Compositional Message Sequence Charts

Elsa L. Gunter
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Anca Muscholl
Universite Paris 7
2, place Jussieu
75251 Paris Cedex 05

Doron Peled
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

October 17, 2000

Abstract

Message sequence charts (MSCs) is a standard notation for describing the interaction between communicating objects. It is popular among the designers of communication protocols. MSCs enjoy both a visual and a textual representation. High level MSCs (HMSCs) allow specifying infinite scenarios and different choices. In this paper we demonstrate a weakness of HMSCs, which disallows one to model certain interactions. Specifically, an HMSC consists of a graph, where each node is a finite MSC with *matched* send and receive events, and vice versa. We will show, by means of an example, that some simple finite state and simple communication protocol cannot be represented using HMSCs. We then propose an extension to the MSC standard, which allows HMSC nodes to include unmatched messages. The corresponding graph notation will be called HCMSC, which stands for High level *Compositional* Message Sequence Charts. With the extended framework, we provide an algorithm for automatically constructing an MSC representation for finite state asynchronous message passing protocols.

1 Introduction

Visual notations are useful in the design of large and complicated systems. They allow a more intuitive understanding of the behavior of the system and the relation between its components. They often allow abstracting away parts of the system that are less relevant for a particular view. Message sequence charts are among the most frequently used formalism for designing communication protocols. Recently, they have been also used in the development of object oriented systems [17]. In the recent years, we observe the development of a growing number of tools and algorithms for the manipulation of MSC based designs [2, 13, 3, 12, 1, 7].

The standard visual and textual notation [10] by ITU allows representing a single execution scenario, as well as a collection of scenarios, including choices and repetition. This is achieved by a notation called HMSC (High Level Message Sequence Chart),

which consists of a graph, where each node contains a single MSC. The system behavior can follow the paths on that graph, starting from some initial node. In this paper we show, by means of an example, a limitation of HMSCs. This limitation stems from the constraint that each HMSC node must have only *matched* send and receive events. We show examples where one cannot break a possibly infinite computation of a finite state system into finitly many nodes with matched communication events. (A finite execution can always be represented as a single node.) We demonstrate that such undecomposable behaviors are not merely a theoretical result, but can represent the execution of real protocols.

To circumvent the problem, we suggest an extension to the MSC standard. This extension allows specifying MSCs with unmatched sends and receives. The semantics of the new construct prescribes how to combine such MSC nodes together. We use the extended notation to suggest an algorithm for the automatic generation of MSC representations for finite state systems.

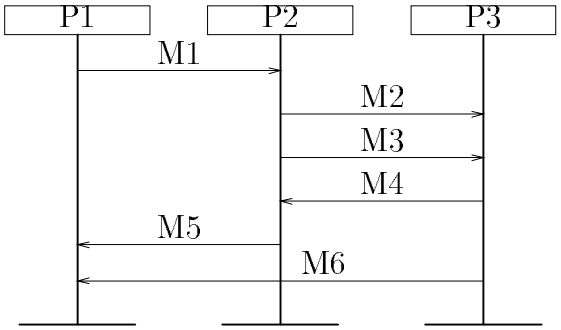
This defficiency of MSCs was also recognized in [11]. The solution suggested there is a different extension to HMSCs. According to this extension, one can use parallel components of MSCs, and allow intercommunication between them, using a mechanism called 'gates'. Our solution differs from that of [11], as we study the effect of allowing communication between sequentially composed MSCs. That is, a communication that starts in one MSC and ends in a subsequent one. The problem of checking whether a finite-state protocol can be translated into an HMSC has been considered in [8, 14]. In the first paper it is shown that this question is decidable, whereas the second paper considers the complexity issue in different settings.

2 Preliminaries

Each MSC describes a scenario where some processes communicate with each other. Such a scenario includes a description of the messages sent, messages received, the local events, and the ordering between them. In the visual description of MSCs, each process is represented as a vertical line, while a message is represented by a horizontal or slanted arrow from the sending process to the receiving one, as in Figure 1. The corresponding ITU Z120 textual representation of the MSC appears on the right side part of Figure 1.

Definition 2.1 *An MSC* M *is a tuple* $\langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m\rangle$.

