ADDING TIME TO SCENARIOS*

Prakash Chandrasekaran and Madhavan Mukund
Chennai Mathematical Institute, Chennai, India
{prakash,madhavan}@cmi.ac.in

Abstract Message Sequence Charts (MSCs) are used to specify the behaviour of communicating systems through scenarios. Though timing constraints are natural for describing the behaviour of real-life protocols, the basic MSC notation has no mechanism to specify such constraints. We propose a notation for specifying collections of timed scenarios and describe a framework for automatic verification of scenario-based properties for communicating finite-state machines equipped with local clocks.

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 message sequence charts (MSCs) [9].

We extend scenarios to incorporate timing constraints, yielding 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.

We propose a simple specification language based on guarded commands, along the lines of Promela [8], for generating collections of timed MSC templates. The semantics of this language is given in terms of a version of HMSCs (high-level MSCs) [7], with annotations attached to edges rather than nodes.

Our aim is to verify properties of timed systems with respect to timed MSC template specifications. Our basic system model consists of 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

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

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.

Specifications in terms of scenarios give rise to several natural verification problems. At preliminary stages of system design, scenario specifications are typically incomplete and can be classified into two categories, 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 a handshake, leading to a collision. This leads to the following verification problem: given a distributed system and a positive (or negative) scenario, does the system exhibit (or avoid) the scenario?

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

As the design of a system evolves, the interpretation of a scenario-based specification also changes. The specification is now typically seen as an exhaustive description of how the system should behave. Universality then becomes an important condition to check—does the implementation exhibit a representative behaviour consistent with each of the timed templates in the specification? Once again, the complication is that each timed template in the specification is compatible with an infinite set of timed behaviours. Moreover, we also have an infinite set of timed templates to verify.

We propose an approach to tackle these verification problems using the modelchecking tool UPPAAL, which is designed to verify properties of timed systems. This paper extends the work reported in [4], where we only consider finite sets of timed templates. Since the basic system model of UPPAAL uses synchronous handshakes, rather than message-passing, we need to encode message-passing channels by creating special processes to model buffers. Exploiting the handshake mechanism in UPPAAL, we can synchronize the system with the specification. This allows us to transform our verification questions into properties for UPPAAL to verify on the composite system.

In the untimed setting, efficient algorithms for the scenario matching problem have been identified in [11]. An approach to solve this problem using the modelchecker SPIN was proposed in [6].

Adding timing constraints to individual scenarios has been proposed in [1], where an algorithm is given to check whether such a set of timing constraints in consistent. At the level of sets of scenarios, the *live sequence chart (LSC)* formalism [5] allows adding interval constraints similar to those we consider.

One important difference is that the semantics of LSCs assumes synchronous composition of scenarios—all processes are assumed to move together from one scenario to the next. We retain the usual asynchronous semantics for MSC composition, which is more natural from the point of view of implementations.

The paper is organized as follows. In the next two sections, we formally define timed MSCs and timed message-passing automata. In Section 4, we propose a new notation for specifying timed scenarios. In the next section, we describe some verification problems for scenario based specifications. In Section 6, we describe our approach to address verification problems for scenario-based specifications using UPPAAL. 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 $\Delta_p = \{p!q(m), p?q(m) \mid p \neq q \in \mathcal{P}, m \in \mathcal{M}\}$ 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*. The set of actions that p performs is given by $\Sigma_p = \Delta_p \cup \{i_p\}$, where i_p is a local action of p. We will use local actions to describe timeouts. 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 $\downarrow e = \{e' \mid e' \leq e\}$.

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.*

(i) Each relation $\leq_{pp}$ is a linear order.

(ii) If $p \neq q$ then for each $m \in \mathcal{M}$, $|E_{p!q(m)}| = |E_{q?p(m)}|$.

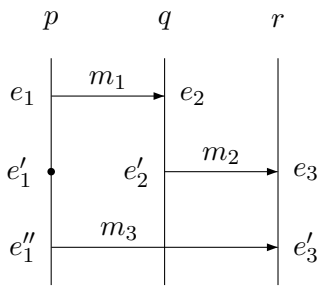

Figure 1. An MSC over $\{p, q, r\}$.

