Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 240 (2009) 149–166
www.elsevier.com/locate/entcs

Multiple Synchrony in MSC

Flávia Falcão Juliano Iyoda[1] Augusto Sampaio

Centro de Informática - UFPE
Cidade Universitária, Recife-PE
Brazil

Abstract

We propose an extension to Message Sequence Charts (MSC); MSC diagrams comprise processes (called instances) and messages. Messages in MSC are either asynchronous or method calls. Our extension adds multiple synchronous messages. We present a transformation algorithm that takes as input a diagram in the extended MSC and generates an equivalent one in the standard MSC. The synchronous messages are transformed to the standard notation via the introduction of several control messages. We also define a semantics for MSC (both the standard and our extension) using the process algebra CSP. Both instances and messages in MSC are characterised as CSP processes. This semantics allows us to formally establish the equivalence between an extended MSC diagram and its corresponding standard diagram (generated by the transformation algorithm). Although our strategy is application independent, the motivation came from an attempt to generate test scripts from MSC diagrams describing the behaviour of mobile phone devices.

Keywords: Message Sequence Charts, MSC, process algebra, multiple synchrony, synchronous communication, Communicating Sequential Processes, CSP.

1 Introduction

Message Sequence Chart (MSC) is a visual trace language, extensively used in academy and industry, to describe the communication behaviour of system components and their environment. The MSC syntax and semantics are now a standard defined by the International Telecommunication Union (ITU) [11].

A typical MSC diagram comprises several processes, called *instances*, and messages exchanged among these processes. There are two kinds of messages: asynchronous and method calls. Asynchronous messages never block the sender, always take a finite amount of time to reach the receiver and are never instantaneous. Method calls block the sender until the receiver sends a return message (similar to the Sequence Diagrams messages of UML [2]). Although a method call can be used to model a synchronous message between two instances, modelling multiple

[1] Juliano Iyoda wishes to thank FACEPE for the financial support.

doi:10.1016/j.entcs.2009.05.050

instances synchronising over the same event might not be simple (nor elegant) with method calls, let alone with asynchronous messages.

The motivation for this work has originated from an effort to model scenarios of mobile device applications for the purpose of automatic test generation. We used the PowerToolKit (PTK) [1], a tool developed by the Motorola Labs which automatically generates test cases from MSC diagrams. Nevertheless, modelling some applications, which involved multi-synchronisation, with standard MSC has resulted in diagrams difficult to understand or with a behaviour different from the intended one. Despite this motivation, our approach is application independent.

We propose an extension to MSC in order to allow messages to be synchronous. The proposed synchronous messages denote events that are instantaneous (we abstract the real time duration for a communication to be established) and may involve *multiple instances*. Our extension is conservative in the sense that it allows diagrams to contain both synchronous and asynchronous messages. We developed a transformation algorithm which takes an extended MSC diagram and generates a diagram in the standard MSC. This transformation algorithm implements synchronous messages as a sequence of asynchronous messages following a particular handshake protocol.

A second contribution of this work is the definition of a semantics for MSC (both the standard and our extension) using the process algebra CSP [16].The semantics is defined in an algebraic way: instances and *messages* of an MSC diagram are CSP processes running in parallel. The formalisation of MSC in CSP allows us to show the equivalence between an extended MSC diagram and its corresponding standard diagram (generated by the transformation algorithm). Moreover, modelling MSC as a CSP process allows us to reason about MSC diagrams by using the rich set of algebraic laws of CSP, as well as its tools, like FDR2 [8] and CSP-prover [10].

There are several proposals of extensions to MSC. Different features have been added to the notation, like liveness [6], scenario triggering [17], object-orientation [4] and shared-variable communication [9]. Not surprisingly, synchronous messages have also been proposed in some works [3,6,12,13]. However, to the best of our knowledge, none have introduced *multiple synchrony* or have used CSP as a semantic model for MSC; as already mentioned, this has the advantage of immediate mechanised reasoning. More on related work can be found in Section 5.

This paper is organised as follows. Section 2 introduces the basics of the standard MSC, followed by the definition of our extension described in Section 3. Section 4 presents the semantics of MSC in CSP and, finally, conclusions, applications and future work are addressed in Section 5.

2 MSC

Message Sequence Chart (MSC) is a trace language for describing the communication behaviour of system components and the environment. The MSC syntax and semantics are defined by the ITU [11]. MSC has two syntaxes: a graphical syntax and a textual one. The graphical syntax is the most commonly used, while

its textual form is mainly adopted by tools that perform automatic formal analysis. Figure 1 shows a simple example of the MSC graphical notation.

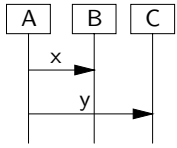

Fig. 1. A simple example of MSC.

An MSC diagram is formed of instances of distributed components involved in some communication. In Figure 1, we declare the instances A, B and C. The communication between instances is represented by arrows which denote *messages* sent from one instance to another. Each message is related to two *events*: a send event and a receive event. In the example, the message x is related to the events send_x and receive_x; these communications are asynchronous.

The MSC semantics dictates that for each instance axis, time progresses as we move down from top to bottom. However, no global time exists. These different instances are only loosely coupled in time. For example, if an instance sends out two messages to two distinct instances, the messages can arrive in any order regardless the order they were sent. For example, the events receive_x and receive_y in Figure 1 can occur in any order in time. MSC event occurrences follow a partial order defined by: 1) the *instance order*: the events are ordered over the axis of each instance; and 2) the *send-receive relation*: the receive event of a message always happens after the send event of the same message. For example, we can derive the following relations from the messages in Figure 1:

$$\begin{aligned} &\mathsf{send\_x} < \mathsf{receive\_x} \\ &\mathsf{send\_y} < \mathsf{receive\_y} \\ &\mathsf{send\_x} < \mathsf{send\_y}, \end{aligned}$$

where $\mathsf{a} < \mathsf{b}$ means that event a occurs before event b. Therefore we can see that a basic MSC can denote more than one trace of events. For instance, the traces of the MSC depicted in Figure 1 are:

$$\begin{aligned} &\langle \mathsf{send\_x}, \mathsf{receive\_x}, \mathsf{send\_y}, \mathsf{receive\_y} \rangle, \\ &\langle \mathsf{send\_x}, \mathsf{send\_y}, \mathsf{receive\_x}, \mathsf{receive\_y} \rangle, \\ &\langle \mathsf{send\_x}, \mathsf{send\_y}, \mathsf{receive\_y}, \mathsf{receive\_x} \rangle. \end{aligned}$$

The language is further enhanced by other operators and hierarchical constructors (like parallel composition and choice) to compose diagrams in a higher level of abstraction. As we are not dealing with them in this work, we omit them here for conciseness.