- V *is a finite set of events,*
- $< \subseteq V \times V$ *is an acyclic relation,*
- $\mathcal{P}$ *is a set of processes,*
- $\mathcal{N}$ *is a set of* message names*,*
- $L : V \to \mathcal{P}$ *is a mapping that associates each event with a process,*
- $T : V \to \{\mathrm{s}, \mathrm{r}, \mathrm{l}\}$ *is a mapping that describes each event as send, receive or local, respectively.*
- $N : V \to \mathcal{N}$ *maps every event to a name.*




```
msc MSC;
inst P1: process Root,
     P2: process Root,
     P3: process Root;
    instance P1;
        out M1 to P2;
        in M5 from P2;
        in M6 from P3;
    endinstance;
    instance P2;
        in M1 from P1;
        out M2 to P3;
        out M3 to P3;
        in M4 from P3;
        out M5 to P1;
    endinstance;
    instance P3;
        in M2 from P2;
        in M3 from P2;
        out M4 to P2;
        out M6 to P1;
    endinstance;
endmsc;
```

Figure 1: Visual and textual representation of an MSC

- $m \subseteq V \times V$ *is a partial function called* matching *that pairs up send and receive events. Each send is paired up with exactly one receive and vice versa. Events* v_1 *and* v_2 *can be paired up with each other, only if* $N(v_1) = N(v_2)$.

A message consists of a pair of matching send and receive events. For two events e *and* f*, we have* $e < f$ *if and only if one of the following holds:*

- e *and* f *are a matching send and receive events, respectively.*
- e *and* f *belong to the same process* P*, with* e *appearing before* f *on the process line.*

We assume fifo (first in first out) message passing, i.e.,

$$(T(e_1) = T(e_2) = \mathrm{s} \wedge T(f_1) = T(f_2) = \mathrm{r} \wedge m(e_1, f_1) \wedge m(e_2, f_2) \wedge \\ L(e_1) = L(e_2) \wedge L(f_1) = L(f_2) \wedge e_1 < e_2) \rightarrow f_1 < f_2$$

Denote by $e \longrightarrow f$ the fact that $e < f$ and either e and f are a matching send and receive events, or e and f belong to the same process and there is no event between e and f on some process line. That is, e immediately precedes f. The transitive closure of the relation $<$ is a partial order called the *visual ordering* of events and it is obtained from the syntactical representation of the chart (e.g. represented according to the standard syntax ITU-Z120 [10]).

A *type* is a triple (i, j, m), including two processes P_i and P_j, and a message name m. Each *send* or *receive* event has a type, according to the origin and destination of the message, and the label on the message. Matching events have the same type. A *linearization* of an MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m \rangle$ is a total order on V, which extends the relation $(V, <)$.

Tools such as Msc [2] support representing message sequence charts. They allow both a visual description of the MSCs, and a textual representation [10]. Thus, an MSC can be obtained either by drawing it using a visual interface, or by typing in its

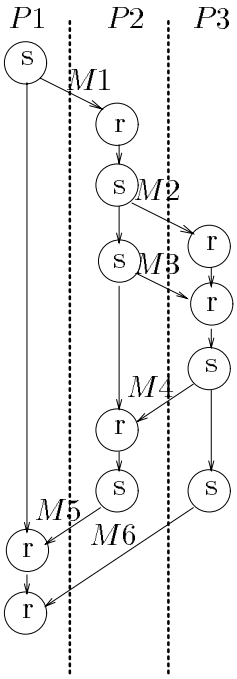


Figure 2: The partial order between the events of the MSC in Figure 1

textual description. The two representations are easily translatable from one another. This approach has the advantage that the visual representation of an MSC is related to some formal representation that can be easily manipulated algorithmically.

The partial order between the send and receive events of Figure 1 is shown in Figure 2. In this figure, only the 'immediately precedes' order is shown. The MSC in Figure 1 describes an interaction between three processes, $P1$, $P2$ and $P3$. Process $P1$ sends the message $M1$ to $P2$. After receiving that message, process $P2$ sends two messages, $M2$ and $M3$ to $P3$. After receiving $M3$, process $P3$ sends the message $M4$ back to $P2$ and later also sends the message $M6$ to $P1$. Process $P2$, after receiving $M4$, sends $M5$ to $P1$. The message $M5$ is received by process $P1$ before the message $M6$. The send events of the two messages, $M5$ and $M6$, are unordered.