(iii) If $e <_{pq} e'$*, then* $|{\downarrow}e \cap \left(\bigcup_{m \in \mathcal{M}} E_{p!q(m)}\right)| = |{\downarrow}e' \cap \left(\bigcup_{m \in \mathcal{M}} E_{q?p(m)}\right)|$.

(iv) 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.

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 seven events $\{e_1, e'_1, e''_1, e_2, e'_2, e_3, e'_3\}$ corresponding to three messages—m_1 from p to q, m_2 from q to r and m_3 from p to r—and one local event on p, e'_1.

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

Figure 3 shows an alternative scenario in which the user does not supply the PIN within the specified time limit, leading to the ATM rejecting the card. Notice that the timeout event is modelled as a local event on the ATM process.

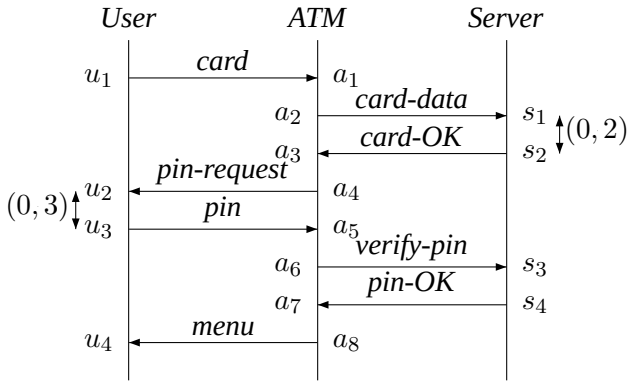

Figure 2. A timed MSC template describing interaction with an ATM.

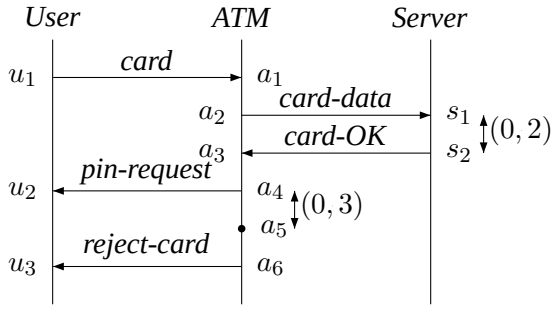

Figure 3. The user's PIN message times out.

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 $\{x \in \mathbb{R}_{\geq 0} \mid m \leq x \leq n\}$.[1]

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

The restriction on the relationship between e_1 and e_2 ensures that an interval constraint is either local to a process or describes the delay in transmitting a single message.

[1]For simplicity, we restrict ourselves to closed timed intervals in this paper. We can easily generalize our approach to include open and half-open time intervals.

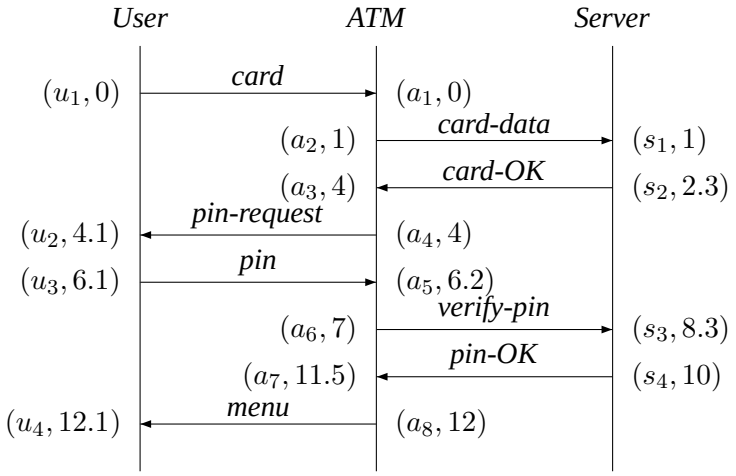

Figure 4. A timed MSC instance describing interaction with an ATM.

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* $\tau : E \to \mathbb{R}_{\geq 0}$ *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* $\tau(e_1) \leq \tau(e_2)$.

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 4 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 [7] to include local clocks on each process and time-bounds on the channels.

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 $v(c)$.*

DEFINITION 9 *A* timed message-passing automaton (timed MPA) *over Σ is a structure $\mathcal{A} = (\{\mathcal{A}_p\}_{p \in \mathcal{P}}, \Sigma, \mathcal{B})$. Each component $\mathcal{A}_p$ is of the form $(S_p, S_{in}^p, \mathcal{C}_p, \rightarrow_p, I_p)$, where:*

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

The function $\mathcal{B} : (\mathcal{P} \times \mathcal{P}) \to (\mathbb{N} \times \mathbb{N})$ associates with each channel a lower and an upper bound on the transmission time of messages on that channel.

