Specifying Interacting Components with Coordinated Concurrent Scenarios

Prakash Chandrasekaran and Madhavan Mukund

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

Abstract. We introduce a visual notation for local specification of concurrent components based on message sequence charts (MSCs). Each component is a finite-state machine whose actions are MSCs that specify its local view of the overall communication in the system. These local MSCs are composed into coherent global scenarios using a separately specified set of transactions.
Intuitively, each MSC represents a phase of interaction. We introduce a mechanism to overlap phases that allows complex interactions to be specified without obscuring the logical structure of the constituent scenarios.
Our notation combines the global view available in models such as high-level message sequence charts (HMSCs) with the local, asynchronous structure captured by message-passing automata (MPA). In fact, both HMSCs and MPAs can be captured as special cases of our formalism.
In this paper we focus on the syntax and formal semantics of our notation, with examples that illustrate why this approach is more natural for capturing real-life specifications. We also describe an approach to use automated tools to analyze systems specified using our notation.

1 Introduction

Specifying concurrent components formally is still a challenge. Operational models such as message-passing automata are difficult to work with because they lack a mechanism to specify the overall behaviour of the components. The other approach is to use visual notations based on Message Sequence Charts (MSCs) [9]. These are very helpful for describing the global interactions in the system but can only express limited aspects of control flow and are hence difficult to execute and implement.

In this paper, we present a specification language that we call *coordinated concurrent scenarios* that bridges the gap between these two extremes. We describe components locally, but in terms of MSCs. We then provide a separate specification of how these local MSCs combine globally to form *transactions*. Our notation incorporates two useful features not found in existing formal specifications. The first is the ability to describe multiple activities interleaving asynchronously on a single process—for example, a server that interacts with multiple clients in parallel. The second is a mechanism to deal with blocking, as happens when a service is waiting for a resource to become available.

Our formalism is most closely related to netcharts [10], an MSC-based notation that, in turn, is an asynchronous version of the communicating transaction processes model [11]. The main difference between our notation and netcharts is that our specifications are completely local, whereas netchart specifications are global.

Our notation is quite expressive and can simulate both message sequence graphs (MSGs) and MPAs. It follows that verification problems for our notation are intractable, in general. However, we can impose overall resource bounds and translate our specifications into models that can be analyzed by automated verification tools—Uppaal, to be specific.

Before plunging into technical details, we motivate our approach by examining different approaches to modelling the alternating bit protocol. After this extended example, we introduce the basic syntax and semantics of coordinated scenarios in Section 3. In the next two sections, we describe how to extend our notation with facilities for describing asynchronous execution and blocking. We present the complete formal semantics of our model in Section 6. In Section 7 we compare our notation with MSGs, MPAs and netcharts. We address the verification problem in Section 8 and conclude with a brief discussion.

2 Motivation

Before proceeding to formal definitions, we motivate our approach in the context of a classical example, the alternating bit protocol.

The alternating bit protocol (abp) is a mechanism to transfer data from one process, the *sender*, to another, *the receiver*, over a channel that may corrupt data. The protocol assumes that the receiver can detect whether a data packet is corrupted—say using a checksum. Other than the problem of data corruption the channel is reliable—messages are never lost and are received in fifo order.

In this protocol, the sender sends data packets tagged alternately with 0 and 1. Whenever a packet tagged i is received intact, the receiver sends an acknowledgment a_i. The sender retransmits the packet tagged i till a_i is received. On receiving a_i, the sender sends the next packet tagged with $1-i$ and retransmits till an acknowledgment a_{1-i} is received. On receiving a_{1-i}, the sender switches back to using tag i for the next data packet, and so on. It is possible that the sender receives "old" acknowledgments a_{1-i} during the phase when it is transmitting packets tagged with i. These "negative acknowledgments" are ignored.

An analysis of the protocol suggests that there are three types of interactions involved.

- The sender sends a packet that is deemed to be corrupted by the receiver.
- A packet tagged 0 is received intact, generating an acknowledgment a_0.
- A packet tagged 1 is received intact, generating an acknowledgment a_1.

Scenario based specifications attempt to describe the overall behaviour of a set of communicating processes in terms of such basic interactions. A run of the

system is represented by a picture called a *message sequence chart* (MSC) indicating the flow of messages between the components of the system. For instance, the three basic interactions in the alternating bit protocol can be represented as MSCs as shown in Figure 1.

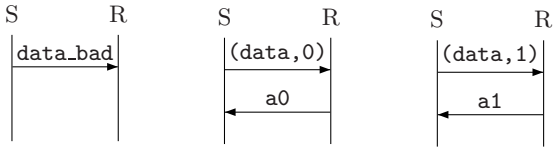

Fig. 1. Basic interactions in the alternating bit protocol

To describe all communication patterns consistent with a protocol specification, we need a mechanism that can generate an infinite family of scenarios. A standard way to do this is to use message sequence graphs (MSGs). An MSG is a finite-state automaton whose alphabet is a set of basic MSCs. A run of the automaton generates an MSC by concatenating the MSCs along the edges traversed by the run.

Every MSC generated by an MSG is *finitely generated*—it can be decomposed into a sequence of basic MSCs drawn from the set used to annotate the transitions of the MSG. Protocols such as the abp permit complex interactions whose MSC representation cannot be sliced in terms of a finite number of basic interactions. Hence, such protocols cannot be accurately modelled using MSGs.

One proposal to overcome this limitation of MSGs is to permit edges to be labelled by "incomplete" MSCs with dangling edges, corresponding to sends without matching receives or vice versa. In this model, called *compositional MSGs* [6], messages sent in one MSC can be received in a later MSC. A run is valid if it generates, overall, an MSC with no dangling edges. The main drawback of this relaxation is that the resulting specification no longer reflects the basic interactions that constitute the protocol.

In fact, one could argue that a compositional MSG is little more than a *message-passing automaton* (MPA) or *communicating finite-state machine* (CFSM). An MPA is a collection of finite-state automata in which each local action corresponds to sending a message on a channel, receiving a message on a channel, or performing a local action. While MPAs are a natural operational model for communicating systems, they are not very readable as specifications. Analyzing the behaviour represented by an MPA is difficult because of the complex manner in which send and receive actions can combine across components. In effect, basic interactions are decoupled, resulting in a loss of structural information about the system.