Definition 2.2 *The* concatenation *of two MSCs* $M_1 = \langle V_1, <_1, \mathcal{P}, \mathcal{N}_1, L_1, T_1, N_1, m_1 \rangle$ *and* $M_2 = \langle V_2, <_2, \mathcal{P}, \mathcal{N}_2, L_2, T_2, N_2, m_2 \rangle$ *over the same set of processes* $\mathcal{P}$ *and disjoint sets of events* $V_1 \cap V_2 = \emptyset$*, denoted* $M_1\, M_2$*, is* $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2 \rangle$*, where*

$$< = <_1 \cup <_2 \cup \{(p, q) \mid L_1(p) = L_2(q) \wedge p \in V_1 \wedge q \in V_2\}$$

That is, the events of M_1 precede the events of M_2 for each process, respectively. If $M = M_1\, M_2$, we say that M_1 is a *prefix* of M. Notice that there is no synchronization of the different processes when moving from one node to the other. Hence, it is possible that one process is still involved in some actions of one node, while another process has advanced to a different node.

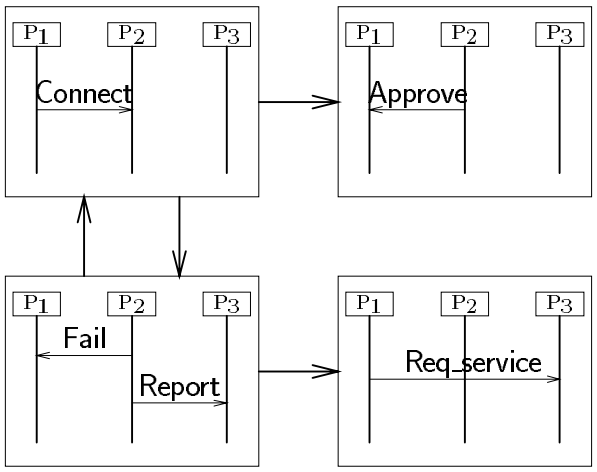


Figure 3: An HMSC graph

Since a communication system usually includes many (or even infinitely many) such scenarios, a high level description is needed for combining them together. The standard description consists of a graph called HMSC (high level MSC), where each node contains one MSC as in Figure 3. Each maximal path in this graph (i.e., a path that is either infinite or ends with a node without outgoing edges), starting from a designated initial state, corresponds to a single *execution* or *scenario*. Such an execution can be used to denote the communication structure of a typical (aka 'sunny day') or an exceptional (aka 'rainy day') behavior of a system, or a counterexample found during testing or model checking.

Definition 2.3 *An HMSC N is a triple $\langle \mathcal{S}, \tau, s_0, c \rangle$ where $\mathcal{S}$ is a finite set of states, each labeled by some MSC over the same set of processes, and with sets of events disjoint from one another. The mapping c associates the state s with an MSC $c(s)$. $\tau \subseteq \mathcal{S} \times \mathcal{S}$ is the* edge relation *and the* initial *state is $s_0 \in \mathcal{S}$. An* execution *of N is an MSC $\xi = c(s_0)\, c(s_1)\, c(s_2) \ldots$ associated with a path of N that starts with the initial state s_0 and either ends with a state without outgoing edges, or is infinite.*

Figure 3 shows an example of an MSC graph where the node in the upper left corner is the starting node. Note that the executions of this system are either finite or infinite. In Figure 3, process $P2$ may send its Report message after process $P1$ has progressed into the next node and has sent its Req_service message.

3 MSC Decomposition

The HMSC model combines the visual notation of message sequence charts with the ability to describe repetitions and alternative computations. In this section we will show that this, seemingly powerful model, cannot describe some basic finite state communication protocols. The main problem lies within the requirement that the *send* and *receive* events in each node must be matched.

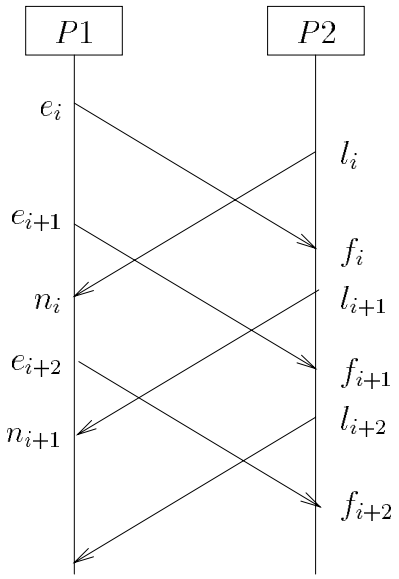


Figure 4: A prefix of an MSC execution that cannot be decomposed

We want to exemplify that there are finite state protocols that do not allow a finite HMSC representation. To do that, we show an infinite executions ξ of a finite state protocols with the following property: There is no way to write ξ as an infinite concatenation of finite MSCs. Given the above property, it is not possible to construct an HMSC such that ξ would correspond to a traversal of one of the HMSC paths. Thus, we cannot represent such a system using HMSCs.