The local transition relation $\rightarrow_p$ specifies how the process p changes state when it performs internal events or sends and receives messages.

A transition of the form (s, φ, a, X, s') says that in state s, p can perform the action a 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 specifies the clocks whose values are reset to 0 when this transition is taken. If a is of the form i_p, this transition corresponds to performing a local event on p. If $a = p!q(m)$, then this transition involves sending a message m from p to q. Symmetrically, if $a = p?q(m)$, then this transition involves p receiving a message m from to q.

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 $n \in \mathbb{N}$.

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 $\rightarrow_p$.

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, $\chi : Ch \rightarrow \mathcal{M}^*$ is the *channel state* describing the message queue in each channel c and $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ is a clock assignment, where $\mathcal{C} = \bigcup_{p\in\mathcal{P}} \mathcal{C}_p$. 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 in the usual way, together with a transition relation $\Longrightarrow \subseteq Conf_\mathcal{A} \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Conf_\mathcal{A}$. A move labelled by $d \in \mathbb{R}_{\geq 0}$ is a time elapsing move. All clocks advance by d, but the local states of processes and the channel contents remain unchanged. A move labelled by $a \in \Sigma$ is a local transition taken by one of processes. For each process p, the local state of p determines the set of moves available for p in the current configuration. If $a = i_p$, only the state of p changes and the rest of the configuration is unchanged. If $a = p!q(m)$, the message m is appended to the channel (p, q). An action of the form $p?q(m)$ is enabled only if m is currently at the head of the channel (q, p). For a more formal definition of the global transition relation, see [4].

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 $\rho : \mathrm{prf}(\sigma) \rightarrow Conf_\mathcal{A}$ where $\rho(\varepsilon)$ is assigned an initial configuration $(\overline{s}_{in}, \chi_\varepsilon, v_0)$ and for each $\sigma' \cdot (a_i, t_i) \in \mathrm{prf}(\sigma)$, $\rho(\sigma') \stackrel{d_i}{\Longrightarrow}\stackrel{a_i}{\Longrightarrow} \rho(\sigma' \cdot (a_i, t_i))$ with $t_i = t_{i-1} + d_i$ and $t_0 = 0$.

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 $L(\mathcal{A}) = \{\sigma \mid \mathcal{A} \text{ has a complete run over } \sigma\}$. $L(\mathcal{A})$ corresponds to the set of timed linearizations of a collection of timed MSCs.

Figure 5 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

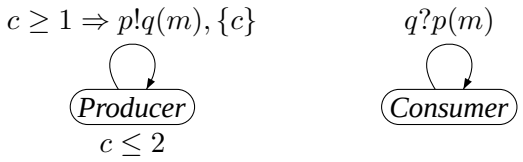

Figure 5. A timed MPA: producer-consumer

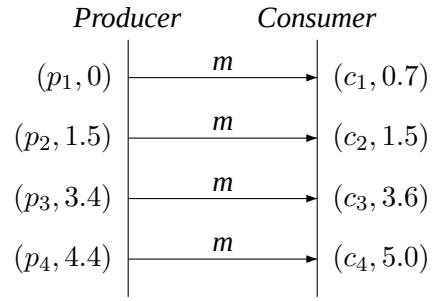

Figure 6. A timed MSC generated by the producer-consumer system.

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 has no timing constraints. Figure 6 shows a typical timed MSC generated by this timed MPA.

4. Specifying timed scenarios

The standard method to describe multiple communication scenarios is to use High-Level Message Sequence Charts (HMSCs). An HMSC is a finite directed graph with designated initial and terminal vertices. Each vertex in an HMSC is labelled by an MSC. The edges represent the natural operation of MSC concatenation. The collection of MSCs represented by an HMSC consists of all those MSCs obtained by tracing a path in the HMSC from an initial vertex to a terminal vertex, concatenating the MSCs that are encountered along the path.

