMATS 2006, 4th Int Conf on Formal Modelling and Analysis of Timed Systems
Proceedings: Eugene Asarin, Patricia Bouyer (eds.)
Springer Lecture Notes in Computer Science **4202** (2006), 98–112

Matching scenarios with timing constraints [⋆]

Prakash Chandrasekaran and Madhavan Mukund

Chennai Mathematical Institute, Chennai, India
{prakash,madhavan}@cmi.ac.in

Abstract. Networks of communicating finite-state machines equipped with local clocks generate timed MSCs. We consider the problem of checking whether these timed MSCs are "consistent" with those provided in a timed MSC specification. In general, the specification may be both positive and negative. The system should execute all positive scenarios "sensibly". On the other hand, negative scenarios rule out some behaviours as illegal. This is more complicated than the corresponding problem in the untimed case because even a single timed MSC specification implicitly describes an infinite family of timed scenarios. We outline an approach to solve this problem that can be automated using Uppaal.

1 Introduction

In a distributed system, several agents interact with each other to generate a global behaviour. The interaction between these agents is usually described in terms of scenarios, using mechanisms such as use-cases and message sequence charts (MSCs) [8].

In general, scenarios could be of two types, positive and negative. Positive scenarios are those that the system is designed to execute—for instance, these may describe a handshaking protocol to set up a reliable communication channel between two hosts on a network. Negative scenarios indicate undesirable behaviours, such as a situation when both hosts independently initiate the activity of setting up a channel, leading to a collision.

This leads to a natural verification problem: given a distributed system and a scenario, does the system exhibit the scenario? In the context of message sequence charts, this is referred to as the scenario matching problem, for which efficient algorithms have been identified [10]. An approach to solve this problem using the modelchecker Spin was proposed in [4].

In this paper, we extend the study of scenario matching to timed systems. We consider communicating finite-state machines equipped with local clocks. Clock constraints are used to guard transitions and specify location invariants, as in other models of timed automata [3]. Just as the runs of timed automata can be described in terms of timed words, the interactions exhibited by communicating finite-state machines with clocks can be described using timed MSCs.

⋆ Partially supported by *Timed-DISCOVERI*, a project under the Indo-French Networking Programme.

We define a version of scenarios with timing constraints that we call timed MSC templates. These templates are built from fixed underlying MSCs by associating a lower and upper bound on the time interval between certain pairs of events. Timed MSC templates are a natural and useful extension of the untimed notation for scenarios, because protocol specifications typically include timing requirements for message exchanges, as well as descriptions of how to recover from timeouts.

In general, a timed MSC template is compatible with infinitely many timed MSCs. Thus, the scenario matching problem is already more complicated than in the untimed case, where a single scenario describes exactly one pattern of interaction. In our setting, the scenario matching problem can be reformulated in terms of checking whether the intersection of two collections of timed MSCs is nonempty.

We propose an approach to tackle this problem using the modelchecking tool Uppaal, which is designed to verify properties of timed systems. Unfortunately, the basic system model of Uppaal consists of a network of timed automata that communicate via synchronous handshakes, rather than message-passing. We thus need to code up message-passing channels by creating special processes to model buffers. However, we can exploit the handshake mechanism to synchronize the system with the template to be verified. This automatically reduces the behaviours of the system to those that are consistent with the template. The scenario matching problem can then be easily transformed into a modelchecking question for Uppaal to verify on the composite system.

The paper is organized as follows. In the next two sections, we formally define timed MSCs and timed message-passing automata. This enables us to precisely define the scenario matching problem for timed systems in Section 4. In Section 5, we describe our approach to address the scenario matching problem in Uppaal. Next, we look at issues related to the expressiveness of our timed message-passing automaton model. In Section 7, we examine some optimizations that can be introduced when translating the scenario matching problem to Uppaal, to improve the efficiency of verification. We conclude with a brief discussion.

2 Timed MSCs

2.1 Message sequence charts

Let $\mathcal{P} = \{p, q, r, \ldots\}$ be a finite set of processes (agents) that communicate with each other through messages via reliable FIFO channels using a finite set of message types $\mathcal{M}$. For $p \in \mathcal{P}$, let $\Sigma_p = \{p!q(m), p?q(m) \mid p \neq q \in \mathcal{P}, m \in \mathcal{M}\}$ be the set of communication actions in which p participates. The action $p!q(m)$ is read as *p sends the message m to q* and the action $p?q(m)$ is read as *p receives the message m from q*. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$. We also denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$.

Labelled posets A Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$, let ${\downarrow}e = \{e' \mid e' \leq e\}$.

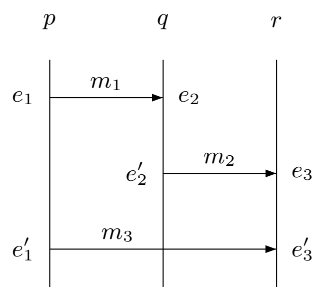

Fig. 1. An MSC over $\{p, q, r\}$.