Figure 2 shows an MPA corresponding to the abp. Notice how the three basic interactions from Figure 1 have been broken up and scattered across the local actions of the two components. If we construct the global state space of this MPA, the resulting transition system can be interpreted as a compositional MSG that suffers from the same lack of transparency as the MPA.

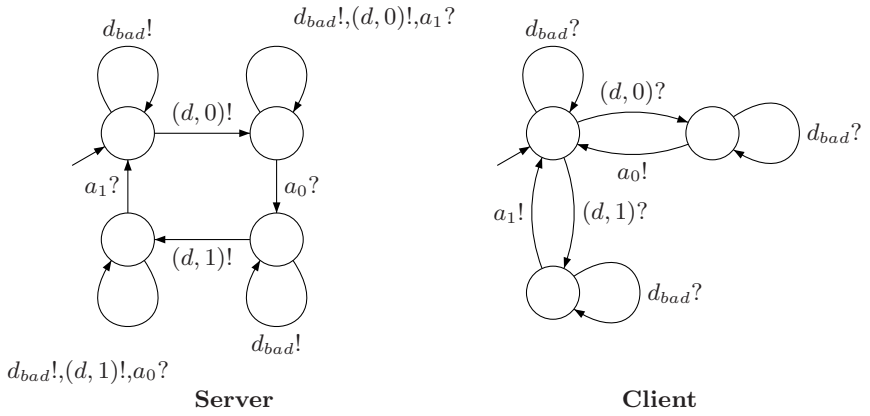

Fig. 2. Message-passing automaton for the alternating bit protocol

Another proposal to get around the limitations of MSGs without resorting to incomplete MSCs is to allow actions on a process to commute. In a *causal MSG*, we have an independence relation on the set of actions of each component, in the style of Mazurkiewicz trace theory [4, 5]. Adjacent independent actions can be reordered, which allows basic interactions to interleave with each other, unlike in standard MSGs. Though causal MSGs have increased expressiveness, postulating independence at the level of local events again forces us to break up basic interactions into smaller units, making the specification more difficult to analyze. For instance, in the causal MSG specification of the abp, the three basic interactions of Figure 1 have to be decomposed into five "unit" interactions as shown in Figure 3.

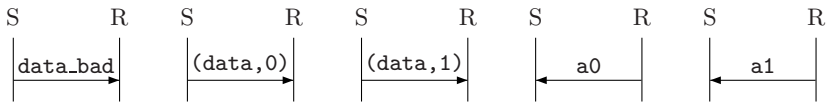

Fig. 3. Basic interactions in the causal MSG for the alternating bit protocol

In our model, we describe each component p as an MSG whose edges are p-local MSCs that only describe interactions involving p. We separately specify how (sets of) local MSCs across components combine to form global MSCs, or *transactions*. Thus, we have local specifications with a coordination mechanism. To this, we add the ability to specify when scenarios local to a process can be interleaved with each other.

Figure 4 illustrates our notation for the alternating bit protocol. The client and server are specified separately, as in an MPA. Transitions are labelled by *local MSCs* that are restricted to messages where the component is directly involved. The MSCs used in the specification correspond to the three basic interactions described in Figure 1. In this case, since there are only two components, all these MSCs are already local MSCs for both the client and the server.

The label *async* attached to M_0 and M_1 in the server indicate that these MSCs can run in parallel. Thus, the acknowledgment $a0$ from an earlier instance of M_0 can arrive after the data message $(data, 0)$ of a later M_0 has been sent.

Here, b_0 and b_1 are local variables of the server that are set in M_0 and M_1. The server's transitions between s_0 and s_1 have guards pertaining to these variables.

We independently describe how the local MSCs in the two components may be combined into global transactions. Here this is particularly simple—MSCs with the same name in the two components form transactions. Thus M_0 in the server overlaps with M_0 in the client, and so on.

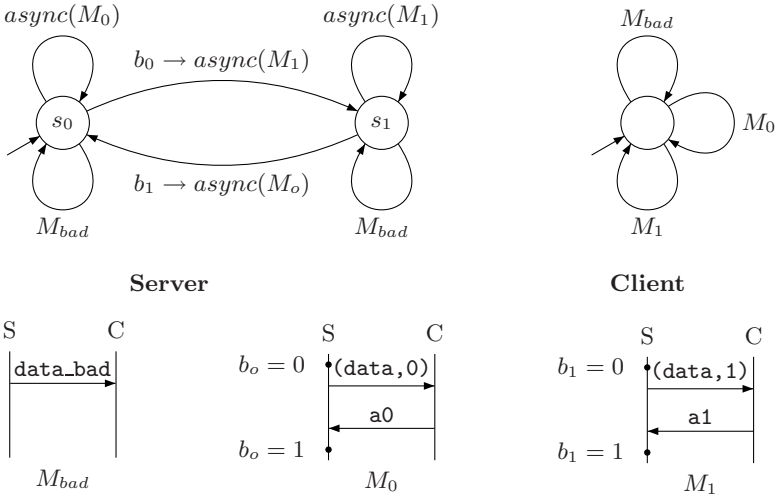

Fig. 4. The alternating bit protocol specified in our notation

3 Coordinated Scenarios

We begin with a vanilla version of our notation, to highlight the key aspects of the syntax and semantics. Later we add features to increase the expressiveness.

3.1 Message sequence charts

We begin with a quick introduction to message sequence charts (MSCs). An MSC visually describes a set of messages exchanged by components in a system.

Formally, let $\mathcal{P} = \{p, q, r, \ldots\}$ be a finite set of processes that communicate via messages sent over reliable FIFO point-to-point channels. We assume a finite set of message types $\mathcal{M}$. Each $p \in \mathcal{P}$ can perform three types of actions.

- $p!q(m)$: p sends message m to $q \in \mathcal{P}$.

– $p?q(m)$: p receives message m from $q \in \mathcal{P}$.
– Local actions, denoted $\{a, b, \ldots\}$.

Let Σ_p denote the set of actions performed by p and $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$. By Δ_p we denote the set of actions $\Sigma_p \cup \{q!p(m), q?p(m) \mid m \in \mathcal{M}, q \in \mathcal{P}, q \neq p\}$. Thus, Δ_p extends Σ_p with send and receive actions on other processes that refer to messages sent to or received from p.