In an HMSC, MSCs are concatenated asynchronously. This corresponds to gluing together the process lines of consecutive MSCs. The implication is that the boundaries between the individual MSCs along a path disappear, as a result of which some processes could move ahead of others. If the asynchrony between processes is bounded, all channels remain universally bounded and the specification is globally finite-state. Unfortunately, it is undecidable in gen-

eral whether an HMSC specification satisfies this property, though sufficient structural conditions are known [7].

We propose a guarded command language inspired by Promela [8] to describe families of timed scenarios generated from basic timed templates. The basic building blocks of the language are finite timed MSC templates, as defined in Section 2.2. Statements are combined using sequential composition (`;`), nondeterministic guarded choice (`if...fi`) and nondeterministic guarded looping (`do...od`). We allow statements to be labelled, and permit labelled breaks from within loops as well as explicit gotos.

Rather than providing a precise grammar describing the syntax, we explain the notation through an example. Continuing with our ATM example, suppose the ATM is programmed to ask for the user's PIN after he has inserted the card. If the user does not enter his PIN within a specified time limit, the ATM repeats the request. At some point, nondeterministically, the ATM can also decide to reject the card. Once the user does respond, there is a possibility that the PIN is wrong. If so, the ATM swallows the card. If the PIN is correct, the user may ask for his balance or may try to make a withdrawal. These scenarios can be combined in our notation as follows, where some of the basic timed templates used in the specification are shown in Figure 7.

```
L0::  Initiate;
L1::  do
      [] NoPin
      [] NoPin; RejectCard; goto L0
      [] SwallowCard; goto L0
      [] OKPin ; break L1
      od;
      if
      [] BalanceEnquiry; goto L0
      [] WithdrawCash; if
                       [] InsufficientFunds; goto L0
                       [] DispenseCash; goto L0
                       fi
      fi
```

It is not difficult to see that our textual notation can be translated into a graphical HMSC-like notation, provided we annotate edges in the HMSC, rather than nodes, by basic timed MSC templates. Figure 8 shows the HMSC corresponding to the current example.