3 Synchronous MSC

Synchronous communication provides a convenient model to specify interactive systems in a higher level of abstraction in comparison to asynchronous communication. In particular, it abstracts away implementation details of handshake protocols, allowing the designer to focus on the communication at the application level. In this section we propose an extension to MSC that allows the user to specify synchronous communication, where two or more MSC instances can be synchronised on the same event. Moreover we propose an algorithm to convert from the extended MSC to the standard MSC.

Figure 2 shows an example of a conference call among three people in the *standard* MSC notation. The instance Flavia starts the conference by sending an invitation to the other two participants.

Flavia starts the conversation by sending two Hello messages, one to each participant. Augusto replies by saying How are you to both Flavia and Juliano. This diagram denotes several traces. For instance:

$$\begin{aligned}&\langle \mathsf{send\_inviteJ}, \mathsf{receive\_inviteJ}, \mathsf{send\_inviteA}, \mathsf{receive\_inviteA},\\ &\mathsf{send\_ackJ}, \mathsf{receive\_ackJ}, \mathsf{send\_ackA}, \mathsf{receive\_ackA},\\ &\mathsf{send\_Hello}_0, \mathsf{send\_Hello}_1, \mathsf{receive\_Hello}_1,\\ &\mathsf{send\_How\ are\ you}_0, \mathsf{send\_How\ are\ you}_1, \mathsf{receive\_How\ are\ you}_1,\\ &\mathsf{receive\_Hello}_0, \mathsf{receive\_How\ are\ you}_0\rangle\end{aligned}$$

Although this could be a valid scenario for some systems, such a trace might be illegal for others. If we want to model a system that prevents traces where Hello messages are interleaved with How are you messages, we have to introduce several control messages to guarantee atomicity and multiple synchrony. For instance, Augusto should only be able to send a message after receiving the system's permission.

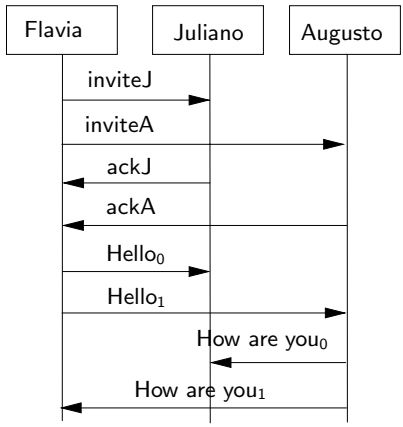

Fig. 2. A conference call with 3 people.

Alternatively, Figure 3 shows the same scenario using our extension to MSC. This diagram contains both synchronous and asynchronous messages. The messages that invite participants to the conference call are asynchronous. For instance,

the invitation to Augusto can be sent before the arrival of Juliano's invitation. In contrast, once all participants have agreed to join the call, the conversation is carried out in a synchronous way. The messages Hello and How are you are atomic, and synchronise over all participants simultaneously. Hello was sent by Flavia and How are you, by Augusto. Although this information is indicated in the diagram (Figure 3), it is omitted here for simplicity.

The diagram shown in Figure 3 denotes several traces.

⟨send_inviteJ, send_inviteA, receive_inviteJ, send_ackJ, receive_ackJ, receive_inviteA, send_ackA, receive_ackA, Hello, How are you⟩,

⟨send_inviteJ, receive_inviteJ, send_ackJ, send_inviteA, receive_ackJ, receive_inviteA, send_ackA, receive_ackA, Hello, How are you⟩, ...

Nevertheless, notice that the synchronous messages are related to a single event.

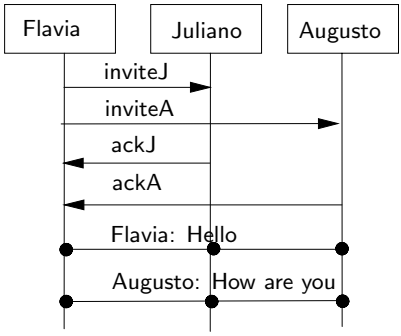

Fig. 3. A synchronous MSC diagram.

As the standard MSC notation is widely used by many designers and, especially, by many tools, we propose a transformation algorithm that generates a standard MSC diagram from the extended one. The algorithm introduces several control messages to implement the synchronous events.

The first step of the algorithm transforms each synchronous message into a sequence of asynchronous messages. This sequence must adhere to the following *generation condition*: every instance involved in the multiple synchrony must either send or receive at least one asynchronous message. For example, the message Hello in Figure 3 is sent by Flavia to all, and synchronises the instances Flavia, Juliano and Augusto. So, these instances must either send or receive an *asynchronous* Hello. In our example, two asynchronous messages sent from Flavia to Juliano and Augusto are generated (see Figure 4). These messages satisfy the generation condition and model the first sentence of the dialogue, where Flavia says Hello to everybody. In fact, we could have added more Hello messages and in a different order if we wished, provided that the generation condition is satisfied. These choices are design decisions left to the engineer. The generation condition minimally ensures that all instances in multiple synchrony are somehow participating in a communication in the asynchronous domain.

Figure 4 shows this step applied to the diagram of Figure 3. The sequence of asynchronous messages generated from a single synchronous message is called a *synchronous section*. The synchronous sections are indicated in Figure 4 by dashed boxes, which are *not* part of the MSC diagram. For instance, the messages Hello_0 and Hello_1 in Figure 4 belong to the same synchronous section as they are derived from the synchronous message Hello in Figure 3. The messages inviteJ, inviteA, ackJ and ackA, which were originally asynchronous, remain the same. We call the instances that either send or receive messages inside a synchronous section S, the *active instances* of S. For example, the active instances of the synchronous section 1 are Flavia, Juliano and Augusto.