Labelled posets A Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a partially ordered set with a labelling function $\lambda : E \to \Sigma$. For $e \in E$, $\downarrow e = \{e' \mid e' \leq e\}$ and for $X \subseteq E$, $\downarrow X = \cup_{e\in X} \downarrow e$. We call $X \subseteq E$ a *prefix* of M if $X = \downarrow X$.

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 $p, q \in \mathcal{P}$, $p \neq q$, we define the relation $<_{pq}$ as follows:

$$e <_{pq} e' \stackrel{\text{def}}{=} \exists m \in \mathcal{M} \text{ such that } \lambda(e) = p!q(m),\ \lambda(e') = q?p(m) \text{ and } |\downarrow e \cap E_{p!q(m)}| = |\downarrow e' \cap E_{q?p(m)}|$$

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 (MSC). *An MSC over $\mathcal{P}$ is a* finite *Σ-labelled poset $M = (E, \leq, \lambda)$ where:*

1. *Each relation $\leq_{pp}$ is a linear (total) 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 $\bigcup_{p,q\in\mathcal{P}} <_{pq}$.*

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.

We can sequentially compose MSCs M_1 and M_2 by inserting all events in M_2 after the events in M_1. More formally, let $M_1 = (E^1, \leq^1, \lambda^1)$ and $M_2 = (E^2, \leq^2, \lambda^2)$ be a pair of MSCs such that E^1 and E^2 are disjoint. The *(asynchronous) concatenation* of M_1 and M_2 yields the MSC $M_1 \circ M_2 = (E, \leq, \lambda)$ where $E = E^1 \cup E^2$, $\lambda(e) = \lambda^i(e)$ if $e \in E^i$, $i \in \{1, 2\}$, and $\leq = (\leq^1 \cup \leq^2 \cup \bigcup_{p\in\mathcal{P}} E^1_p \times E^2_p)^*$.

3.2 Local MSCs and MSGs

For $p \in \mathcal{P}$, a *p-local MSC* is an MSC in which every event has a label from Δ_p. Thus, every message sent or received in a p-local MSC involves process p. Given

an arbitrary MSC M, the *p-projection of* M, $M{\downarrow_p}$, is obtained by erasing all events whose label is not in Δ_p. Note that $M{\downarrow_p}$ is always a p-local MSC.

Let $\mathbb{M}_p$ be a collection of p-local MSCs. A *p-local message sequence graph (MSG)* over $\mathbb{M}_p$ is a finite-state automaton $\mathcal{A}_p = (S, S_{in}, \rightarrow, F)$ where S is the set of states with subsets of initial and final states denoted by S_{in} and F, respectively, and $\rightarrow \subseteq Q \times \mathbb{M}_p \times Q$ is the transition relation.

3.3 Transactions

Let $\mathcal{A}_p = (S, S_{in}, \rightarrow, F)$ be a p-local MSG over $\mathbb{M}_p$. A *p-local transaction* is a (possibly empty) sequence $M_1 \circ M_2 \circ \cdots \circ M_k$ of p-local MSCs from $\mathbb{M}_p$. Let $\{\mathcal{A}_p\}_{p\in\mathcal{P}}$ be a collection of p-local MSGs. A *(global) transaction* is a tuple $T = \{T_p\}_{p\in\mathcal{P}}$ of p-local transactions T_p such that there exists an MSC M with $M{\downarrow_p} = T_p$ for each $p \in \mathcal{P}$. The situation $M{\downarrow_p} = \emptyset$ corresponds to a transaction where p does not participate, in which case the p-local transaction T_p corresponds to the empty sequence of p-local MSCs.

3.4 Semantics

Let $\{\mathcal{A}_p = (S_p, S_{in}^p, \rightarrow_p, F_p)\}_{p\in\mathcal{P}}$ be a collection of p-local MSGs and let $\mathcal{T}$ be a set of global transactions. The pair $(\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \mathcal{T})$ defines an executable specification whose informal semantics is as follows.

Each component $\mathcal{A}_p$ begins in an initial state. Whenever a component p makes a transition of the form $s \xrightarrow{M} s'$, it either

- initiates a fresh instance of a global transaction $T = \{T_p\}_{p\in\mathcal{P}}$ where M is the first p-local MSC in T_p, or
- continues an existing instance of a global transaction $T = \{T_p\}_{p\in\mathcal{P}}$ where M forms part of T_p.

As usual, within an MSC, a send event is always enabled and a receive event is enabled if the corresponding send event has already occurred.

When all components of an active global transaction have been completed, the transaction has been discharged and is removed from the list of pending transactions. A *run* of the system is one in which each component begins in an initial state and ends in a final state so that all transactions have been discharged.

An important feature is that components are not required to execute transactions in the same global sequence. This permits transactions to interleave with each other. For example, consider the specification in Figure 5. In the figure, $M_i = M_i^p = M_i^q$ for $i \in \{1, 2\}$. The set of transactions is $\mathcal{T} = \{M_1 = (M_1^p, M_1^q), M_2 = (M_2^p, M_2^q)\}$. A run of the system consists of k_1 copies of M_1 and k_2 copies of M_2. However, p initially executes k_1 copies of M_1^p while q begins by executing k_2 copies of M_2^p, resulting in an MSC $M_\times^{k_1 k_2}$ in which the messages from the two transactions cross each other.

To formally describe the semantics, we define a *configuration* to be a triple $(\{s_p\}, \tau, \alpha)$ where

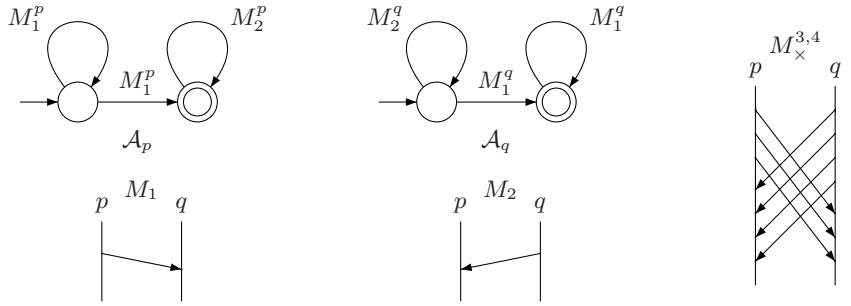

Fig. 5. Crossing Transactions

- $s_p \in S_p$ is the current local state of each $p \in \mathcal{P}$.
- τ is a *transaction table*. Each entry in τ consists of a (uniquely) labelled copy (t, ℓ_t) of a transaction from $\mathcal{T}$, together with a pair (M_t, φ_t) where M_t is the (global) MSC corresponding to t and φ_t is a "colouring" function used to mark events in M_t that have already occurred.
- $\alpha : \mathcal{P} \to \tau$ is a partial function indicating which transaction each process is currently executing.