For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$ and $E_a = \{e \mid \lambda(e) = a\}$, respectively. For each $(p, q) \in Ch$, we define the relation $<_{pq}$ as follows:

$$e <_{pq} e' \iff \lambda(e) = p!q(m),\ \lambda(e') = q?p(m) \text{ and } |{\downarrow}e \cap E_{p!q(m)}| = |{\downarrow}e' \cap E_{q?p(m)}|$$

The relation $e <_{pq} e'$ says that channels are FIFO with respect to each message—if $e <_{pq} e'$, the message m read by q at e' is the one sent by p at e.

Finally, for each $p \in \mathcal{P}$, we define the relation $\leq_{pp} = (E_p \times E_p) \cap \leq$, with $<_{pp}$ standing for the largest irreflexive subset of $\leq_{pp}$.

Definition 1. *An MSC (over $\mathcal{P}$) is a* finite *Σ-labelled poset $M = (E, \leq, \lambda)$ that satisfies the following conditions.*

1. *Each relation $\leq_{pp}$ is a linear order.*
2. *If $p \neq q$ then for each $m \in \mathcal{M}$, $|E_{p!q(m)}| = |E_{q?p(m)}|$.*
3. *If $e <_{pq} e'$, then $|{\downarrow}e \cap \left(\bigcup_{m\in\mathcal{M}} E_{p!q(m)}\right)| = |{\downarrow}e' \cap \left(\bigcup_{m\in\mathcal{M}} E_{q?p(m)}\right)|$.*
4. *The partial order $\leq$ is the reflexive, transitive closure of the relation $\bigcup_{p,q\in\mathcal{P}} <_{pq}$.*

The second condition ensures that every message sent along a channel is received. The third condition says that every channel is FIFO across all messages.

In diagrams, the events of an MSC are presented in *visual order*. The events of each process are arranged in a vertical line and messages are displayed as horizontal or downward-sloping directed edges. Figure 1 shows an example with three processes $\{p, q, r\}$ and six events $\{e_1, e'_1, e_2, e'_2, e_3, e'_3\}$ corresponding to three messages—m_1 from p to q, m_2 from q to r and m_3 from p to r.

For an MSC $M = (E, \leq, \lambda)$, we let $\text{lin}(M) = \{\lambda(\pi) \mid \pi$ is a linearization of $(E, \leq)\}$. For instance, $p!q(m_1)\ q?p(m_1)\ q!r(m_2)\ p!r(m_3)\ r?q(m_2)\ r?p(m_3)$ is one linearization of the MSC in Figure 1.

2.2 Timed MSC templates

A timed MSC template is an MSC annotated with time intervals between pairs of events along a process line. For instance, consider the interaction between a

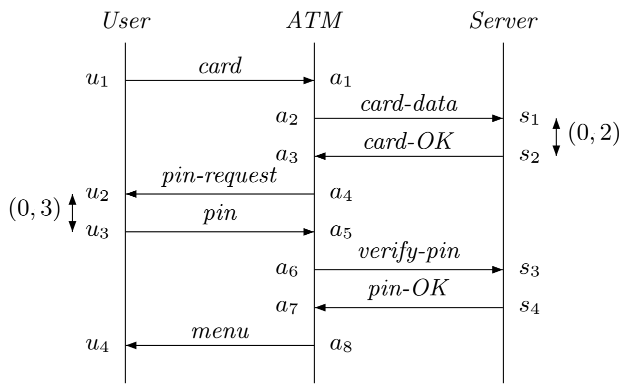

Fig. 2. A timed MSC template describing interaction with an ATM.

user, an ATM and a server depicted in Figure 2. This MSC has sixteen events generated by eight messages. The events u_2 and u_3 are linked by a time interval $(0, 2)$, as are the events s_2 and s_3. These time intervals represent constraints on the delay between the occurrences of the events. Thus, this template specifies that the server is expected to respond to a request to authenticate an ATM card within 2 units of time. Similarly, a user has to type in his PIN within 3 units of time of the ATM requesting the PIN.

For simplicity, we assume that time intervals are bounded by natural numbers. A pair of time points (m, n), $m, n \in \mathbb{N}$, $m \leq n$, denotes the time interval $\{x \in \mathbb{R}_{\geq 0} \mid m \leq x \leq n\}$.

Definition 2. *Let $M = (E, \leq, \lambda)$ be an MSC. An* interval constraint *is a tuple $\langle (e_1, e_2), (t_1, t_2) \rangle$, where:*

- *$e_1, e_2 \in E$ with $e_1 \leq_{pp} e_2$ for some $p \in \mathcal{P}$.*
- *$t_1, t_2 \in \mathbb{N}$ with $t_1 \leq t_2$.*

The restriction on the relationship between e_1 and e_2 ensures that an interval constraint is local to a process.

Definition 3. *A* timed MSC template *is pair $\mathcal{T} = (M, \mathcal{I})$ where $M = (E, \leq, \lambda)$ is an MSC and $\mathcal{I} \subseteq (E \times E) \times (\mathbb{N} \times \mathbb{N})$ is a set of interval constraints.*

2.3 Timed MSCs

In a timed MSC, events are explicitly time-stamped so that the ordering on the time-stamps respects the partial order on the events.