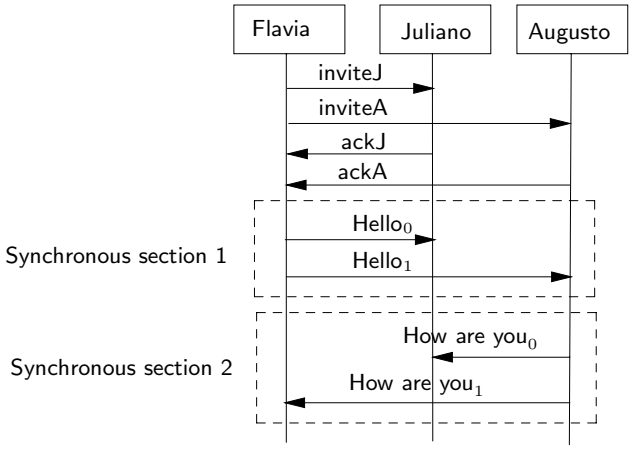

Fig. 4. First step of the algorithm.

The second step introduces control messages τ in order to isolate each synchronous section from the messages that occur before and after it. These messages are introduced according to the following rules.
For each synchronous section:

Above Add τ messages above the synchronous section from all active instances to the active instances which produce the first "send" events of the synchronisation section.

Below Add τ messages below the synchronous section from the active instances which produce the last "receive" events of the synchronisation section to all the active instances.

In both cases, we do not need to send τ messages from an instance to itself.

The τ messages above the synchronisation section force the completion of all preceding events of messages that occur before the section. This happens because every instance has to receive or send a τ message before synchronising. As every instance obeys a total order of events on its axis, no preceding event on that axis occurs after *send_*τ or *receive_*τ events (recall the *instance order* described in Section 2). Moreover, all τ messages arrive before the first messages of the synchronisation section are sent. The *send-receive* relation (Section 2) ensures the completion of all τ messages before starting the synchronisation. Therefore τ messages isolate the

synchronisation section to ensure that every preceding event occurs in advance so all active instances get ready to synchronise. Similarly, the τ messages below the synchronisation section force every event following the section to occur after its last messages are received. As mentioned above, we can omit τ messages sent from an instance to itself. According to the algorithm, such instances are those which either initiate or terminate the synchronisation. Therefore, these instances are precisely those which are either the target of τ messages sent from others (above) or the source of τ messages sent to others (below). This automatically guarantees the isolation of such instances.

Figure 5 shows the transformation from the extended MSC to the standard MSC using the rules above. Again the dashed boxes only indicate the synchronous sections (now extended with the τ messages) but are *not* part of the MSC diagram. Note that the message τ_0 forces any event above the synchronisation section on the Juliano's axis to occur before $\mathsf{send\_Hello}_0$, which is the only initial event of this section (in general, more than one initial events might be present due to messages sent in parallel). Any such event must be followed by send_τ_0, $\mathsf{receive}\_\tau_0$ and $\mathsf{send\_Hello}_0$. Similarly, the τ messages inserted below the synchronisation section prevents any event to occur before the last events of the section. Notice that in this example, both $\mathsf{receive\_Hello}_0$ and $\mathsf{receive\_Hello}_1$ may be considered the last events of the synchronisation section. There is no total order for their occurrences, in contrast to the only possible first event $\mathsf{send\_Hello}_0$. This justifies the addition of τ_2 and τ_3 for $\mathsf{receive\_Hello}_1$, and τ_4 and τ_5 for $\mathsf{receive\_Hello}_0$. The τ messages related to $\mathsf{How\ are\ you}_0$ and $\mathsf{How\ are\ you}_1$ follow a similar reasoning. Some τ messages are redundant in this example, but this might not be the case in general.

We still have a final issue to address: How do we relate the asynchronous diagram in Figure 5 to the extended one in Figure 3? The diagram in Figure 5 denotes several traces like:

$$\begin{aligned}&\langle \mathsf{send\_inviteJ}, \mathsf{receive\_inviteJ}, \mathsf{send\_inviteA}, \mathsf{receive\_inviteA}, \mathsf{send\_ackJ},\\ &\mathsf{send\_ackA}, \mathsf{receive\_ackJ}, \mathsf{receive\_ackA}, \mathsf{send}\_\tau_0, \mathsf{receive}\_\tau_0,\\ &\mathsf{send}\_\tau_1, \mathsf{receive}\_\tau_1, \mathsf{send\_Hello}_0, \mathsf{receive\_Hello}_0, \mathsf{send\_Hello}_1, \mathsf{receive\_Hello}_1,\\ &\mathsf{send}\_\tau_2, \mathsf{receive}\_\tau_2, \mathsf{send}\_\tau_3, \mathsf{receive}\_\tau_3, \mathsf{send}\_\tau_4, \mathsf{receive}\_\tau_4,\\ &\mathsf{send}\_\tau_5, \mathsf{receive}\_\tau_5, \mathsf{send}\_\tau_6, \mathsf{receive}\_\tau_6, \mathsf{send}\_\tau_7, \mathsf{receive}\_\tau_7, \mathsf{send\_How\ are\ you}_0,\\ &\mathsf{receive\_How\ are\ you}_0, \mathsf{send\_How\ are\ you}_1, \mathsf{receive\_How\ are\ you}_1, \mathsf{send}\_\tau_8,\\ &\mathsf{receive}\_\tau_8, \mathsf{send}\_\tau_9, \mathsf{receive}\_\tau_9, \mathsf{send}\_\tau_{10}, \mathsf{receive}\_\tau_{10}, \mathsf{send}\_\tau_{11}, \mathsf{receive}\_\tau_{11}\rangle\end{aligned}$$

We need a mapping which transforms such traces into the traces of Figure 3. For instance, the equivalent trace of the one shown above in the extended MSC is:

$$\begin{aligned}&\langle \mathsf{send\_inviteJ}, \mathsf{receive\_inviteJ}, \mathsf{send\_inviteA}, \mathsf{receive\_inviteA}, \mathsf{send\_ackJ},\\ &\mathsf{send\_ackA}, \mathsf{receive\_ackJ}, \mathsf{receive\_ackA}, \mathsf{Hello}, \mathsf{How\ are\ you}\rangle\end{aligned}$$

Both diagrams of figures 3 and 5 denote several traces. We have to find a mapping between their events which relate them all.