Initially, the system is in a configuration $(\{s_p^0\}_{p\in\mathcal{P}}, \tau_0, \alpha_0)$ where $s_p^0 \in S_{in}^p$ is an initial state for each p, τ_0 is the empty table and $\alpha_0(p)$ is undefined for each p.

A component $\mathcal{A}_p$ can make two types of moves.

- If $\mathcal{A}_p$ is currently in the middle of an MSC M that forms part of an active transaction, it can execute the minimum enabled p-event $e \in M$. The configuration is then updated as follows. Let $\alpha(p) = (t, \ell_t, M_t, \varphi_t) \in \tau$. The colour $\varphi_t(e)$ of $e \in M_t$ is updated to indicate that it has occurred. If there are no more pending events in M_t, the entry $(t, \ell_t, M_t, \varphi_t)$ is removed from τ and $\alpha(p)$ is set to undefined for all p such that $\alpha(p) = (t, \ell_t, M_t, \varphi_t)$.
- Otherwise, $\mathcal{A}_p$ makes a transition $s \xrightarrow{M} s'$. In this case, the local state of $\mathcal{A}_p$ in the configuration changes from s to s'. At this point, $\mathcal{A}_p$ is allowed to initiate a new transaction, continue the current transaction $\alpha(p)$ or switch to a different transaction already present in τ.
  In the first case, a new entry $(t', \ell_{t'}, M_{t'}, \varphi_{t'})$ is added to τ with $\varphi_{t'}(e)$ initialized to show that no events in $M_{t'}$ have occurred. Also, $\alpha(p)$ is set to $(t', \ell_{t'}, M_{t'}, \varphi_{t'})$. In the second case, there is no change in $\alpha(p)$. In the third case, $\alpha(p)$ is made to point to a different transaction in τ.

A *final configuration* is a configuration of the form $(\{s_p^f\}_{p\in\mathcal{P}}, \tau_0, \alpha_0)$ where $s_p^f \in F_p$ is a final state for each p, and, as before, τ_0 is the empty table and $\alpha_0(p)$ is undefined for each p.

A *run* ρ is a sequence of moves that starts in an initial configuration and ends in a final configuration. Each run defines an MSC M_ρ that is built up incrementally along the run, as described above.

Notice that the semantics permits a process to switch transactions at MSC boundaries without completing the currently active transaction. Figure 6 indicates why this is useful. In this primitive client-server system, there are three basic MSCs, corresponding to the client logging in, servicing of a data request and the client logging out. It is useful to make $M_{login} \circ M_{logout}$ a single transaction to indicate that they form a logical unit. However, this transaction must be interrupted by M_{data} for the system to achieve anything useful. Later, we will revisit this example in a more realistic setting that further clarifies the need for splitting transactions in this manner.

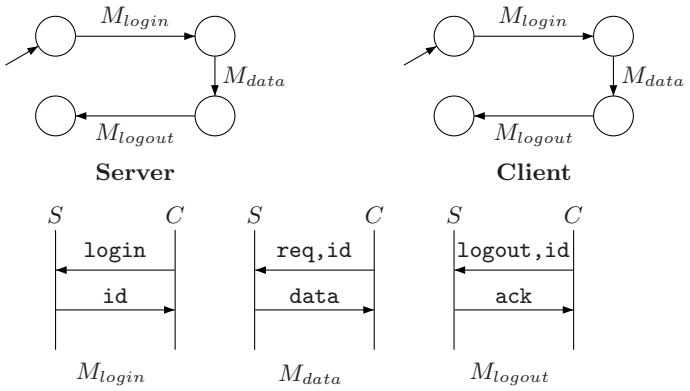

Fig. 6. A simple client-server

4 Adding Local Concurrency

In the client-server example shown in Figure 6, a more natural specification would allow the server to explicitly interleave $M_{login} \circ M_{logout}$ with M_{data}. To permit this, we permit an MSC M labelling a transition in a p-local MSGs to be tagged *async*, indicating that M is spawned as a separate asynchronous thread. For this we augment the transition function $\rightarrow$ of $\mathcal{A}_p$ with a labelling function $async_p$.

$\rightarrow_p : S_p \longrightarrow S_p \times \mathbb{M}_p \times \{async, sync\}$

When no label is specified in the visual notation, we assume the default behaviour to be "synchronous". For example, in figure 7 the server executed M_{data} synchronously, and M_{auth} asynchronously. Thus, every component $\mathcal{A}_p$ executes a sequential "synchronous" thread in parallel with some "asynchronous" threads spawned when traversing edges labelled $async(M)$. The control flow is always dictated by the synchronous thread—asynchronously spawned MSCs execute to completion in a stateless manner. The most general possibility would be to permit all secondary parallel threads to be interleaved in an arbitrary manner with the main synchronous thread. However, we restrict asynchronous threads to execute only at MSC boundaries of the synchronous computation.

This captures the intuition that each MSC in the main computation constitutes an "indivisible" phase of the overall communication patter, and hence should not be interrupted by external events. On the other hand, there is no such restriction between asychronous threads—if a process has spawned more than one asynchronous thread, these can interleave with each other in an arbitrary manner.

Before presenting the formal semantics, we illustrate this extension of our syntax through an example. Consider the enhanced client-server system in Figure 7. The client is as before and executes in three phases. The server now has a single transition with an MSC incorporating both login and logout. This MSC is marked *async* so it can interleave with the MSC corresponding to data transfer. However, since interleaving is restricted to occur at MSC boundaries, the logout message cannot interrupt M_{data}.