Definition 4. *A* timed MSC *is pair (M, τ) where $M = (E, \leq, \lambda)$ is an MSC and $\tau : E \longrightarrow \mathbb{R}_{\geq 0}$ assigns a nonnegative time-stamp to each event, such that for all $e_1, e_2 \in E$, if $e_1 \leq e_2$ then $\tau(e_1) \leq \tau(e_2)$.*

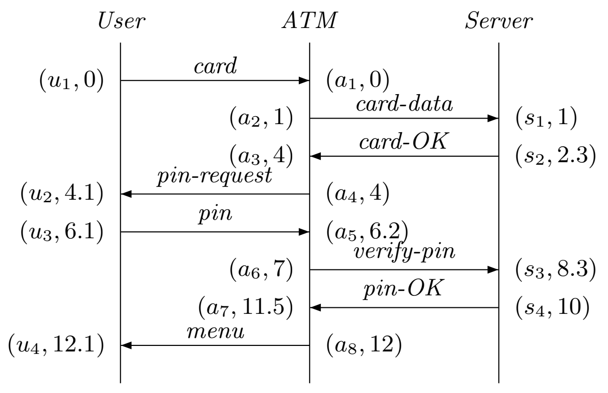

Fig. 3. A timed MSC instance describing interaction with an ATM.

A timed MSC satisfies a timed MSC template if the time-stamps assigned to events respect the interval constraints specified in the template.

Definition 5. *Let $M = (E, \leq, \lambda)$ be an MSC, $\mathcal{T} = (M, \mathcal{I})$ a timed template and $M_\tau = (M, \tau)$ a timed MSC. M_τ is said to* satisfy *$\mathcal{T}$ if the following holds*

$$\text{For each } \langle (e_1, e_2), (t_1, t_2) \rangle \in \mathcal{I}, t_1 \leq \tau(e_2) - \tau(e_1) \leq t_2.$$

Definition 6. *Let $\mathcal{T}$ be a timed MSC template. We denote by $L(\mathcal{T})$ the set of timed MSCs that satisfy $\mathcal{T}$.*

Figure 3 shows a timed MSC that satisfies the template in Figure 2.

Let $M_\tau = (M, \tau)$ be a timed MSC, where $M = (E, \leq, \lambda)$, and let $\pi = e_0 e_1 \ldots e_m$ be a linearization of $(E, \leq)$. By labelling each event with its timestamp, this linearization gives rise to a timed linearization $(e_0, \tau(e_0))(e_1, \tau(e_1)) \cdots (e_n, \tau(e_n))$. As is the case with untimed MSCs, under the FIFO assumption for channels, a timed MSC can be faithfully reconstructed from any one of its timed linearizations.

3 Timed Message-Passing Automata

Message-passing automata are a natural machine model for generating MSCs. We extend the definition used in [6] to include clocks.

Definition 7. *Let $\mathcal{C}$ denote a finite-set of real-valued variables called* clocks. *A* clock constraint *is a conjunctive formula of the form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}$, $n \in \mathbb{N}$ and $\sim \in \{\leq, <, =, >, \geq\}$. Let $\Phi(\mathcal{C})$ denote the set of clock constraints over the set of clocks $\mathcal{C}$.*

Clock constraints will be used as guards and location invariants in timed message-passing automata.

Definition 8. *A* clock assignment *for a set of clocks $\mathcal{C}$ is a function $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ that assigns a nonnegative real value to each clock in $\mathcal{C}$.*

A clock assignment v is said to satisfy *a clock constraint φ if φ evaluates to true when we substitute for each clock c mentioned in φ the corresponding value $v(c)$.*

Let $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ be a clock assignment. For $d \in \mathbb{R}_{\geq 0}$, we write $v + d$ to denote the clock assignment that maps each $x \in \mathcal{C}$ to $v(x) + d$. For $X \subseteq \mathcal{C}$, we write $v[X \leftarrow 0]$ to denote the clock assignment that agrees with v for all clocks in $\mathcal{C} \setminus X$ and maps all clocks in X to 0.

Definition 9. *A* timed message-passing automaton (timed MPA) *over Σ with a set of clocks $\mathcal{C}$ is a structure $\mathcal{A} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \Sigma, \mathcal{C})$. Each component $\mathcal{A}_p$ is of the form $(S_p, S^p_{in}, \to_p, I_p)$, where:*

- *S_p is a finite set of p-local states.*
- *$S^p_{in} \subseteq S_p$, is a set of initial states for p.*
- *$\to_p \subseteq S_p \times \Phi(\mathcal{C}) \times \Sigma_p \times 2^{\mathcal{C}} \times S_p$ is the p-local transition relation.*
- *$I_p : S \to \Phi(\mathcal{C})$ assigns an* invariant *to each state.*

The local transition relation $\to_p$ specifies how the process p sends and receives messages.

The transition $(s, \varphi, p!q(m), X, s')$ says that in state s, p can send the message m to q and move to state s'. This transition is *guarded* by the clock constraint φ—the transition is enabled only when the current values of all the clocks satisfy φ. The set X specfies the clocks whose values are reset to 0 when this transition is taken.

Similarly, the transition $(s, \varphi, p?q(m), X, s')$ signifies that at state s, p can receive the message m from q and move to state s' provided the current clock values satisfy φ. Once again, all clocks in X are reset to 0.

A process can remain in a state s only if the current values of all the clocks satisfy the invariant $I(s)$. To make our model amenable for automated verification, we restrict location invariants to constraints that are downward closed—that is, constraints of the form $x \leq n$ or $x < n$, where x is a clock and n is a natural number.