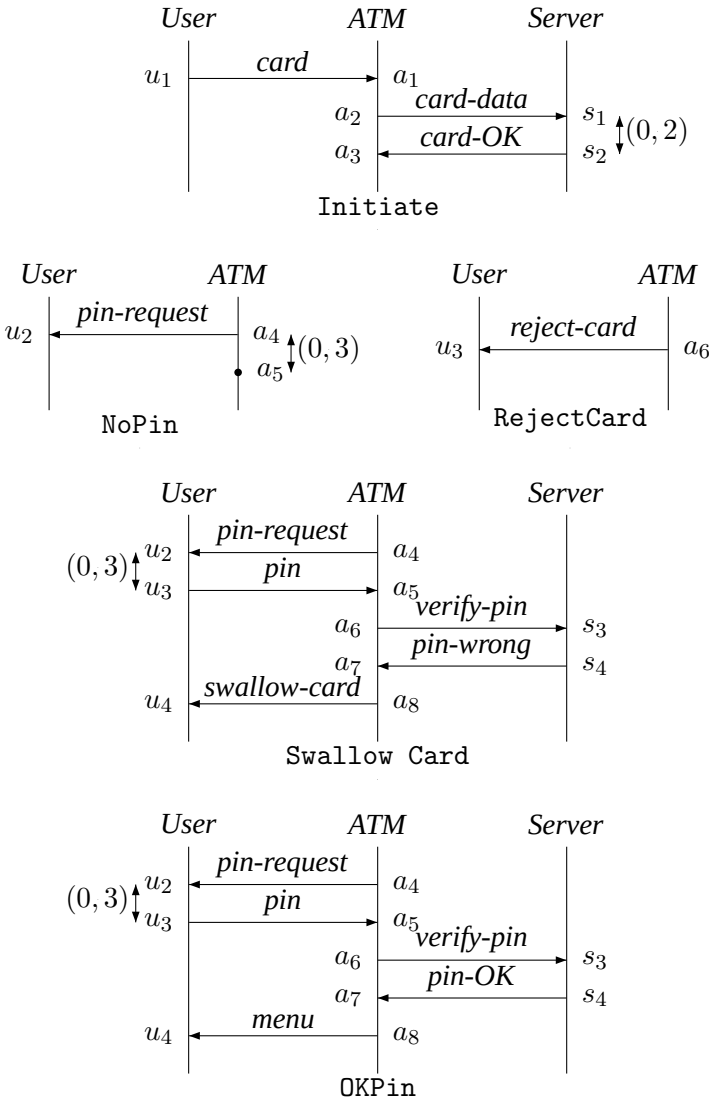

Figure 7. Some basic timed MSC templates used in the sample specification

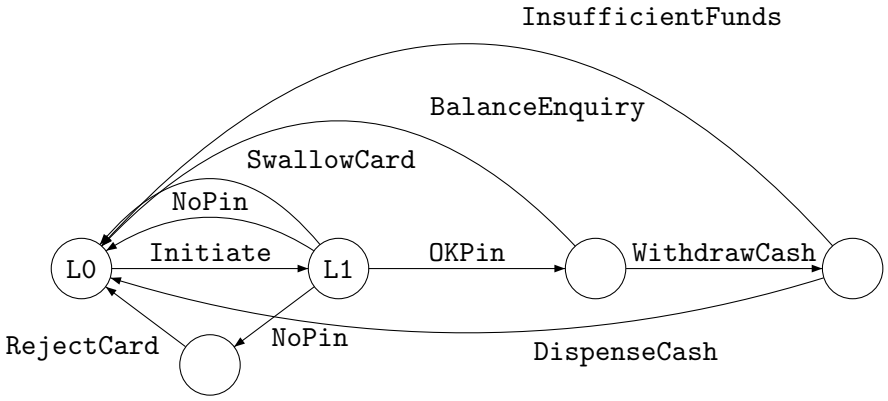

Figure 8. HMSC corresponding to the sample specification

5. Verification questions for timed scenarios

In the setting of timed MSC templates and timed MPAs, there are multiple verification questions that one can address. We focus on two of them here.

5.1 Scenario matching

Given a timed MSC template $\mathcal{T}$ and a timed MPA $\mathcal{A}$, we ask 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. This question is natural in the early stages of a specification, when scenarios are not expected to exhaustively describe the system's behaviour.

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.

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.

5.2 Universality

As the specification evolves, it is expected that it more exactly describes the desired behaviour. In an untimed setting, it would be natural at this stage to demand that the behaviour of the implementation match the specification upto, say, language equivalence. However, in a timed setting, we may have a specification with generous time constraints to be compared with an implementation

that is more restrictive. Hence, the natural analogue of language equivalence is to ask whether for every timed MSC template in the specification, there is at least one timed behaviour in the implementation that is consistent with the template. We refer to this problem as *universality*.

6. Using UPPAAL for scenario verification

In [4], we present a framework for verifying properties of timed scenarios using UPPAAL, a modelchecker for timed systems [2]. The framework in [4] is designed to deal with finite sets of timed MSC templates, which can essentially be handled one at a time. Here, we extend this framework to tackle with specifications that encompass a possibly infinite set of scenarios.

UPPAAL supports the analysis of networks of timed automata for timing properties. Unfortunately, UPPAAL does not have a direct way of modelling asynchronous communication. We can simulate asynchronous communication by creating explicit buffer processes. Moreover, we can exploit the synchronous communication paradigm built-in to UPPAAL to synchronize the system with the specification at each communication action. This allows the system to evolve only along trajectories that are consistent with the specification, thus automatically restricting the behaviours of the composite system to those that are of interest.

6.1 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 moves the free slot pointer 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}.

6.2 Modelling channel delays

In an MPA, clocks are local and must be associated with a fixed process. However, UPPAAL permits global clocks. To faithfully model channel delays, we associate an array of clocks with each channel, one for each position in the queue. With universally bounded channels, we can always assign a fresh clock from this array to each new message sent on a channel that is initialized when the message is sent. The receive action for this message is guarded by clock constraints corresponding to the time bounds associated with the channel.

6.3 Modelling timed MSC specifications in UPPAAL

To verify a timed MSC specification, the first step is to convert the specification into a timed MPA, preserving the language of timed MSCs of the specification.

For a single timed MSC template, the communication structure of the MPA is fixed and can be computed easily, using the FIFO property of channels. We introduce a new local clock for each local timing constraint and add clock constraints using these clocks to guard the actions of the MPA so that it respects the timed template.

Since we can interpret a general timed MSC specification as an HMSC in which edges are labelled by basic timed MSC templates, we construct an MPA for each basic timed MSC template and connect these up using internal actions and dummy states to reflect the overall structure of the corresponding HMSC.