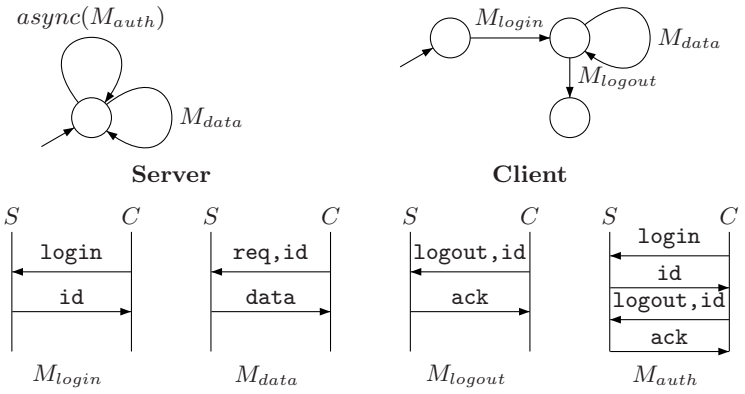

Fig. 7. A more realistic client-server

We have to extend the definition of a configuration to account for the fact that a process can actively participate in more than one transaction in the transaction table, thanks to spawning asynchronous threads. Notice that the main synchronous computation is always located in a unique transaction. We continue to use $\alpha(p)$ to denote the active synchronous transaction for process p and we add a map $\beta : \mathcal{P} \to 2^{\tau}$ to capture the set of asynchronous transactions currently active for each process. Thus, a configuration is now a tuple of the form $(\{s_p\}_{p\in\mathcal{P}}, \tau, \alpha, \beta)$ where each entry in the transaction table τ is of the form $(t, \ell_t, M_t, \varphi_t)$, as before. The rules for updating configurations when an event occurs are similar to the earlier case. The complete semantics is described in Section 6. We allow events of asynchronous transactions to occur only when the synchronous component is at an MSC boundary. As before, a run ρ from an initial configuration to a final configuration describes an MSC M_ρ.

4.1 Local variables and guards

In Figure 7, we expect that the server will honour a data transfer request only if the client is currently logged in. To specify such requirements, we equip processes with local variables that are assigned values through local actions. We then permit transitions to be annotated by boolean guards that test the values assigned to these local variables—as usual, a transition is enabled only if the accompanying guard is true. Thus, the server can have a boolean variable `authenticated` that is set to true when it receive a login message and reset to false when it receives a logout message. The transition labelled M_{data} is then equipped with a guard that checks if `authenticated` is true.

The alternating bit protocol example that we saw earlier in Figure 4 makes use of *async* transitions and guards. The server has two local variables b_0 and b_1 that control the bit that is to be sent with the current data packet. When b_i is set, the sender uses bit i. Receiving an acknowledgment a_i resets b_i to false and sets b_{1-1} to true. These variables are then used to guard the transitions that switch the state of the sender. Since channels are fifo, negative acknowledgments do not create any spurious resets of these local variables—all negative acknowledgments a_{1-i} must arrive before the first positive acknowledgment a_i.

5 Specifying Blocked Threads

A common situation that arises when specifying the behaviour of concurrent components is to have one asynchronous thread wait for a resource that is currently in use by another component. In such a situation, the first thread blocks until the resource becomes available. This kind of behaviour cannot be captured naturally in conventional MSCs (or MPAs). To overcome this limitation, we enhance our model to permit edges in p-local MSGs to be annotated by simple *MSC programs* rather than just p-local MSCs.

5.1 p-local MSC programs

For $p \in \mathcal{P}$, p-local MSC programs are built up from simple p-local MSCs using three new constructs, *atomic*, *if* and *await*. Let M represent a basic p-local MSC and C a boolean assertion about the values of p's local variables. The set of p-local MSC programs is then given by the following grammar

$$P ::= M \mid M \circ P \mid \mathit{atomic}\ P \mid \mathit{if}\ C\ P \mid \mathit{await}\ C\ \mathit{while}\ C\ \mathit{blocked}\ M\ \mathit{do}\ P$$

Informally, *atomic* P asserts that P is to be executed atomically—that is, p cannot interleave any other events till P is completed. As expected, *if* C P is interpreted as "perform P if condition C holds". Finally, *await* C_1 *while* C_2 *blocked* M *do* P means that the process waits for C_1 to be become true. If it is blocked because C_1 is false, the MSC M is executed *once*. The block is removed either by C_1 becoming true or C_2 becoming false. The first case is the normal

one and the process goes on to execute P. If C_2 becomes true before C_1, the *await* is aborted without executing P.

Once again, to illustrate this extended notation, we work through an example. Figure 8 shows a print server that connects a client to a printer. The client behaviour is simple—it logs in to the server, sends a sequence of print requests and logs out. On the other side, the server switches on the printer, sends a sequence of print jobs and then switches off the printer. Since the printer's buffer is limited, the server has to wait for a `ready` signal from the printer before sending the next job.

The interesting interaction is the server's MSC program M_{spool} that combines with the client's MSC M_{print} and the printer's MSC M_{work} to form a global transaction. In M_{spool} the server checks if the printer is ready. If not, it blocks and sends the client a message `work` indicating that this is working. (In M_{print}, the message `work` is shown with a dotted line. This is a syntactic sugar to indicate an optional message—that is, this MSC represents a nondeterministic choice between two MSCs, one in which the optional message is present and another in which it is absent.) In this example, there is no secondary condition that causes the *await* to abort. Once the printer is ready, the current job is sent for printing.

Formally, in this example the global transactions are as follows, specified in the sequence (Server,Client,Printer).

$$\{\ (M_{client}, M_{login} \circ M_{logout}, M_\epsilon), (M_{printer}, M_\epsilon, M_{off} \circ M_{on}),\\ (M_{spool} \circ M_{ready}, M_{print}, M_{work})\ \}$$

We have used M_ϵ to indicate the empty MSC, signifying that a component does not play an active role in the given transaction.

5.2 Formal Semantics of MSC Programs

We provide a formal semantics for MSC programs by translating each program into combinations of basic MSCs, whose execution is controlled by the MSC progam. We translate the *await* instruction into a sequence of *if* and *waitfor*statements, as below.

$$\begin{array}{rcl} (\mathit{await}\ C_1\ \mathit{while}\ C_2 & \Longrightarrow & (\mathit{if}\ [C_2 \wedge \neg C_1]\ \mathit{atomic}\ (\ b_i\ );\\ \mathit{blocked}\ b_i\ \mathit{do}\ P_1;\ P_2) & & \mathit{waitfor}\ [\neg C_2 \vee C_1]\ ;\\ & & \mathit{if}\ [C_1]\ P_1;\\ & & P_2;) \end{array}$$

The translated program has now three 'primitive'-programming constructs, namely *if* , *atomic* and *waitfor* . We define the run-time semantics of these primitives in the next section.