As an example, consider the MSC whose prefix appears in Figure 4. To show that it cannot be decomposed, we will show an infinite sequence of events that cannot be separated to different MSC. We will start with the *send* event e_i and *receive* event f_i. Obviously, because of the compulsory matching in HMSCs, they must belong to the same MSC node. We have a *send* event l_i preceding f_i, on the same process line, while its corresponding *receive* event n_i succeeds the *send* e_i matching with f_i. Thus, the events l_i and n_i must be in the same node with e_i and f_i. For the same reason, we have that e_{i+1} and f_{i+1} must belong to the same node with l_i, and n_i and so forth.

While the repeated crossing of message edges seems to be untypical for MSCs, the above behavior ξ describes a possible execution of an actual protocol [16], where messages and acknowledgements are being sent between two processes, with (bounded) buffering.

4 Compositional MSCs

In order to represent communication protocols, whose description could only be approximated using standard MSCs, we suggest an extension of the MSC standard. Intuitively, a *compositional MSC*, or CMSC, may include *send* events that are not matched by corresponding *receive* events and vice versa. An unmatched send event

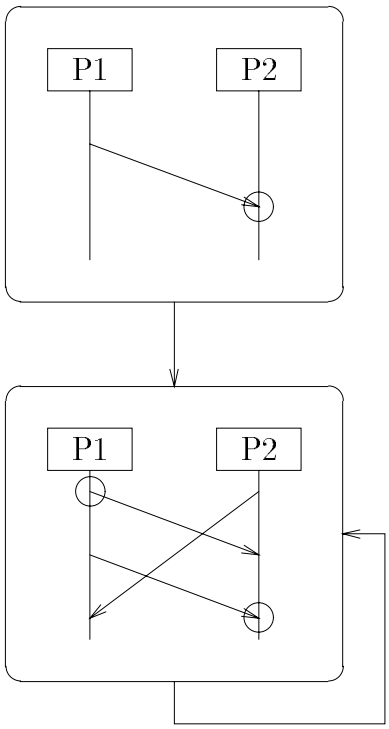


Figure 5: A decomposition of the execution in Figure 4

may be matched in future HCMSC nodes (on some path). Similarly, an unmatched receive event may be matched in previous HCMSC nodes. The definition of a CMSC is hence similar to an MSC, except that unmatched *send* and *receive* messages are allowed. (For its similarity to Definition 2.1, we will omit repeating the formal definition with the corresponding change.)

We denote an unmatched *send* by a message arrow, where the *receive* end (the target of the arrow) appears within an empty circle. Similarly, an unmatched *receive* is denoted by an arrow where the *send* part (the source of the arrow) appears within a circle. Events occurring in an CMSC are also denoted as *real* events, whereas their missing counterparts are called *virtual* events. In an CMSC arrows where both the send and the receive are virtual events are forbidden. Moreover, we also disallow a virtual receive event to be followed by a real receive event of the same type in the same CMSC node. Similarly, we disallow a virtual send event to be preceded by a real send event of the same type in the same CMSC node. In Figure 5, we can see an HCMSC that represents the execution that is approximated in Figure 4.

Next we define a special case of CMSCs:

Definition 4.1 *A* left-closed *CMSC is a CMSC that does not contain unmatched receive events.*

The semantics of a left-closed CMSC is defined in exactly the same way as in the semantics definition. The only difference is that only *send* events are allowed to be

unmatched. We define the concatenation between a left-closed CMSC and a CMSC as follows:

Definition 4.2 *The* concatenation *of two CMSCs* $M_1 = \langle V_1, <_1, \mathcal{P}, \mathcal{N}, L_1, T_1, N_1, m_1\rangle$ *and* $M_2 = \langle V_2, <_2, \mathcal{P}, \mathcal{N}, L_2, T_2, N_2, m_2\rangle$, *denoted* $M_1 M_2$, *is defined when the following conditions hold:*

- M_1 *is left-closed.*
- *For any type* t, *the number of unmatched receive events of type* t *in* M_2 *is at most equal to the number of unmatched send events of type* t *in* M_1.
- $V_1 \cap V_2 = \emptyset$.