The usual difficulty with this construction is to ensure that all processes follow consistent paths in the HMSC. For this, we add a monitor process that tracks the path followed by each process in the system. We have to ensure that the information maintained by the monitor process is bounded. As we have observed earlier, with asynchronous concatenation, some processes may be arbitrarily far ahead of others and the overall behaviour may be non-regular. We can ensure regularity by imposing structural restrictions on the HMSC [7]. Instead, we impose a bound on the number of live instances of each basic timed MSC template in the system. This allows us to perform a form of bounded model-checking for arbitrary timed template specifications.

6.4 Scenario matching

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 $s \stackrel{p!q(m)}{\Longrightarrow} s'$ in the original timed MPA now breaks up, in the UPPAAL implementation, into a sequence of three moves $s \stackrel{s_{pc}=m}{\Longrightarrow} s_1 \stackrel{a_{pc}}{\Longrightarrow} s_2 \stackrel{b_{pc}}{\Longrightarrow} s'$. 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 $s \stackrel{p?q(m)}{\Longrightarrow} s'$, the UPPAAL implementation executes a sequence of the form $s \stackrel{r_{pc}=m}{\Longrightarrow} s_1 \stackrel{\bar{a}_{cp}}{\Longrightarrow} s_2 \stackrel{\bar{b}_{cp}}{\Longrightarrow} s'$, where, by convention, 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 satisfy a given property. This answers the scenario verification problems posed in the previous section.

6.5 Universality

Recall that universality is the property that the implementation exhibits at least one timed behaviour consistent with each timed template generated by the specification. In general, we do not know how to solve this problem. Instead, we address a weaker version that we call *coverage*.

We assume that the user provides a (finite) set of paths through the specification that he would like to see exhibited in the implementation. In particular, we can always ensure that we cover all the edges in the HMSC through such a collection of paths. In UPPAAL, we can verify reachability properties written in CTL. This includes formulas that assert that there exists a path along which a sequence of state properties holds. By adding state labels to the UPPAAL implementation, we can mark when a basic timed MSC template is executed by the composite system obtained by synchronizing the specification with the implementation. Each path to be covered can then be described using an appropriate CTL formula of the form permitted by UPPAAL. The overall problem then reduces to verifying a finite conjunction of such CTL formulas.

7. Discussion

Adding time to specifications of distributed systems appears to be a problem of both practical and theoretical interest.

Augmenting scenarios with timing constraints allows us to specify and verify, more accurately, the interactions associated with typical protocol specifications. Timing constraints give rise to new variants of verification questions, some of which we do not know how to tackle, such as universality.

Global time indirectly synchronizes processes, leading to undecidability—for instance, boundedness of channels is undecidable even if we have only local clocks [10]. It would be interesting to explore whether it is possible to relax the correlation the time across components without completely decoupling all clocks and yet obtain some positive results.

Another interesting theoretical question is to explore the relationship between automata, logic and languages in a setting that incorporates both distribution and time. A first step in this direction is the work reported in [12].

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] P. Chandrasekaran and M. Mukund: Matching Scenarios with Timing Constraints *Proc. FORMATS 2006*, Springer LNCS 4202 (2006) 98–112.

[5] W. Damm and D. Harel: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* **19(1)** (2001) 45–80.

[6] D. de Souza and M. Mukund: Checking consistency of SDL+MSC specifications, *Proc. SPIN Workshop 2003*, LNCS **2648**, Springer-Verlag (2003) 151–165.

[7] 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.

[8] G.J. Holzmann: The model checker SPIN, *IEEE Trans. on Software Engineering*, **23**, 5 (1997) 279–295.

[9] ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU, Geneva (1999).

[10] P. Krcal and Wang Yi: Communicating Timed Automata: The More Synchronous, the More Difficult to Verify, *CAV 2006*, LNCS, Springer-Verlag (2006), to appear.

[11] A. Muscholl, D. Peled, and Z. Su: Deciding properties for message sequence charts. *Proc. FOSSACS'98*, LNCS **1378**, Springer-Verlag (1998) 226–242.

[12] Akshay Sunderaraman, *Formal Specification and Verification of Timed Communicating Systems*, Master's thesis, LSV, ENS Cachan (2006). Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/Akshay-M2.pdf`