5.3 Relaxing Global Behaviour on *await*

When defining the global behaviour of a system defined in terms of MSC programs, we make one relaxation to our rule that asynchronous MSCs can only

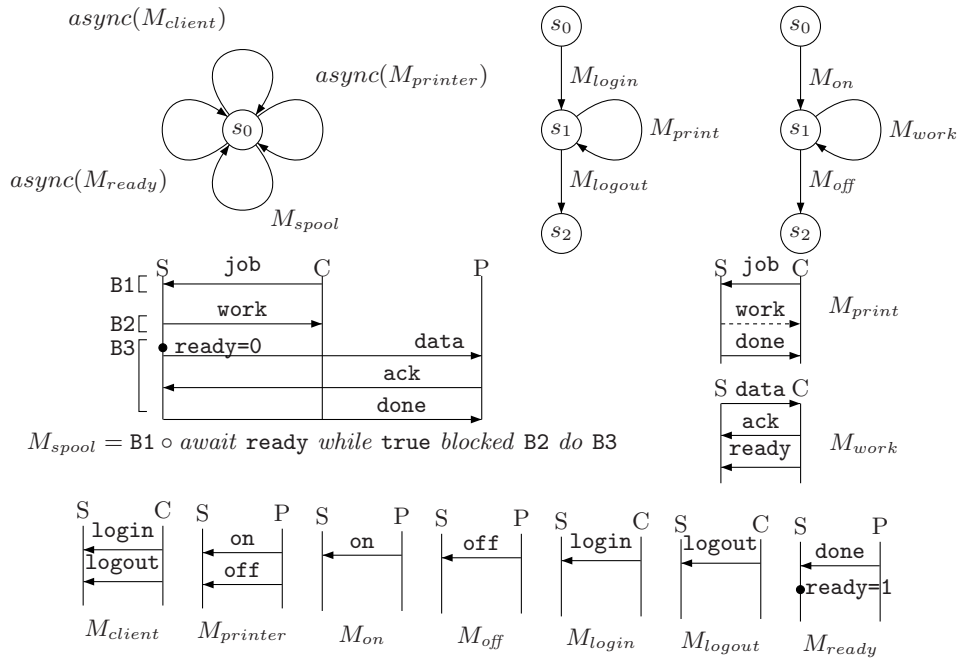

Fig. 8. PrintServer

be interleaved at MSC boundaries of the synchronous thread. We now permit asynchronous MSCs to interleave at points where the transaction involving the synchronous thread is blocked. Without this relaxation, a blocked MSC could never get unblocked. In the print server example, when an instance of the transaction $(M_{spool} \circ M_{ready}, M_{print}, M_{work})$ blocks because the printer is not ready, it means that M_{ready} from the previous instance of this transaction has not completed. M_{spool} is a synchronous transaction for the server, but the previous (asynchronous) M_{ready} is allowed to complete when the currently active M_{spool} is blocked so that the server can proceed.

6 Formal Sematics

In this section we define the full formal semantics for coordinated concurrent scenarios.

Let (S, τ, α, β) be some configuration with $S = \{s_p\}_{p \in \mathcal{P}}$. Now a component $\mathcal{A}_p$ can make either a transition or an MSC-action e as per Figure 9. And, when it makes a state transition it can either start a new transaction, or continue an existing transaction. It can execute event MSC-action e if e is enabled.

For our semantics, we assume the coloring function φ colors the events 0 if its not completed and 1 otherwise. And, the color of the event will be undefined if the corresponding p-local MSC has not started execution. We also introduce a new labelling function *atomic* that says which transaction is executing an *atomic* instruction.

Given a transaction $(t, \ell_t, M_t, \varphi_t)$, we say that a p-event $e \in M_t$ is enabled if all of the following hold:

- e is a minimal event such that $varphi_t(e) = 0$, and
- $\nexists (t', \ell'_t, M'_t, \varphi_{t'})$, such that $atomic(t', \ell'_t, p) = 1$.
- e is either a send event or is a receive whose send has been executed.
  - $e = p!q(m)$ or
  - $e = p?q(m)$, and $\varphi_t(p!q(m)) = 1$

We define the run-time semantics for the MSC program primitives as another constraint on the execution of the MSC events. We now say that a p-event $e \in M_t$ is enabled only if all the above conditions are satisfied, and the basic MSC b_i containing e is being executed by the MSC program. Given a transaction $(t, \ell_t, M_t, \varphi_t)$, the execution of the MSC program is given by the following semantics for the primitives.

$$
\begin{array}{rcll}
\textit{if } [C]\ P_1;P_2 & & & \\
\quad \textbf{\textit{when}}\ C & \Longrightarrow & P_1;P_2 & \\
\quad \textbf{\textit{when}}\ \neg C & \Longrightarrow & P_2 & \\
\textit{atomic } P_1\ ;\ P_2 & \Longrightarrow & atomic(t, \ell_t, p) = 1; & \\
 & & P_1; & \\
 & & atomic(t, \ell_t, p) = 0; & \\
 & & P_2 & \\
\textit{waitfor } [C];\ P_1 & & & \\
\quad \textbf{\textit{when}}\ C & \Longrightarrow & P_1 & \\
\quad \textbf{\textit{when}}\ \neg C & \Longrightarrow & \textit{waitfor } [C];\ P_1 & \textbf{[blocked]} \\
\textit{(basic MSC) } b_i & \Longrightarrow & \textbf{\textit{execute}}\ b_i &
\end{array}
$$

We call a run of the system, with final state (S, τ, α, β), as an accepting iff

- all the components are in their accepting state
  $\forall s_p \in S,\ s_p \in F_p$,
- all asychronous MSCs have completed execution
  $\forall p\ \beta(p)$ is undefined, and
- all transactions have been discharged
  $\tau = \phi$.

7 Expressiveness

The main motivation for introducing coordinated concurrent scenarios is to provide a notation that naturally captures the kinds of specifications that arise when designing systems with interacting components. In Section 2, we have already discussed an example that compares our notation with existing formalisms. It turns out that our approach is flexible enough to directly encode both message sequence graphs and message-passing automata.