From the example above, it is not hard to see that this mapping has to remove

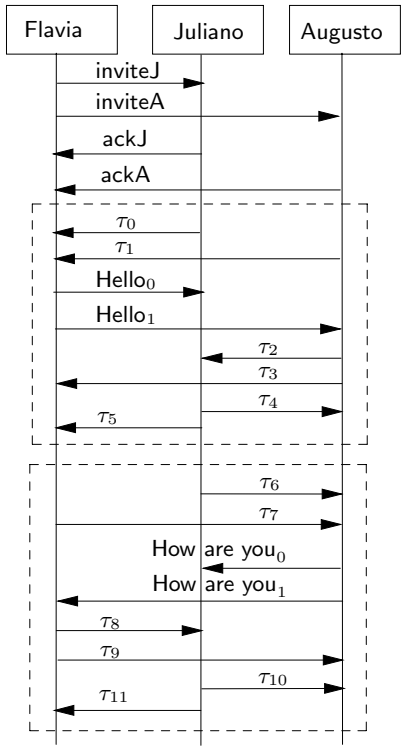

Fig. 5. The standard MSC diagram derived from the extended one.

all send_τ_i and receive_τ_i and has to relate, say, the set

$$\{\mathsf{send\_Hello}_0, \mathsf{receive\_Hello}_0, \mathsf{send\_Hello}_1, \mathsf{receive\_Hello}_1\}$$

to the event Hello. Similarly, the events {send_How are you$_0$, receive_How are you$_0$, send_How are you$_1$, receive_How are you$_1$} must be associated with How are you.

Modelling systems at different levels of abstraction has the advantage of capturing several architectural views. In general these views are presented using different alphabets. Therefore it is necessary to provide a mapping between them [5]. Synchronous communication is observed by the environment as an atomic event. At this level of abstraction, we observe the system at specific relevant moments. However, any synchronous communication is implemented by some kind of handshake protocol dealing with asynchronous communication. A designer of such system must observe it in a higher frequency (see Figure 6). Our mapping simply ignores control messages τ and allows system observations only when the last asynchronous message of the handshake occurs. This last event characterises a successful handshake and, therefore, the end of a transaction. These concepts are partly inspired by the time abstraction concept used in hardware design [15]. The asynchronous messages present in the extended MSC are mapped to themselves. The following sections describe how this mapping can be formally defined to provide a solid justification for our transformation algorithm.

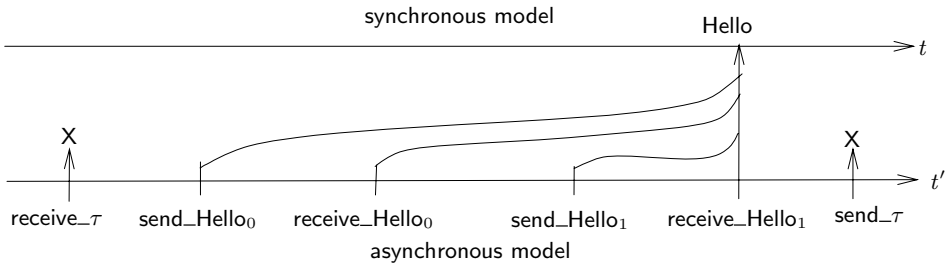

Fig. 6. Mapping asynchronous events to synchronous ones.

4 MSC Semantics

In this section we propose a formal semantics for MSC based on CSP (Communicating Sequential Processes). This semantics allows us to formally define the transformation from synchronous MSC to asynchronous MSC and to reason about the relationship between them, possibly using tools such as FDR: a model checker for CSP.

4.1 CSP

CSP allows describing systems in terms of processes that operate independently and interact with each other through *synchronous* message passing-communication. CSP offers a rich repertoire of process algebra operators and a few primitive constructors. Communications in CSP are atomic and instantaneous events. Events can transmit information through unidirectional channels. The occurrence of an event characterises a communication with another process or with the environment.

Processes are behavioural description units described by the way they communicate with their environment. There are two primitive processes: *Stop*, which represents canonical deadlock; and *Skip*, which represents successful termination. The simplest CSP operator is the prefix. Let a be an event and P a process, then $a \rightarrow P$ is the process that waits indefinitely to communicate a and, after that, behaves like the process P.

For example, suppose we define $(S = a \rightarrow b \rightarrow c \rightarrow \mathit{Skip})$, $(T = r \rightarrow a \rightarrow U)$ and $(U = d \rightarrow \mathit{Skip})$. S waits for the events a, b and c to happen in sequence and then terminates successfully. T communicates the events r and a, and behaves like U. Finally U waits for the event d before terminating with success.

The set of all events that a process can engage in is called its *alphabet*. For example, the alphabets of S, T and U are $\{a, b, c\}$, $\{r, a\}$ and $\{d\}$, respectively.

Processes can be composed in a sequential way using the sequential operator. In $(P = Q\ ;\ R)$ the process P behaves like Q initially. If Q terminates successfully, the process behaves like R. If Q does not terminate, the process P also does not terminate and never behaves like R.

Alternative behaviour is provided by two operators: internal and external choice. The internal or non-deterministic choice $\sqcap$ allows the future behaviour of a process to be defined internally to the system, with no control from the environment. External

or deterministic choice $\Box$ allows the environment to choose between two processes by communicating the initial event of one of them. For example, $(a \rightarrow \mathit{Stop}) \Box (b \rightarrow \mathit{Stop})$ communicates either a or b, depending on what the environment offers firstly. We denote by $(?x : A \rightarrow P)$ the process which accepts any element of the set A and behaves like P.

Processes can also be combined in parallel. Let A and B be sets of events. The *alphabetised parallel* operator ${}_A||_B$ combines processes that must cooperate upon every event in the set $A \cap B$. Events outside the intersection are communicated with the environment. For instance, the process

$$(a \rightarrow b \rightarrow b \rightarrow \mathit{Stop}) \ {}_{\{a,b\}}||_{\{b,c\}} \ (b \rightarrow c \rightarrow b \rightarrow \mathit{Stop})$$

behaves like $(a \rightarrow b \rightarrow c \rightarrow b \rightarrow \mathit{Stop})$. Usually processes combined with the alphabetised parallel operator follow the pattern

$$(P \ {}_X||_{Y \cup Z} \ (Q \ {}_Y||_Z \ R)).$$

If we have to compose a large network of processes this way, the notation becomes very clumsy. A convenient indexed notation is provided to handle very large compositions like this [16].

$$\Big|\Big|_{i=1}^{n}(P_i, X_i) \ = \ P_1 \ {}_{X_1}||_{X_2 \cup \ldots \cup X_n} \ (\ldots (P_{n-1} \ {}_{X_{n-1}}||_{X_n} \ P_n) \ldots)$$

The *interface parallel* operator $\underset{C}{||}$ explicitly shows the set of processes C over which the component processes synchronise on. For example, the process $(P \underset{C}{||} Q)$ synchronises P and Q in all communication events in the set C. Events outside C proceed independently.