As is customary with timed automata, we allow timed MPA to perform two types of moves: moves where the automaton does not change state and time elapses, and moves where some local component p changes state instantaneously as permitted by $\to_p$.

A global state of $\mathcal{A}$ is an element of $\prod_{p\in\mathcal{P}} S_p$. For a global state $\overline{s}$, $\overline{s}_p$ denotes the pth component of $\overline{s}$. A *configuration* is a triple $(\overline{s}, \chi, v)$ where $\overline{s}$ is a global state, $\chi : Ch \to \mathcal{M}^*$ is the *channel state* describing the message queue in each channel c and $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is a clock assignment. An *initial configuration* of $\mathcal{A}$ is of the form $(\overline{s}_{in}, \chi_\varepsilon, v_0)$ where $\overline{s}_{in} \in \prod_{p\in\mathcal{P}} S^p_{in}$, $\chi_\varepsilon(c)$ is the empty string ε for every channel c and $v_0(x) = 0$ for every $x \in \mathcal{C}$.

The set of reachable configurations of $\mathcal{A}$, $Conf_{\mathcal{A}}$, is defined inductively, together with a transition relation $\Longrightarrow \subseteq Conf_{\mathcal{A}} \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Conf_{\mathcal{A}}$.

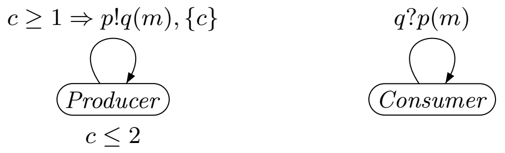

Fig. 4. A timed MPA: producer-consumer

- Every initial configuration $(\overline{s}_{in}, \chi_\varepsilon, v_0)$ is in $Conf_{\mathcal{A}}$.
- If $(\overline{s}, \chi, v) \in Conf_{\mathcal{A}}$ and $d \in \mathbb{R}_{\geq 0}$ such that v and $v + d$ satisfy the invariants $\{I_p(\overline{s}_p)\}_{p \in \mathcal{P}}$, then there is a global move $(\overline{s}, \chi, v) \overset{d}{\Longrightarrow} (\overline{s}, \chi, v + d)$ and $(\overline{s}, \chi, v + d) \in Conf_{\mathcal{A}}$.
- If $(\overline{s}, \chi, v) \in Conf_{\mathcal{A}}$ and $(\overline{s}_p, \varphi, p!q(m), X, \overline{s}'_p) \in \rightarrow_p$ such that v satsfies φ, there is a global move $(\overline{s}, \chi, v) \overset{p!q(m)}{\Longrightarrow} (\overline{s}', \chi', v[X \leftarrow 0])$ with $(\overline{s}', \chi', v[X \leftarrow 0]) \in Conf_{\mathcal{A}}$, where, for $r \neq p$, $\overline{s}_r = \overline{s}'_r$, $\chi'((p, q)) = \chi((p, q)) \cdot m$, and for $c \neq (p, q)$, $\chi'(c) = \chi(c)$.
- Similarly, if $(\overline{s}, \chi, v) \in Conf_{\mathcal{A}}$ and $(\overline{s}_p, \varphi, p?q(m), X, \overline{s}'_p) \in \rightarrow_p$ such that v satisfies φ, there is a global move $(\overline{s}, \chi, v) \overset{p?q(m)}{\Longrightarrow} (\overline{s}', \chi', v[X \leftarrow 0])$ with $(\overline{s}', \chi', v[X \leftarrow 0]) \in Conf_{\mathcal{A}}$, where, for $r \neq p$, $\overline{s}_r = \overline{s}'_r$, $\chi((q, p)) = m \cdot \chi'((q, p))$, and for $c \neq (q, p)$, $\chi'(c) = \chi(c)$.

Let $\mathrm{prf}(\sigma)$ denote the set of prefixes of a timed word $\sigma = (a_1, t_1)(a_2, t_2) \ldots (a_k, t_k) \in (\Sigma \times \mathbb{R}_{\geq 0})^*$. A run of $\mathcal{A}$ over σ is a map $\rho : \mathrm{prf}(\sigma) \rightarrow Conf_{\mathcal{A}}$ such that $\rho(\varepsilon)$ is assigned an initial configuration $(\overline{s}_{in}, \chi_\varepsilon, v_0)$ and for each $\sigma' \cdot (a_i, t_i) \in \mathrm{prf}(\sigma)$, $\rho(\sigma') \overset{d_i}{\Longrightarrow}\overset{a_i}{\Longrightarrow} \rho(\sigma' \cdot (a_i, t_i))$ with $t_i = t_{i-1} + d_i$ and t_0 implicitly set to 0.

The run ρ is *complete* if $\rho(\sigma) = (s, \chi_\varepsilon, v)$ is a configuration in which all channels are empty. When a run on σ is complete, σ is a timed linearization of a timed MSC. We define $L(\mathcal{A}) = \{\sigma \mid \mathcal{A} \text{ has a complete run over } \sigma\}$. Thus, $L(\mathcal{A})$ corresponds to the set of timed linearizations of a collection of timed MSCs.