Define a matching function m *that pairs up unmatched send events of* M_1 *with unmatched receive events of* M_2 *according to their order on their process lines. That is, the* i*th unmatched send of some type in* M_1 *is paired up with the* i*th unmatched receive event of the same type in* M_2. *We require that each two unmatched events will be mapped to the same name. Notice that the function* m *is uniquely defined.*

The concatenation is then defined as follows: $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2 \cup m\rangle$, *where*

$$< = <_1 \cup <_2 \cup \{(p,q) \mid L(p) = L(q) \wedge p \in V_1 \wedge q \in V_2\} \cup$$

$$\{(p,q) \mid (p,q) \in m \wedge T(p) = \mathrm{s} \wedge T(q) = \mathrm{r}\}.$$

It is easy to see that a concatenation always results in a left-closed CMSC. Moreover, if M_1 and M_2 both satisfy the fifo restriction, then $M_1 M_2$ also does. HCMSCs are defined using CMSC nodes, in an analogous way to Definition 2.3.

5 Undecidability

Extending the MSC standard allows representing the execution of a bigger class of protocols than what is allowed by the ITU standard. However, unsurprisingly, with the added expressiveness we loose some of the power of analyzing such systems.

Unlike simple HMSC, one cannot decide even the trivial properties of this HCMSC, e.g., whether a particular message can be sent or received in at least one computation. The undecidability reduction will be from Post Correspondence Problem (PCP). An instance of PCP is a set of pairs of words

$$C = \{(v_1, w_1), (v_2, w_2), \ldots, (v_m, w_m)\}$$

over some mutual alphabet Σ. We want to find out if we can find a nonempty sequence of indexes $i_1, i_2, \ldots i_n$ such that $v_{i_1} v_{i_2} \ldots v_{i_n} = w_{i_1} w_{i_2} \ldots w_{i_n}$ and $i_n = 1$. Using a suitable encoding we may assume that whenever $w_{i_1} w_{i_2} \ldots w_{i_{n-1}} w_1$ is a prefix of $v_{i_1} v_{i_2} \ldots v_{i_{n-1}} v_1$, then these two words are equal.

We will construct a HCMSC with five processes P_1 to P_5, and with CMSC nodes $E_1, E_2, \ldots, E_m, E'_1, E'_2, \ldots, E'_m, F, F'$.

- Messages from P_1 to P_2 correspond to the letters of Σ. Each CMSC E_i contains a sequence of unmatched *send* events from P_1 to P_2, representing the sequence of messages of v_i. Each CMSC E_i' contains a sequence of unmatched *receive* events from P_1 to P_2, representing the sequence of messages of w_i.
- Messages from P_3 to P_4 correspond to the index of the PCP word being sent. Each CMSC E_i contains also a single unmatched *send* from P_3 to P_4 representing the current index i. Each CMSC E_i' contains the corresponding unmatched *receive* event.

The HMSC graph starts at some initial node F, which contains only one unmatched *send* from P_1 to P_5. Then one can repeatedly take nodes of the form E_i, followed by nodes of the form E_i', any number of times. From the node labeled by E_1' we have a transition to a sink node F' (i.e., a node with no outgoing edges) which contains a message from P_2, then a message from P_4 to P_5 and finally an unmatched *receive* (matching the *send* from node F) from P_1 to P_5. Notice that whenever the message from P_1 to P_5 is received, the corresponding word $v_{i_1} \dots v_{i_n}$ is a prefix of $w_{i_1} \dots w_{i_n}$ and $i_n = 1$. By the assumption on PCP solutions we get equality, hence a solution. Thus, the message from P_1 to P_5 is received if and only if iff there is a nonempty solution to the corresponding PCP instance.

The undecidability result above is related to the fact that HCMSCs (as HMSCs) can describe infinite state systems. When restricting to finite state systems (in particular, the message queues are finite), we regain decidability.