The *interleaving* operator $|||$ combines processes in parallel which are completely independent from each other. The process $(P \ ||| \ Q)$ behaves as both P and Q simultaneously. The events are interleaved in time. The indexed notation for interleaving is defined as follows.

$$\Big|\Big|\Big|_{i=1}^{n} P_i \ = \ P_1 \ ||| \ \ldots \ ||| \ P_n$$

Events can be made invisible to the environment by the hiding operator $\setminus$. This operator is used to remove some events from the interface. Let P be a process and A be a set of events. All the events in A are removed from the interface of the process $P \setminus A$.

The simplest semantic model of a CSP process is that which denotes a set of traces. For instance, the trace $\langle a, b \rangle$ is a trace of the process $P = a \rightarrow b \rightarrow c \rightarrow \mathit{Skip}$. The set of all traces of a process P is denoted by $\mathit{traces}(P)$. In the traces model, a process P is refined by a process Q, denoted by $P \sqsubseteq_T Q$, whenever $\mathit{traces}(Q) \subseteq \mathit{traces}(P)$. Two processes P and Q are equivalent in the traces model whenever $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$. There are other models of CSP which are more

elaborate than the traces model. The *failures* model takes into account the events a process can refuse to do. The *failures/divergences* model takes into account the possibility of a process to enter an infinite sequence of internal actions (called a *divergence*). These models also have their notions of refinement, denoted by $P \sqsubseteq_F Q$ for the failures model and $P \sqsubseteq_{FD} Q$ for the failures/divergences model (see [16] for their formal definitions).

4.2 Formal Semantics for Standard and Extended MSC in CSP

The semantics of an MSC diagram is defined as a CSP process. This mapping is defined based on the partial order of events used to generate the traces of an MSC diagram. Inspired by the MSC trace generation rules, we look at a diagram from two perspectives: the vertical and the horizontal dimensions. The first perspective works only with the vertical axis where the instances run. Each instance becomes a CSP process defined by the sequence of events that occurs along its axis. For example, the instance A in Figure 1 is represented by the following process.

$$A = send\_x \rightarrow send\_y \rightarrow Skip$$

The CSP events *send_x* and *send_y* are precisely those events that occur along the axis of the instance A in the diagram of Figure 1. Similarly, the instances B and C have the following CSP representation.

$$B = receive\_x \rightarrow Skip$$
$$C = receive\_y \rightarrow Skip$$

The horizontal dimension captures the messages themselves. Every message of an MSC diagram becomes a process defined by two events: the send and the receive events. The messages in Figure 1 are defined as follows.

$$x = send\_x \rightarrow receive\_x \rightarrow Skip$$
$$y = send\_y \rightarrow receive\_y \rightarrow Skip$$

These two dimensions are then combined to form the formal semantics of an MSC diagram. The partial order of the events defined in a diagram naturally occurs in the CSP process once we compose the processes that represent the two dimensions in parallel. The parallel composition is defined according to the following rule.

Let $instance_i$ be a process that represents an instance, and $message_j$ a process that represents a message. Let Σ be the union of the alphabets of all processes. The semantics of a *standard* MSC diagram is defined by

$$standardMSC = (|||_{i=1}^{n} instance_i) \underset{\Sigma}{||} (|||_{j=1}^{m} message_j)$$

For the diagram example in Figure 1, the process that captures its semantics is

defined as

$$simpleMSC = (A \;|||\; B \;|||\; C) \underset{\Sigma}{||} (x \;|||\; y)$$

where $\Sigma = \{send\_x, receive\_x, send\_y, receive\_y\}$.

The traces generated by this CSP process are exactly the same defined by the corresponding MSC diagram. Intuitively, each process enforces the partial order rules (recall the *instance order* and the *send-receive relation* introduced in Section 2). The processes $instance_i$ ensure the correct order of events along the life axis of each instance. The processes $message_j$ enforce that every send event comes before a receive event. In the standard MSC, there is no synchronisation among instances as they always communicate via asynchronous messages. Messages also do not interact among themselves. Thus we can group them together using interleaving. However, messages and instances do synchronise, which is captured by their parallel composition. For instance, the processes B and x synchronise on the event $receive\_x$.

In the case of the proposed MSC with multi synchrony, the primitive processes are defined in the same way. For instance, the processes for the extended MSC depicted in Figure 3 are shown below.

$$\begin{aligned}
&Flavia = send_inviteJ \rightarrow send_inviteA \rightarrow receive_ackJ \rightarrow receive_ackA \rightarrow \\
&\qquad Hello \rightarrow How\ are\ you \rightarrow Skip \\
&Juliano = receive_inviteJ \rightarrow send_ackJ \rightarrow Hello \rightarrow How\ are\ you \rightarrow Skip \\
&Augusto = receive_inviteA \rightarrow send_ackA \rightarrow Hello \rightarrow How\ are\ you \rightarrow Skip \\
&m_inviteJ = send_inviteJ \rightarrow receive_inviteJ \rightarrow Skip \\
&m_inviteA = send_inviteA \rightarrow receive_inviteA \rightarrow Skip \\
&m_ackJ = send_ackJ \rightarrow receive_ackJ \rightarrow Skip \\
&m_ackA = send_ackA \rightarrow receive_ackA \rightarrow Skip \\
&m_Hello = Hello \rightarrow Skip \\
&m_HowAreYou = How\ are\ you \rightarrow Skip
\end{aligned}$$

Note that the synchronous messages Hello and How are you are represented as a single event in CSP. Nevertheless, the top-level process is constructed in a slightly different way. As the instances now synchronise among themselves, we have to use the parallel operator to group them.

Let $instance_i$ and $message_j$ be processes which represent an instance and a message, respectively. Let α_i be the alphabet of $instance_i$ and $\Sigma = \bigcup_{i=1}^{n} \alpha_i$. We define the semantics of *extended* MSC diagrams as follows.

$$extendedMSC = (\Big|\Big|_{i=1}^{n}(instance_i, \alpha_i)) \underset{\Sigma}{||} (\Big|\Big|\Big|_{j=1}^{m}(message_j))$$

The top-level process for the example of Figure 3 is shown below.

$$\begin{aligned} confCall = (&Flavia \ {}_{\alpha_F}||_{\alpha_J \cup \alpha_A} \ (Juliano \ {}_{\alpha_J}||_{\alpha_A} \ Augusto)) \underset{\Sigma}{||} \\ &(m\_inviteJ \ ||| \ m\_inviteA \ ||| \ m\_ackJ \ ||| \ m\_ackA \ ||| \ m\_Hello \ ||| \\ &m\_HowAreYou) \end{aligned}$$

where $\alpha_F = \{send\_inviteJ, send\_inviteA, receive\_ackJ, receive\_ackA, Hello, How\ are\ you\}$, $\alpha_J = \{receive\_inviteJ, send\_ackJ, Hello, How\ are\ you\}$, $\alpha_A = \{receive\_inviteA, send\_ackA, Hello, How\ are\ you\}$ and $\Sigma = \alpha_F \cup \alpha_J \cup \alpha_A$.