Figure 4 is a simple example of a timed MPA. Here, the traditional producer-consumer system is augmented with a clock c in the producer process. The constraint $c \geq 1$ on the transition ensures that each new message is generated by the producer at least one unit of time after the previous one. The location invariant $c \leq 2$ forces the producer to generate a new message no later than two units of time after the previous one. The consumer process has no timing constraints. Figure 5 shows a typical timed MSC generated by the timed MPA in Figure 4.

4 Verifying timed scenarios

Given a timed MSC template $\mathcal{T}$ and a timed MPA $\mathcal{A}$, the verification question that we address is whether $\mathcal{A}$ exhibits any timed scenario that is consistent with $\mathcal{T}$. In other words, we would like to check that $L(\mathcal{T}) \cap L(\mathcal{A})$ is nonempty.

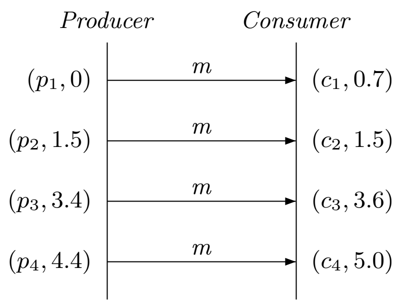

Fig. 5. A timed MSC generated by the producer-consumer system.

More generally, we identify a state property α that we would like the system to satisfy at the end of the template. We would then like to check that every timed MSC $M_{\mathcal{T}} \in L(\mathcal{T}) \cap L(\mathcal{A})$ satisfies α.

For instance, in the ATM example, suppose the template to be verified is the one in Figure 2. We could associate with this template the condition that the *User* is in a state where it can select an item from the menu. We would then insist that every timed MSC that satisfies this template leaves the *User* in the appropriate state.

Sometimes, it is fruitful to describe forbidden scenarios as timed templates. Let $\mathcal{T}$ be such a *negative* template. We then want to check that a timed MPA $\mathcal{A}$ does *not* exhibit a timed scenario consistent with $\mathcal{T}$. In other words, we would like $L(\mathcal{T}) \cap L(\mathcal{A})$ to be empty.

Instead of looking for exact matches, we can also consider the scenario verification problem modulo embedding. The notion of one MSC being embedded in another is the usual one—there is an injective function mapping the messages in the first MSC into the messages in the second MSC that preserves the partial order between the events of the first MSC. Checking for patterns modulo embedding is useful, for instance, when a system introduces auxiliary messages to implement a protocol. For instance, most implementations of the standard telnet protocol exchange additional messages to verify operating system details, even though this is not part of the telnet protocol specification.

When checking for scenarios modulo embedding, we could constrain the nature of the embedding. A *strict* embedding is one in which all additional messages in the target MSC carry labels that are disjoint from those in the source. In other words, if we restrict the target MSC to the message alphabet of the source MSC, we obtain an exact match. In a *weak* embedding, we do not impose this restriction.

For timed MSCs, it seems important to work with weak embeddings. Consider the example in Figure 6. The template we are looking for is a simple message followed by an acknowledgment within 2 units of time, shown on the left. The implementation is designed to resend the message if the acknowledgment does not arrive within the specified time of 2 units. Thus, the implementation may

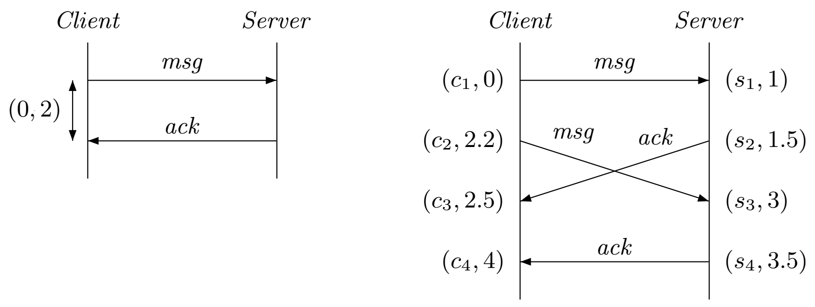

Fig. 6. Weak embedding is required for timed MSCs.

generate a timed MSC such as the one on the right, where the message is sent twice. Nevertheless, it seems reasonable to argue that this system does exhibit a scenario consistent with the timed template.

In the untimed setting, the problem of matching MSCs upto embedding was considered in [10], where it was shown that a straightforward greedy algorithm works under an interpretation in which MSCs are closed with respect to *race conditions* [1], which permits the reordering along a process line of some pairs of events that are not explicitly causally ordered. If one does not assume closure with respect to race conditions, backtracking seems unavoidable in solving the problem [4].

The scenario matching problem for timed MSCs is more complicated than the same problem for untimed MSCs in one obvious way. Even though a timed template is defined with respect to a single underlying MSC, the set of timed MSCs that satisfy a given template is in general infinite. Thus, even with a single template, the matching problem comes down to one of comparing infinite collections of (timed) MSCs.

5 Using Uppaal for scenario verification

In [4], an approach is presented for solving the embedding problem for untimed MSCs using the modelchecker Spin [7]. Message-passing automata are naturally represented in Spin. The strategy suggested by [4] is to augment the underlying system with an auxiliary *monitor process*. Each send and receive action in the system is accompanied by a synchronization action with the monitor process. In this way, the monitor process builds up a global picture of the MSC being generated by the system. When the monitor detects that the given scenario has been embedded, it enters a good state. The scenario embedding problem can then be translated into a standard LTL model-checking problem for the composite system, where the property to be checked includes the condition that the monitor enters a good state.