$\mathcal{A}_p$ **starts transaction** (t, ℓ_t) **with MSC** $M = (E, \leq, \lambda)$

Configuration (S, τ, α, β), where $\alpha(p)$ is undefined

(sync execution)

when $s_p \rightarrow (s'_p, M_p, sync)$, $t_p = M_p \circ \ldots$ $\implies$ $(S \backslash \{s_p\} \cup \{s'_p\}, \tau', \alpha', \beta)$
where
$\tau_p = (t, \ell_t, M_t, \varphi_t = \{e = 0 | e \in E\})$,
$\tau' = \tau \cup \{\tau_p\}$, $\alpha' = \alpha / [p \rightarrow \tau_p]$

(async execution)

when $s_p \rightarrow (s'_p, M_p, async)$, $t_p = M_p \circ \ldots$ $\implies$ $(S \backslash \{s_p\} \cup \{s'_p\}, \tau', \alpha, \beta')$
where
$\tau_p = (t, \ell_t, M_t, \varphi_t = \{e = 0 | e \in E\})$,
$\tau' = \tau \cup \{\tau_p\}$
$\beta'(p) = \beta(p) \cup \{\tau_p\}$,
$\beta'(p') = \beta(p'), \forall p' \neq p$

$\mathcal{A}_p$ **continues transaction** (t, ℓ_t) **with msc** $M = (E, \leq, \lambda)$

Configuration (S, τ, α, β), where $\alpha(p)$ is undefined, $\tau_p = (t, \ell_t, M_t, \varphi_t) \in \tau$

(sync execution)

when $s_p \rightarrow (s'_p, M_p, sync)$, $t_p = M_p^1 \circ M_p^i \circ M_p \circ \ldots$ $\implies$ $(S \backslash \{s_p\} \cup \{s'_p\}, \tau', \alpha', \beta)$
where
$\tau'_p = (t, \ell_t, M_t, \varphi_t \cup \{e \rightarrow 0 | e \in M\})$,
$\tau' = \tau \backslash \{\tau_p\} \cup \{\tau'_p\}$, $\alpha' = \alpha / [p \rightarrow \tau'_p]$

(async execution)

when $s_p \rightarrow (s'_p, M_p, async)$, $t_p = M_p^1 \circ M_p^i \circ M_p \circ \ldots$ $\implies$ $(S \backslash \{s_p\} \cup \{s'_p\}, \tau', \alpha, \beta')$
where
$\tau'_p = (t, \ell_t, M_t, \varphi_t \cup \{e \rightarrow 0 | e \in M\})$,
$\tau' = \tau \backslash \{\tau_p\} \cup \{\tau'_p\}$,
$\beta'(p) = \beta(p) \backslash \{\tau_p\} \cup \{\tau'_p\}$,
$\beta'(p') = \beta(p'), \forall p' \neq p$

$\mathcal{A}_p$ **completes sync MSC** M **in transaction** (t, ℓ_t)

(S, τ, α, β), $\exists \tau_c \in \tau$, $\tau_c = (t, \ell_t, M_t, \varphi_t)$, $\varphi_t(e) = 1 \forall e \in M$, $\alpha(p) = \tau_c$ $\implies$ $(S, \tau, \alpha / [p \rightarrow undefined], \beta)$

$\mathcal{A}_p$ **completes async MSC** M **in transaction** (t, ℓ_t)

(S, τ, α, β), $\exists \tau_c \in \tau$, $\tau_c = (t, \ell_t, M_t, \varphi_t)$, $\varphi_t(e) = 1 \forall e \in M$, $\tau_c \in \beta(p)$ $\implies$ $(S, \tau, \alpha, \beta')$
$\beta'(p) = \beta(p) \backslash \{\tau_c\}$,
$\beta'(p') = \beta(p'), \forall p' \neq p$

$\mathcal{A}_p$ **completes transaction** (t, ℓ_t)

(S, τ, α, β), $\tau_c = (t, \ell_t, M_t, 1) \in \tau$ $\implies$ $(S, \tau \backslash \{\tau_c\}, \alpha, \beta)$

$\mathcal{A}_p$ **executes** e **in** $(t, \ell_t, M_t, \varphi_t)$

(S, τ, α, β), $\tau_c = (t, \ell_t, e \in M_t, \varphi) \in \tau$ $\implies$ $(S, \tau \backslash \{\tau_c\} \cup \{\tau'_c\}, \alpha, \beta)$
$\tau'_c = (t, \ell_t, M_t, \varphi / [e \rightarrow 1])$

Fig. 9. Complete Semantics

7.1 Encoding message sequence graphs

MSGs are global versions of what we have called p-local MSGs in which transitions can be labelled with arbitrary MSCs. We start with an MSG $\mathcal{G} = (S, S_{in}, \rightarrow, F)$. If $|S_{in}| > 1$, we create a dummy initial state s_{in} and new transitions $s_{in} \xrightarrow{M_\epsilon} s'$ for each $s' \in S_{in}$. We assume that the MSCs used in $\mathcal{G}$ all have distinct names. We assign distinct labels to each of the new empty MSCs M_ϵ that we add out of the dummy initial state. We create an isomorphic copy $\mathcal{G}_p = (S_p, S^p_{in}, \rightarrow_p, F_p)$ for each $p \in \mathcal{P}$. We convert each transition $s \xrightarrow{M} s'$ in $\mathcal{G}$ into a p-local transition $s \xrightarrow{M\downarrow_p} s'$. We create a set of transactions $\mathcal{T} = \{\{M\downarrow_p\}_{p\in\mathcal{P}} \mid M \text{ appears in } \mathcal{G}\}$. It is not difficult to see that the system $(\{\mathcal{G}_p\}_{p\in\mathcal{P}}, \mathcal{T})$ has the same global behaviour as $\mathcal{G}$.

7.2 Encoding message passing automata

Encoding MPAs is even more direct. Recall that each component p of an MPA is a finite-state automaton over the alphabet Σ_p. Corresponding to each action $p!q(m)$ we define a p-local MSC M^s_{pqm} consisting of a single message m from p to q. Similarly, for each action $p?q(m)$ we define a p-local MSC M^r_{pqm} consisting of a single message m from q to p. We relabel the send and receive actions of p by the corresponding MSCs to get a p-local MSG. We then define global transactions of the form (M^s_{pqm}, M^r_{qpm}) for all combinations $p, q \in \mathcal{P}$ and $m \in \mathcal{M}$. These global transactions allow each send event in one process to be paired up with an arbitrary matching receive event in the other process. Again, it is not difficult to see that the resulting coordinated concurrent scenario description matches the behaviour of the original MPA.