The semantics of the standard MSC is actually a special case of the semantics of the extended MSC. The alphabets of the instances $instance_i$ in the standard MSC form a partition; therefore, their alphabets are pairwise disjoint. From the definition of the parallel operator, these processes run independently: in interleaving.

4.3 An Equivalence Notion for Standard and Extended MSC

In order to show that our translation from extended to standard MSC diagrams preserve behaviour, we need to define an equivalence notion for such diagrams. Clearly, we cannot compare these diagrams directly, since their alphabets are not the same; as illustrated by Figure 6, a synchronous message can be regarded as an abstraction of a sequence of asynchronous messages. Therefore, we need to take into account such a mapping to be able to define an equivalence notion for extended and standard MSC diagrams.

Let *Sync* be the set of synchronous events of an extended diagram whose semantics is given by the CSP process *extended*. Similarly, let *Async* be the set of asynchronous messages of the standard diagram whose semantics is given by the CSP process *standard*. These processes are compared through a mapping $M : Sync \mapsto \wp(\text{seq } Async)$ that relates each (abstract) synchronous message into a *set* of corresponding traces of (concrete) asynchronous messages. Intuitively, we consider *extended* and *standard* equivalent diagrams if and only if replacing the synchronous messages with the corresponding concrete traces in *extended* results in a process, say *mappedExtended*, equivalent to *standard* in the failures/divergences model of CSP.

As an example, we show below the corresponding sequences of asynchronous messages related to the synchronous message Hello (see Figure 5).

$$\begin{aligned} M(Hello) \mapsto \{ \ &\langle send\_Hello_0, receive\_Hello_0, send\_Hello_1, receive\_Hello_1 \rangle, \\ &\langle send\_Hello_0, send\_Hello_1, receive\_Hello_0, receive\_Hello_1 \rangle, \\ &\langle send\_Hello_0, send\_Hello_1, receive\_Hello_1, receive\_Hello_0 \rangle \ \} \end{aligned}$$

Our first step is to build the process *mappedExtended*. This can be achieved by first characterising the mapping as a CSP process.

$$\begin{aligned} Map(Async, Sync) = &(?async : Async \rightarrow Map(Async, Sync)) \\ &\Box \ (?sync : Sync \rightarrow P(M(sync))) \\ &\Box \ (f \rightarrow Skip) \end{aligned}$$

where $P(M(sync))$ denotes the process whose traces are those returned by $M(sync)$. For instance, $P(M(Hello))$ denotes the process

$$
\begin{aligned}
&send\_Hello_0 \rightarrow \\
&\quad (receive\_Hello_0 \rightarrow send\_Hello_1 \rightarrow receive\_Hello_1 \rightarrow Map(Async, Sync)) \\
&\quad \Box\ (send\_Hello_1 \rightarrow (receive\_Hello_0 \rightarrow receive\_Hello_1 \rightarrow Map(Async, Sync)) \\
&\qquad\qquad \Box\ (receive\_Hello_1 \rightarrow receive\_Hello_0 \rightarrow Map(Async, Sync))).
\end{aligned}
$$

The process *Map* is designed to run in parallel with *extended*. They synchronise in every event. Whenever they synchronise over an asynchronous event, *Map* simply recurses and no mapping is done. In contrast, if they synchronise over a synchronous event *sync*, all the corresponding sequences of asynchronous events of *sync* are generated by $P(M(sync))$. The event f is used to detect termination of *extended* and, consequently, of *Map*.

The process *mappedExtended* can then be defined as:

$$
\begin{aligned}
mappedExtended(Async, Sync) = ((extended;\ (f \rightarrow Skip)) \underset{\Sigma \cup \{f\}}{||} \\
Map(Async, Sync)) \setminus (Sync \cup \{f\})
\end{aligned}
$$

where Σ is the alphabet of *extended*. Notice that *extended* is composed in sequence with $(f \rightarrow Skip)$. The event f signals to *Map* the termination of *extended*. The final step hides the synchronous events and f from *mappedExtended*, leaving it with asynchronous events only.

We say that *standard* and *extended* are equivalent if and only if the following holds:

$$
\begin{aligned}
&mappedExtended(Async, Sync) \ \sqsubseteq_{FD}\ (standard \setminus taus) \\
&(standard \setminus taus) \ \sqsubseteq_{FD}\ mappedExtended(Async, Sync)
\end{aligned}
$$

where *taus* is the set of all τ events introduced. The advantage of this characterisation is that it can be mechanically checked using FDR.

As an example, in the reminder of this section we show that the translation of the diagram in Figure 3 into that in Figure 5 does preserve behaviour. First we define the CSP process for the diagram of Figure 5. Every instance and every

message becomes a process as shown below.