This approach is not suitable in our setting for two reasons. The first is that Spin has no way of dealing with time. Another complication is that timed MSC templates generate infinite families of timed MSCs. This makes the design of a monitor process more complicated.

Instead, we move over to Uppaal, a modelchecker for timed systems [2]. Uppaal supports the analysis of networks of timed automata for timing properties. Unfortunately, unlike Spin, Uppaal does not have a direct way of modelling asynchronous communication. However, we can simulate asynchronous communication by creating explicit buffer processes. Moreover, we can exploit the synchronous communication paradigm built-in to Uppaal to get around the problem of having a separate process to monitor the communication patterns of the system. Instead, we synchronize the system with the template at each communication action. This allows the system to evolve only along trajectories that are consistent with the template, thus automatically restricting the behaviours of the composite system to those that are of interest.

5.1 Modelling channels in Uppaal

Since Uppaal has no notion of buffered communication, we construct an explicit buffer process for each channel between processes. Message passing is simulated by a combination of shared memory and binary synchronization. Let p and q be processes and let c be the channel between p and q. We create a separate process c which maintains, internally, an array of messages M_{pq} whose size corresponds to the capacity of c. This array is used by c as a circular buffer to store the state of the channel. The process c maintains two pointers into the array: the next free slot into which p can write and the slot at the head of the queue from which q will next read a message.

The channel c shares two variables s_{pc} and r_{cq} with p and q, respectively. These are used to transfer information about the actual message between the processes and the channel. The channel c also uses two special actions a_{pc} and a_{cq} to synchronize with p and q, respectively. These synchronizations represent the actual insertions and deletions of messages into and from the channel.

When p sends a message m to q, it sets the shared variable s_{pc} to m and synchronizes with c on a_{pc}. When c synchronizes with p, it copies the message from s_{pc} into the array slot that currently corresponds to the end of the queue and then increments the free slot pointer to point to the next position in the array.

Symmetrically, when q wants to read a message m from p, it sets the shared variable r_{cq} to m and then synchronizes with c on action a_{cq}. In c, this synchronization is guarded by conditions that check that there is at least one message in the queue and that the message at the head of the queue matches the one q is looking for, as recorded in the shared variable r_{cq}.

5.2 Handling timed MSC templates

To verify timed MSC templates, the first step is to convert such a template into a timed MPA whose language of timed MSCs corresponds to that of the template. Since the template is built from a single MSC, the communication structure of the MPA is fixed and can be computed easily, as explained in [4]. Each time constraint in an MSC template is local to a process. We introduce a new clock for each constraint and add clock constraints using these clocks to guard the actions of the MPA so that it respects the timed template.

5.3 Computing $L(\mathcal{T}) \cap L(\mathcal{A})$

We can now augment the system description in UPPAAL so that the evolution of the system to be verified is controlled by the external template specification. Recall that each action corresponding to sending or receiving a message by a local process is broken up into two steps in the UPPAAL implementation, one which sets the value of a shared variable s_{pc} and another which communicates with the buffer process via a shared action a_{pc}. We extend this sequence to a third action, b_{pc}, by which the system synchronizes with the specification. A move of the form $s \overset{p!q(m)}{\Longrightarrow} s'$ in the original timed MPA now breaks up, in the UPPAAL implementation, into a sequence of three moves $s \overset{s_{pc}=m}{\Longrightarrow} s_1 \overset{a_{pc}}{\Longrightarrow} s_2 \overset{b_{pc}}{\Longrightarrow} s'$. The third action, b_{pc} synchronizes with the corresponding process p in the timed MPA derived from the timed template that is being verified. Thus, the system can progress via this action only if it is consistent with the constraints specified by the template.

Symmetrically, for a receive action of the form $s \overset{p?q(m)}{\Longrightarrow} s'$, the UPPAAL implementation would execute a sequence of the form $s \overset{r_{pc}=m}{\Longrightarrow} s_1 \overset{\bar{a}_{cp}}{\Longrightarrow} s_2 \overset{\bar{b}_{cp}}{\Longrightarrow} s'$, where we use the convention that an action a synchronizes with a matching action $\bar{a}$.

By construction, it now follows that the timed MSCs executed by the composite system are those which are consistent with both the timed template and with the underlying timed MPA being modelled in UPPAAL. Thus, we have restricted the behaviour of the system to $L(\mathcal{T}) \cap L(\mathcal{A})$, for a given timed template $\mathcal{T}$ and a given timed MPA $\mathcal{A}$. From this, it is a simple matter of invoking the UPPAAL modelchecker to verify whether this set of behaviours is empty and whether all behaviours in this set satify a given property. Hence, we get a direct answer to the scenario verification problems posed in the previous section.

5.4 Matching modulo weak embedding

To match scenarios modulo weak embedding, the template has to be relaxed to permit the system to exchange additional messages that are not present in the scenario being verified. This is easily done by introducing a self loop at each state of the template automaton. These self loops are labelled with additional actions and have no timing constraint. For arbitrary weak embeddings, all possible actions are enabled at each self loop. For strict embeddings, only actions that are not mentioned in the template are enabled. We can further tune the nature of the embedding we are looking for by varying the choice of additional actions from one self loop to the next.