6 Balanced HCMSCs

A natural restriction for HCMSC is to require that each maximal execution defines a left-closed CMSC. Note that we want to allow executions with pending send events. For example, the HCMSC of Figure 5 is such that every finite execution is a left-closed CMSC with pending sends. However, the (unique) infinite execution corresponds to an (infinite) MSC.

Definition 6.1 *An HCMSC is* balanced *if the execution of every finite path starting with the initial state is a left-closed CMSC.*

We show in the following how to test whether a HCMSC is balanced. Assume that $M = c(s_0)c(s_1)\cdots c(s_n)$ is a CMSC associated with a finite path $\chi = s_0, s_1, \dots, s_n$ of the HCMSC N which starts in the initial state s_0. Let t be a type, then the *t-deficit* $D_t(\chi)$ of χ is the difference between the number of send events and the number of receive events of type t in χ. A necessary condition for N to be balanced is that $D_t(\chi) \geq 0$ for every loop χ and every type t. Note also that a balanced HCMSC N has bounded message queues if and only if $D_t(\chi) = 0$ for every loop χ in N and every type t. More generally, an HCMSC $N = \langle \mathcal{S}, \tau, s_0, c \rangle$ is balanced if and only if every node which is accessible from the initial node satisfies the following condition: Assume that the ith instance line contains x unmatched receives of type t. Then $D_t(\chi) \geq x$ for all paths χ from s_0 to s' with $(s', s) \in \tau$.

We describe now an algorithm for checking that an HCMSC N is balanced. We define for each state s and each type t the t-deficit $d_t(s)$ of s as the difference between unmatched sends of type t and unmatched receives of type t in s. We can view N as

a weighted directed graph $G_t(N) = \langle \mathcal{S}, \tau, \gamma \rangle$, with edges weighted by $\gamma(s', s) = d_t(s')$. Then all we have to do is to check that $G_t(N)$ has no cycle with negative weight and that for all states s, the minimal weight d of a path from s_0 to s' is such that $d \geq x$, where x is the number of unmatched receives of type t in s and s' runs over all predecessors of s, $(s', s) \in \tau$. We can apply either a dynamic programming algorithm, which actually computes the shortest paths between all pairs of nodes in time $O(|\mathcal{S}|^3)$. Alternatively, we can use the Bellman-Ford algorithm, [4]. This algorithm computes in time $O(|\mathcal{S}||\tau|)$ all shortest paths from a given source in a graph G with negative weights, provided that G contains no negative cycle (detecting such a cycle, if one exists). Doing this for all graphs $G_t(N)$ yields an $O(|\mathcal{P}|^2|\mathcal{S}||\tau|)$ algorithm for checking whether N is balanced.

We conclude this section with a remark on the regularity of the set of executions of an HCMSC. Note first that bounded message queues are not a guarantee that the set of linearizations of executions in an HMSC or an HCMSC is regular. In the case of HMSCs a syntactic restriction which is sufficient for regularity has been proposed in [3, 12]. This condition states that the communication graph of every loop in the HMSC must be strongly connected. The *communication graph* of an MSC M is a directed graph with vertex set consisting of all processes which occur in M. An edge exists from process P to process Q if there is a message from P to Q in M. The communication graph of a path π in an HMSC is the communication graph of the MSC associated with π. We show in the following a similar syntactic condition for HCMSC which is sufficient for obtaining a regular set of linearizations, provided that the message queues are bounded. For this we define the communication graph of an CMSC M as follows. As before, vertices are those processes with events occurring in M. We have an edge from P to Q if there is a (matched or unmatched) send event on P with corresponding receive end on Q. As before, we require that the communication graph of any loop in the HCMSC is strongly connected.

Proposition 6.2 *Let N be an HCMSC with bounded message queues, i.e., the deficit of every execution χ of N is such that $D_t(\chi) \leq k$, for some constant k depending on N and for any type t. Let us assume that the communication graph of any loop in N is strongly connected. Then the set of linearizations of N is regular.*

The proposition above can be shown using the same ideas as for HMSCs. The difference here concerns the number of states of the automaton accepting the linearizations of N. We can show that for any linearization of an execution $c(s_0)c(s_1)\cdots c(s_m)$ of N it suffices to store a polynomial number of prefixes of CMSCs $c(s_i)$. We use here the fact that the deficit $D_t(\chi)$ is at most equal to the size of the HCMSC N.