$$
\begin{aligned}
Flavia' &= send\_inviteJ \rightarrow send\_inviteA \rightarrow receive\_ackJ \rightarrow receive\_ackA \rightarrow \\
&\quad receive\_\tau_0 \rightarrow receive\_\tau_1 \rightarrow send\_Hello_0 \rightarrow send\_Hello_1 \rightarrow \\
&\quad receive\_\tau_3 \rightarrow receive\_\tau_5 \rightarrow send\_\tau_7 \rightarrow receive\_How\ are\ you_1 \rightarrow \\
&\quad send\_\tau_8 \rightarrow send\_\tau_9 \rightarrow receive\_\tau_{11} \rightarrow Skip \\
Juliano' &= receive\_inviteJ \rightarrow send\_ackJ \rightarrow send\_\tau_0 \rightarrow receive\_Hello_0 \rightarrow \\
&\quad receive\_\tau_2 \rightarrow send\_\tau_4 \rightarrow send\_\tau_5 \rightarrow send\_\tau_6 \rightarrow receive\_How\ are\ you_0 \rightarrow \\
&\quad receive\_\tau_8 \rightarrow send\_\tau_{10} \rightarrow send\_\tau_{11} \rightarrow Skip \\
Augusto' &= receive\_inviteA \rightarrow send\_ackA \rightarrow send\_\tau_1 \rightarrow receive\_Hello_1 \rightarrow \\
&\quad send\_\tau_2 \rightarrow send\_\tau_3 \rightarrow receive\_\tau_4 \rightarrow receive\_\tau_6 \rightarrow \\
&\quad receive\_\tau_7 \rightarrow send\_How\_are\_you_0 \rightarrow send\_How\_are\_you_1 \rightarrow \\
&\quad receive\_\tau_9 \rightarrow receive\_\tau_{10} \rightarrow Skip \\
inviteJ &= send\_inviteJ \rightarrow receive\_inviteJ \rightarrow Skip \\
inviteA &= send\_inviteA \rightarrow receive\_inviteA \rightarrow Skip \\
ackJ &= send\_ackJ \rightarrow receive\_ackJ \rightarrow Skip \\
ackA &= send\_ackA \rightarrow receive\_ackA \rightarrow Skip \\
Hello_0 &= send\_Hello_0 \rightarrow receive\_Hello_0 \rightarrow Skip \\
Hello_1 &= send\_Hello_1 \rightarrow receive\_Hello_1 \rightarrow Skip \\
HowAreYou_0 &= send\_How\ are\ you_0 \rightarrow receive\_How\ are\ you_0 \rightarrow Skip \\
HowAreYou_1 &= send\_How\ are\ you_1 \rightarrow receive\_How\ are\ you_1 \rightarrow Skip \\
\tau_0 &= send\_\tau_0 \rightarrow receive\_\tau_0 \rightarrow Skip \sqcap \tau_1 = send\_\tau_1 \rightarrow receive\_\tau_1 \rightarrow Skip \\
\tau_2 &= send\_\tau_2 \rightarrow receive\_\tau_2 \rightarrow Skip \\
\tau_3 &= send\_\tau_3 \rightarrow receive\_\tau_3 \rightarrow Skip \\
\tau_4 &= send\_\tau_4 \rightarrow receive\_\tau_4 \rightarrow Skip \\
\tau_5 &= send\_\tau_5 \rightarrow receive\_\tau_5 \rightarrow Skip \\
\tau_6 &= send\_\tau_6 \rightarrow receive\_\tau_6 \rightarrow Skip \\
\tau_7 &= send\_\tau_7 \rightarrow receive\_\tau_7 \rightarrow Skip \\
\tau_8 &= send\_\tau_8 \rightarrow receive\_\tau_8 \rightarrow Skip \\
\tau_9 &= send\_\tau_9 \rightarrow receive\_\tau_9 \rightarrow Skip \\
\tau_{10} &= send\_\tau_{10} \rightarrow receive\_\tau_{10} \rightarrow Skip \\
\tau_{11} &= send\_\tau_{11} \rightarrow receive\_\tau_{11} \rightarrow Skip
\end{aligned}
$$

Following the strategy presented in the previous section, these processes are combined to capture the behaviour of the entire diagram, giving rise to the following process.

$$
\begin{aligned}
confCall' = (&Flavia' \,|||\, Juliano' \,|||\, Augusto') \;\underset{\Sigma}{||}\; (inviteJ \,|||\, inviteA \,|||\, ackJ \,|||\\
&ackA \,|||\, Hello_0 \,|||\, Hello_1 \,|||\, HowAreYou_0 \,|||\, HowAreYou_1 \,|||\\
&\tau_0 \,|||\, \tau_1 \,|||\, \tau_2 \,|||\, \tau_3 \,|||\, \tau_4 \,|||\, \tau_5 \,|||\, \tau_6 \,|||\, \tau_7 \,|||\, \tau_8 \,|||\, \tau_9 \,|||\, \tau_{10} \,|||\, \tau_{11})
\end{aligned}
$$

where Σ is the union of the alphabets of all processes.

Now we can focus on the relations between *confCall'* and *confCall*, which is defined in Section 4.2. The mapping M between the synchronous and the asyn-

chronous messages is defined as follows:

$$M(Hello) \mapsto \{\ \langle send\_Hello_0, receive\_Hello_0, send\_Hello_1, receive\_Hello_1\rangle,\\ \langle send\_Hello_0, send\_Hello_1, receive\_Hello_0, receive\_Hello_1\rangle,\\ \langle send\_Hello_0, send\_Hello_1, receive\_Hello_1, receive\_Hello_0\rangle\ \}$$

$$M(How\ are\ you) \mapsto \{\ \langle send\_How\ are\ you_0, receive\_How\ are\ you_0,\\ send\_How\ are\ you_1, receive\_How\ are\ you_1\rangle,\\ \langle send\_How\ are\ you_0, send\_How\ are\ you_1,\\ receive\_How\ are\ you_0, receive\_How\ are\ you_1\rangle,\\ \langle send\_How\ are\ you_0, send\_How\ are\ you_1,\\ receive\_How\ are\ you_1, receive\_How\ are\ you_0\rangle\ \}$$

Let Σ be the alphabet of *confCall* and *Async* be the alphabet of *confCall′*. The set of synchronous events $Sync = \{Hello, How\ are\ you\}$. The CSP process that represents this mapping can be automatically obtained by instantiating *Map* with the relevant parameters; its extended definition is given below (notice that we inlined $P(M(Hello))$ and $P(M(How\ are\ you))$).

$$\begin{aligned}
MapConf = {} & (?async : Async \rightarrow MapConf(Async))\\
& \Box\ Hello \rightarrow send_Hello_0 \rightarrow\\
& \quad ((receive_Hello_0 \rightarrow send_Hello_1 \rightarrow receive_Hello_1 \rightarrow MapConf)\\
& \quad \Box\ (send_Hello_1 \rightarrow ((receive_Hello_0 \rightarrow receive_Hello_1 \rightarrow MapConf)\\
& \qquad \Box\ (receive_Hello_1 \rightarrow receive_Hello_0 \rightarrow MapConf))))\\
& \Box\ How\ are\ you \rightarrow send_How\ are\ you_0 \rightarrow\\
& \quad ((receive_How\ are\ you_0 \rightarrow send_How\ are\ you_1 \rightarrow\\
& \qquad receive_How\ are\ you_1 \rightarrow MapConf)\\
& \quad \Box\ (send_How\ are\ you_1 \rightarrow ((receive_How\ are\ you_0 \rightarrow\\
& \qquad receive_How\ are\ you_1 \rightarrow MapConf)\\
& \qquad \Box\ (receive_How\ are\ you_1 \rightarrow\\
& \qquad receive_How\ are\ you_0 \rightarrow MapConf))))\\
& \Box\ (f \rightarrow Skip)
\end{aligned}$$