7.3 Netcharts

Our notation is closest in spirit to netcharts, introduced in [10]. A netchart is a global scenario specification with local control flow. Unlike an MSG, all processes are not required to traverse the specification in the same manner, although all transactions have to eventually be completed, as in our model. If we ignore features such as *async* and *await*, the key difference is that our specifications are local whereas netcharts are global. Thus, all processes share the same global control flow graph in a netchart, whereas we describe each process as an independent automaton. Moreover, we describe global transactions as tuples of local transactions. Multiple local transactions can share a name, allowing for many different ways to instantiate a global transaction in terms of local transactions. In a netchart, each of these combinations has to be explicitly represented in the specification. To summarize, at the vanilla level of our formalism, we can regard a netchart as the result of blowing up our local presentation into a global one. This is analogous to the difference between presenting a product of automata in terms of local components as opposed to directly describing the global state graph of the system.

8 Formal Verification

It is well known that most verification questions are undecidable for message sequence graphs and message-passing automata. Since we can simulate these formalisms in our model, we cannot expect tractable solutions to verification questions in our framework either.

In MSGs and MPAs, many problems become decidable if channels are bounded [7, 8]. This restriction makes the set of global configurations of the system finite. In our framework, there are two additional sources of infiniteness: the global semantics permits a process to initiate fresh transactions before earlier ones have completed, and *async* can generate an unbounded number of parallel threads.

One approach to address the verification problem is to identify sufficient structural conditions that make specifications tractable. However, such restrictions often limit the usefulness of the notation in practice. Instead, we follow a pragmatic approach and place upper bounds on the various resources—channel capacity, number of active synchronous transactions, and number of active asynchronous transactions. We then translate our specifications into the language of an automated verification tool. We have implemented a translator from a textual representation of coordinated concurrent scenarios to the verification tool UPPAAL [1].

8.1 Modelling using UPPAAL

For each component we use the runtime semantics to translate it into a MPA, that can be coded as an UPPAAL template. Since UPPAAL doesn't support buffered channels, and dynamic processes we work around this by creating additional process templates.

For each directed communication channel we create a "Buffer" process, that does a synchronization with the sender and receiver. UPPAAL supports no exchange of message-tags/values across a synchronization. We circumvent this by using a global variable for each channel to specify the message-tags. We transform the sends and receives as below:

- $p!q(m) \Longrightarrow gblSnd_{p_q} = m;\ buf_{p_q}!$
- $p?q(m) \Longrightarrow gblRcv_{q_p} = m;\ buf_{q_p}!$

To handle *async* , we create an independent UPPAAL template for each asynchronously executed MSC. And we make these wait for a *start* synchronization message from its component. We then instantiate a predefined (given by the bound on *async*) number of instances for each asynchronously executed MSC template. Thus, to start an *async* MSC, the component only needs to do a *start* synchronization for that MSC. The semantics of UPPAAL guarantees that only one of the many waiting templates will receive the synchronization.

Since the communication channels are named by the components, all instances of the asynchronous MSC templates can also share the same channel.

To verify properties of the specification in UPPAAL , using the technique described in [3], we first translate the property we want to verify into an MPA

that can be coded in UPPAAL template (lets call it *spec*). Now, to verify that the system conforms to *spec*, we require that every communication action be in accordance with *spec*. Which means that the sends and receives get transformed as below.

- $p!q(m) \Longrightarrow gblSnd_{p_q} = m;\ spec!;\ buf_{p_q}!$
- $p?q(m) \Longrightarrow gblRcv_{q_p} = m;\ spec!;\ buf_{q_p}!$

Now, in this setup, we can say that the system conforms to the property *spec* if it reaches its final state. In addition to constraining messages, *spec* can also specify additional properties to be verified. Modulo, modelling *async* in UPPAAL the rest of the technique is same as in [3], and we skip further details here.

9 Discussion

We have described a notation to specify concurrent components in terms of their local views of the system and argued that our approach is more natural than existing scenario based approaches, with features to capture asynchronous interleaving of scenarios as well as blocking. We can translate our models into UPPAAL to verify properties of specifications, for which we need to impose overall bounds on system resources.

A natural next step is to use these specifications to synthesize actual code for components such as device drivers. In this context, there seem to promising connections to the CLARITY paradigm [2] that provides a programming framework for concurrent code where programmers can express situations like blocking and waiting without explicitly having to manipulate low level queues.

References

1. G. Behrmann, A. Davida and K.G. Larsen: A Tutorial on Uppaal, *Proc. SFM 2004*, LNCS **3185**, Springer-Verlag (2004) 200–236.
2. P. Chandrasekaran, C.L. Conway, J.M. Joy, S.K. Rajamani: Programming asynchronous layers with CLARITY, *Proc. ESEC/SIGSOFT FSE 2007*, IEEE (2007) 65–74.
3. P. Chandrasekaran and M. Mukund: Matching Scenarios with Timing Constraints, *Proc. FORMATS 2006*, LNCS **4202**, Springer-Verlag (2006) 98–112.
4. V. Diekert and G. Rozenberg (eds): *The Book of Traces*, World Scientific (1995).
5. T. Gazagnaire, B. Genest, L. Hélouët, P.S. Thiagarajan and S. Yang: Causal Message Sequence Charts. *Proc. CONCUR 2007*, LNCS **4703**, Springer-Verlag (2007) 166–180.
6. B. Genest, L. Hélouët and A Muscholl: High-Level Message Sequence Charts and Projections, *Proc. CONCUR 2003*, LNCS **2761**, Springer-Verlag (2003) 308–322.
7. B. Genest, D. Kuske and A. Muscholl: A Kleene Theorem for a Class of Communicating Automata with Effective Algorithms. *Proc DLT 2004*, LNCS **3340**, Springer-Verlag (2004) 30–48.
8. 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.

9. ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU, Geneva (1999).
10. M. Mukund, K. Narayan Kumar and P.S. Thiagarajan: Netcharts: Bridging the gap between HMSCs and executable specifications, *Proc. CONCUR 2003*, LNCS **2761**, Springer-Verlag (2003) 293–307.
11. A. Roychoudhury and P.S. Thiagarajan: Communicating transaction processes. *Proc. ACSD'03*, IEEE Press (2003) 157–166.