7 An HCMSC Representation for Finite State Systems

The HCMSC extension suggested in this paper broadens the scope of HMSCs and allows us to capture many more protocols. We present now an automatic translation from finite state systems based on asynchronous message passing to (balanced) HCMSC.

We are given a finite state space $G = (S, S_0, E, \Sigma)$, with states S, initial states $S_0 \subseteq S$, edges $E \subseteq S \times \Sigma \times S$, over a set of actions Σ. The actions in Σ are *send*,

receive and *local* actions. The states in S contain information about the system, including the contents of the various interprocess message queues.

We start with a trivial translation, which establishes the theoretic possibility of performing such a translation for a class of finite state systems with asynchronous message passing. We later proceed to suggest a more informative translation. The trivial translation is performed by constructing the dual graph $H = (N, N_0, F)$ of G as follows:

- The *nodes* N of H correspond to the *edges* of G. That is, $N = E$.
- The *initial nodes* $N_0 \subseteq N$ of H correspond to the edges of G that exit from an initial state of S_0. That is,

$$N_0 = \{s \longrightarrow s' \mid s \longrightarrow s' \in E \wedge s \in S_0\}$$

- The edges F of H correspond to pairs of edges of G such that the target of the first edge is the source of the second. That is,

$$F = \{e_1 \longrightarrow e_2 \mid \exists s_1, s_2, s\, e_1 = s_1 \longrightarrow s,\ s \longrightarrow s_2 \in E\}$$

The graph H is an HCMSC, where each node in N is a trivial MSC, representing a single (unmatched) message passing event of *send* or *receive*, or a *local* event.

The above trivial construction does not provide any new insight, since the HCMSC graph follows closely the state space and each CMSC block includes a single local or unmatched event. We thus look into a translation that would often construct more reasonable HCMSCs. The translation aims at optimizing the following goals:

1. Minimize the number of unmatched messages appearing in the individual CMSC blocks, if possible obtaining an HCMSC without any unmatched messages (however, recall from Section 3 that this is not always an attainable goal).
2. Present relatively long scenarios with the CMSCs, in order to obtain an intuitive understanding of the interprocess interaction.
3. Minimize the number of individual CMSC blocks, so that the HCMSC would not become too big (there can be much more sequences than there are states).

Notice that the second and third goal may contradict each other in some systems. The above 'trivial' translation gives a rather reasonable solution to the third goal, while providing unacceptable solution for the second goal. Notice further that the size of an HCMSC graph can easily get prohibitively large. Thus, in practice, the HCMSC construction algorithm should be applied only to small parts of communication protocols, rather than to complete protocols.

It is easy to see that different execution paths in the state space may correspond to a single CMSC. For example, consider an execution path in which we have a *send* from $P1$ to $P2$, then the matching *receive*, then another *send* of the same type, and finally another matching *receive*. Consider now another execution path, in which we have first the two *send* transitions, and then the two *receive* transitions. These two paths obviously correspond to the same MSC. The *partial order reduction* algorithms were constructed for this particular reason. The *sleep set* method of Godefroid, adapted to our case, is in particular appropriate.

The Algorithm

Definition 7.1 *For a letter $e \in \Sigma$ (an event), define the set of events dep(e) that include exactly events f such that either e and f are from the same process, or e and f are a matching pair.*

Notice that this definition is tailored for a message passing communication system and need to be adapted when using other kinds of concurrency (e.g., with shared variables).

Let '$\prec$' be a total order over the events in Σ satisfying that all the *receive* events precede the *send* events. Denote by $en(s)$ the set of transitions that are enabled at a state s.

1. Make a first guess of a set of nodes such that every cycle must pass through one of these nodes. One possibility is to set $Z \subseteq S$ to include every node in which all the queues are empty. Another possibility is to start with the single set that includes the initial node. One heuristics is to perform simple DFS on the state space and include in Z every node in the target of a back edge. Notice that this is not optimal (finding a minimal set of such nodes is an NP-complete problem). The nodes in Z are new cutpoints for the finite state space in the sense that every cycle must pass at least one of these points. Thus, the paths from Z to Z contain no cycles.