The process *confCallMapped* with its synchronous messages mapped to the corresponding asynchronous sequences is defined as:

$$confCallMapped = ((confCall;\ (f \rightarrow Skip)) \underset{\Sigma\cup\{f\}}{||} MapConf) \setminus (Sync \cup \{f\})$$

Let $taus = \{send\_\tau_0, receive\_\tau_0, \ldots, send\_\tau_{11}, receive\_\tau_{11}\}$. Finally we check their equivalence by submitting the following assertions to FDR:

$$confCallMapped \sqsubseteq_{FD} (confCall' \setminus taus)$$
$$(confCall' \setminus taus) \sqsubseteq_{FD} confCallMapped$$

which holds, as expected.

5 Conclusion

We extended MSC with synchronous messages, which denote an atomic and instantaneous single event in the MSC traces. Multiple instances are able to synchronise over the same message. Our extended MSC allows both synchronous and asynchronous messages to appear in a diagram. An example shows how a conference call dialogue is modelled with the extended MSC. Asynchronous messages are sent to all participants, followed by multiple synchronous messages once the conference call is established.

A transformation algorithm which takes an extended MSC diagram and generates a standard one was proposed. The algorithm implements multiple synchronous messages via the introduction of several asynchronous control messages.

In addition to that, we also defined a formal semantics for both the standard and the extended MSC in CSP. Our formalisation mapped each instance and each *message* of an MSC diagram into a CSP process. By composing them in parallel, we capture the MSC semantics of the partial order of events. All traces in CSP are precisely the same as those in MSC. We showed in CSP that an extended MSC diagram and its corresponding standard diagram (generated by the transformation algorithm) are equivalent by replacing a synchronous message by its corresponding asynchronous messages. We checked their equivalence in FDR [8].

There are several dialects of MSC. Most of them offer only asynchronous and method call messages like Object MSC [4] and Sequence Diagrams [2]. Extended Event Traces [3] and Interworkings [13] provide only synchronous messages, while Live Sequence Charts [6] and Ladkin and Leue's extension [12] offer both synchronous and asynchronous messages. However, none of them provide multiple synchrony or an algorithm which converts an MSC dialect into the standard MSC. Engels et al. propose an approach in the other direction, though [7]. They present formal definitions to verify whether a standard MSC diagram can be implemented in several communication models, including the synchronous messages of Interworkings. Most of related works have also proposed a formal semantics for MSC. However, to our knowledge, there are no semantics based on CSP, although there are semantics based on process algebra [14].

As already mentioned, the motivation for this work resulted from our effort to use the PowerToolKit (PTK) [1]: a tool developed by Motorola Labs which generates executable test scripts by reading and analysing MSC diagrams together with associated message specifications. The tool can generate multiple test scripts which test the behaviour of a standard MSC. With extended MSC, we are now able to properly model mobile device applications with multi-synchronisation and to investigate a development process using our extended notation and PTK. We also plan to mechanise the generation of standard MSC diagrams from extended ones and to provide a general proof of correctness of the translation.

References

[1] Baker, P., P. Bristow, C. Jervis, D. J. King and B. Mitchell, *Automatic generation of conformance tests from message sequence charts*, in: E. Sherratt, editor, *Telecommunications and beyond: The Broader Applicability of SDL and MSC*, Lecture Notes in Computer Science **2599** (2002), pp. 170–198.

[2] Booch, G., J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide," Addison Wesley, Reading, Massachusetts, 1998.

[3] Broy, M., C. Hofmann, I. Kruger and M. Schmidt, *Using extended event traces to describe communication in software architectures*, in: *Proceedings of the Asia-Pacific Software Engineering Conference and International Computer Science Conference* (1997).

[4] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, "Pattern-Oriented Software Architecture. Volume I: A System of Patterns," Wiley, 1996.

[5] Cabral, G. and A. Sampaio, *Formal specification generation from requirement documents*, Electronic Notes in Theoretical Computer Science **195** (2008), pp. 171–188.
URL http://dx.doi.org/10.1016/j.entcs.2007.08.032

[6] Damm, W. and D. Harel, *LSCs: Breathing life into Message Sequence Charts*, Formal Methods in System Design **19** (2001), pp. 45–80.

[7] Engels, A. G., S. Mauw and M. A. Reniers, *A hierarchy of communication models for Message Sequence Charts*, Science of Computer Programming **44** (2002), pp. 253–292.

[8] "Failures-Divergence Refinement: FDR2 User Manual," (2005).

[9] Grosu, R., I. Kruger and T. Stauner, *Hybrid sequence charts*, in: *ISORC'00* (2000), p. 104.

[10] Isobe, Y. and M. Roggenbach, *A generic theorem prover of CSP refinement*, in: N. Halbwachs and L. D. Zuck, editors, *TACAS'05*, Lecture Notes in Computer Science **3440** (2005), pp. 108–123.

[11] ITU-TS, *ITU-TS recommendation Z.120: Message sequence chart 1999 (MSC99)*, Technical report, ITU-TS, Geneva (1999).

[12] Ladkin, P. B. and S. Leue, *What do Message Sequence Charts mean?*, in: *FORTE'93, Proceedings of the IFIP TC6/WG6.1.* (1994), pp. 301–316.

[13] Mauw, S. and M. Reniers, *A process algebra for Interworkings*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra* (2001), pp. 1269–1327.

[14] Mauw, S. and M. A. Reniers, *An algebraic semantics of basic Message Sequence Charts*, The Computer Journal **37** (1994), pp. 269–278.

[15] Melham, T. F., *Abstraction mechanisms for hardware verification*, in: G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis* (1988), pp. 129–157.
URL http://citeseer.ist.psu.edu/melham87abstraction.html

[16] Roscoe, A. W., "Theory and Practice of Concurrency," Prentice-Hall, 1998.

[17] Sengupta, B. and R. Cleaveland, *Triggered Message Sequence Charts*, IEEE Transactions on Software Engineering **32** (2006), pp. 587–607.