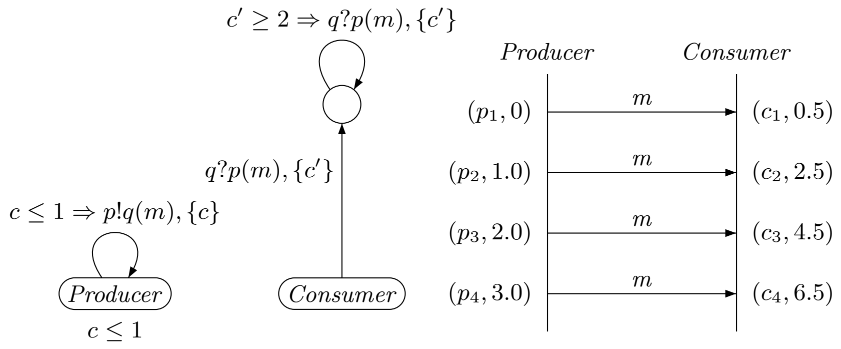

Fig. 7. The difficulty with existentially bounded channels.

6 Expressiveness issues

6.1 Bounded channels

Our implementation implicitly assumes that, in any run of the system, channels are uniformly bounded by the size of the queue that is maintained by the channel process. Recent work on timed message-passing automata [9] shows that checking whether channels are universally bounded is undecidable for in general (in particular, it is undecidable with three components and two channels).

On the other hand, we can examine the structure of the underlying untimed system, without timing constraints. It is clear that if the underlying untimed automaton has universally bounded channels, then so will the corresponding automaton with timing constraints. Message-passing automata with universally bounded channels have a well understood theory [6], so this is a natural class of systems to which we can restrict our attention.

In the untimed case, many of the characterizations and decidability results for systems with universally bounded channels also go through for systems with existentially bounded channels [5]. A collection of MSCs is said to be existentially bounded if there is a bound B such that, for every MSC in the collection, there is at least one interleaving in which no channel's capacity exceeds B. Since, with the FIFO assumption, we can reconstruct an MSC from even a single interleaving, an existentially bounded collection of MSCs has a faithful sequential represention as a regular language, which allows us to perform operations analogous to those defined for systems with universally bounded channels.

Unfortunately, the existentially bounded linearization that one requires for an MSC may be disallowed by timing constraints. Consider again the example of a producer-consumer system, this time with local timing constraints on both send and receive actions, as shown in Figure 7. The underlying untimed system is 1-bounded and the corresponding linearization is $p_1\ c_1\ p_2\ c_2\ \cdots$, where each message is consumed as soon as it is sent. However, in the timed version shown in the figure, receive events are forced to lag behind send events, so the only timed linearizations are those in which the channel is unbounded.

6.2 Imposing bounds on channels

We have assumed that all clocks used in constraints are local to a process. Notice, however, that time itself is global, so all clocks proceed at a uniform rate. This is more a simplifying assumption than a technical necessity, because no comparisons are made across clocks.

One reason to introduce global clocks would be to specify bounds on channel delays. For instance, we could associate a clock with a channel that is set when a message is sent and use a constraint on this clock's value to impose a bound on the delay before it is received. However, a naïve implementation of this idea fails, as shown in Figure 8. Here, though the clock r seems to bound the channel delay by 1, it is possible for the first message to be received after more than 1 unit (2.5 units, in the example) because the clock r is reset by the next send event. To faithfully model channel delays, we would have to associate an array of clocks with each channel, one for each position in the queue. Even with universally bounded channels, it is not clear how to use such an array of clocks when specifying a timed MPA, because the transition associated with a send action may generate messages at different depths in the queue, depending on the history. We can unfold the MPA, keeping channel information as part of the state, but this leads to a very verbose and cumbersome specification.

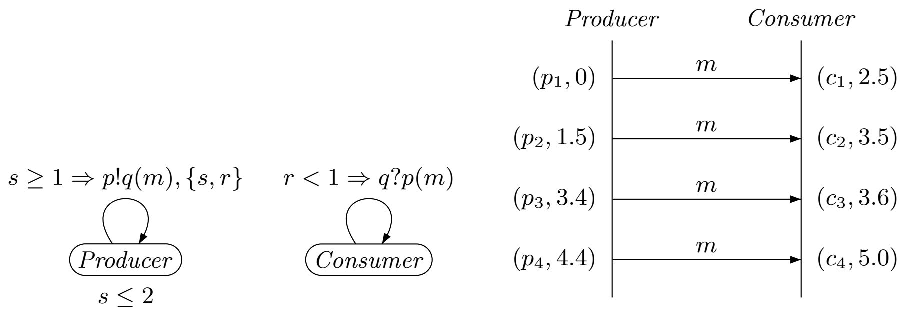

Fig. 8. Modelling channel delays

7 Optimizing the translation into Uppaal

We describe some optimizations that can be incorporated when translating the scenario matching problem into UPPAAL to enable more efficient modelchecking.

7.1 Controlling Interleavings with *committed* locations

Our implementation of buffered channels adds one new state to the UPPAAL system for each send or receive action in the original timed MPA. The introduction of an extra synchronize action with the corresponding process in the specification adds one new state to the system for each communication in the intersection of $L(\mathcal{T}) \cap L(\mathcal{A})$. These extra states introduce extra interleavings in the execution of the system. In the worst case, when all the actions in the system are communication events from the intersection, the total number of locations in the system gets tripled.

Consider the three moves $s \overset{s_{pc}=m}{\Longrightarrow} s_1 \overset{a_{pc}}{\Longrightarrow} s_2 \overset{b_{pc}}{\Longrightarrow} s'$. Here, the first transition is always enabled, and can be taken at the earliest possible instant. We would like the system to follow the specification, and hence the third transition should ideally be enabled when we reach the state s_2.

Uppaal allows locations be labelled *committed*. If any process is in a committed location, the next transition must involve an edge from one of the committed locations, and time cannot progress while in that location.

Marking the location s as *committed* forces the system to move to s_1 as soon as it enters location s. The system waits in state s_2 only if the corresponding process in the specification is not ready to synchronize. This would mean that the behaviour of the system is already deviating from the template. Since, we are only interested in computing $L(\mathcal{T}) \cap L(\mathcal{A})$, we mark location s_2 as *committed* as well. These committed locations remove the extra interleavings introduced into the system by splitting up a single message send/receive action into a sequence of actions and increases the efficiency of the model checker.

7.2 Guiding Uppaal using priorities

Uppaal also permits the allocation of priorities to channels and processes. A higher priority transitions always blocks the lower priority one. Process priorities can be used to guide the model checker, and get quicker results.

In the ATM example, the *User* process chooses from one of the many options in the menu. The *ATM* process accepts the choice made by the *User* process. In the Uppaal model, both the processes have choices regarding sending or receiving the menu options. However, in reality, the choices of the *ATM* are decided by the *User* process. We can inform Uppaal about this by declaring the *User* process to be of higher priority than the *ATM* process. Similarly, since the *Server* process needs to only service requests made by the *ATM* process, it has a lower priority. In Uppaal, these priorities are specified by the directive "`system server < atm < user ;`".

7.3 Meta Variables

The Uppaal system allows some variables to be declared as *meta variables*. Meta variables do not contribute to the state space. By declaring the global arrays within the buffers as meta variables, we can cut down on significantly on the overall set of configurations that the modelchecker has to explore.

8 Discussion

We have considered a useful extension of the scenario matching problem to timed systems. This allows us to specify and verify, more accurately, the interactions associated with typical protocol specifications.

To the best of our knowledge, ours is the first attempt to connect message-passing automata with timing constraints to timed MSCs. The problem of checking whether an MSC with interval timing constraints admits a feasible schedule has been discussed in [1], but this work studies MSCs in isolation, without reference to a system model. On the other hand, timed message-passing automata very similar those defined in this paper have been very recently considered in [9]. In this work, the emphasis is on proving (un)decidability results for simple verification criteria such as reachability and channel boundedness. This paper does not address the semantics of such automata in terms of (timed) MSCs.

Preliminary results show that our approach to solve the problem of scenario matching in timed systems can be effectively automated using a tool like Uppaal. We are working on more detailed examples to understand better the practical issues involved with timed scenario verification. Another issue to be explored is the extent to which we can enhance the expressive power of timed MSC templates.

Acknowledgments We thank Benedikt Bollig, Paul Gastin, K. Narayan Kumar, Akshay Sunderaraman and the anonymous referees for helpful comments and suggestions on the preliminary version of this paper.

References

1. R. Alur, G. Holzmann and D. Peled: An analyzer for message sequence charts. *Software Concepts and Tools*, **17(2)** (1996) 70–77.
2. G. Behrmann, A. Davida and K.G. Larsen: A Tutorial on Uppaal, *Proc. SFM 2004*, LNCS **3185**, Springer-Verlag (2004) 200–236.
3. J. Bengtsson and Wang Yi: Timed Automata: Semantics, Algorithms and Tools, *Lectures on Concurrency and Petri Nets 2003*, LNCS **3098**, Springer-Verlag (2003) 87–124.
4. D. de Souza and M. Mukund: Checking consistency of SDL+MSC specifications, *Proc. SPIN Workshop 2003*, LNCS **2648**, Springer-Verlag (2003) 151–165.
5. B. Genest, A. Muscholl and D. Kuske: A Kleene Theorem for a Class of Communicating Automata with Effective Algorithms. *Proc DLT 2004*, LNCS **3340**, Springer-Verlag (2004) 30–48.
6. J.G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni and P.S. Thiagarajan: A Theory of Regular MSC Languages. *Inf. Comp.*, **202(1)** (2005) 1–38.
7. G.J. Holzmann: The model checker SPIN, *IEEE Trans. on Software Engineering*, **23**, 5 (1997) 279–295.
8. ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU, Geneva (1999).
9. P. Krcal and Wang Yi: Communicating Timed Automata: The More Synchronous, the More Difficult to Verify, *CAV 2006*, LNCS, Springer-Verlag (2006), to appear.
10. A. Muscholl, D. Peled, and Z. Su: Deciding properties for message sequence charts. *Proc. FOSSACS'98*, LNCS **1378**, Springer-Verlag (1998) 226–242.