2. Start a *minimized DFS* from nodes in Z or at an initial state. The search stops at nodes in Z (after progressing at least one step) or to a terminating node. The minimization algorithm, related to Godefroid's sleep set algorithm [5], and to the variant of that algorithm presented in [15] is shown in Figure 6. This version allows removing nodes that have an empty number of successors under the reduction.[1]

3. Construct CMSCs for the paths from the nodes in Z according to the paths generated during the reduced DFS of the previous step. Since the number of paths can be enormous, one can split the reduced graph further, e.g., at points that have a relatively large number of incoming or outgoing edges. In this way, we generate shorter paths, but possibly more of them. The matching algorithm in Section 7.1 can be used to match corresponding *send* and *receive* events in the same CMSC.

4. Connect the separate CMSCs in the following way: If one CMSC ends at some state $s \in Z$ and another CMSC starts with that state, make an edge from the former to the latter.

Properties of the Algorithm

Definition 7.2 *Define the relation '$\longrightarrow$' between strings over Σ by $\sigma \longrightarrow \rho$ if $\sigma = v\,e\,f\,w$ and $\rho = v\,f\,e\,w$, where v, w are sequences of transitions and f, e are individual transitions and $f \notin dep(e)$. Let $\stackrel{*}{\longrightarrow}$ be the transitive and reflexive closure of $\longrightarrow$.*

Definition 7.3 *Define the relation '$\sqsubseteq$' between strings over Σ such that $v \sqsubseteq w$ when*

[1] Another change from the original algorithm is that the new nodes are pairs of a state and a sleep set, and two states that are paired with different sleep sets are considered different nodes.

```
function expand_node(s, sleep);
local explored, working_set, new_sleep, fixed;
    explored := ∅;
    fixed := false;
    if en(s) = ∅ then return true fi;
    working_set := en(s) \ sleep ;
    while working_set ≠ ∅ do
      α := biggest action in working_set according to '≺';
      working_set := working_set \ {α};
      s' := α(s);
      new_sleep := (sleep ∪ explored ) \ dep(α);
      explored := explored ∪ {α};
      if s' ∈ Z orelse s' is terminal orelse exists_node(s', new_sleep)
           orelse expand_node(s', new_sleep) then
        fixed := true;
        create_edge((s, sleep), α, (s', new_sleep)) fi;
      fi
    end while;
    if fixed then store_node_in_hash(s, sleep);
    return fixed;
end expand_node.
```

Figure 6: A reduced state space generation algorithm

- *v is smaller than w according to the alphabetical order based on '$\prec$'.*
- *$w \xrightarrow{*} w'$, and v is a prefix of w'.*

Lemma 7.4 *If $v \sqsubseteq w$, then an CMSC with a linearization v is a prefix of an CMSC with a linearization w.*

Sketch of proof. We can show by induction on the number of permutations done in '$\xrightarrow{*}$' that the transitions of each process in v are a prefix of the transitions of the same process in w. This follows from definition 7.1. ∎

Lemma 7.5 *If $v \sqsubseteq w$, v is not a prefix of w, and w is generated during the reduced DFS, then v is not generated by the algorithm.*

Sketch of proof. Take the longest common prefix u of v and w (u can be empty). Let b be the first letter after u in w, and a the first letter after u in v. Then from the definition of the relation '$\sqsubseteq$', we have that $a \prec b$, $a \notin dep(b)$, and b appears in v after u, following some sequence of transitions u' that are not included in $dep(b)$. According to the algorithm, during the DFS, $u\,b$ is reached before $u\,a$. When the search backtracks from u, it has b in its sleep set, since $a \notin dep(b)$. If the search reaches $u\,u'$, then b is still in the sleep set, since b is independent of all the events in u'. Because of this, $u\,u'\,b$ is not generated. ∎

Lemma 7.6 *If v is not generated during the search, then there is some w such that $v \sqsubset w$, and w is generated.*

Sketch of proof. First, observe that '$\sqsubseteq$' is a reflexive and transitive relation. The proof is by an induction on the order '$\sqsubseteq$'. Suppose that v is not generated. This is because $v = u\,u'\,a\,w$ for some sequences u, u' and w, and a transition a, and a was in the sleep set paired with the state obtained after the reduced DFS has searched u. Furthermore, the transition a was taken after u, and is independent of the transitions in u' and is bigger according to '$\prec$' than the first letter in u'. Thus, we have that $v \sqsubseteq u\,a\,u'\,w$. Then, by the induction hypothesis, either $u\,a\,u'\,w$ is expanded, or a string w' such that $u\,a\,u'\,w \sqsubseteq w'$ is expanded. But by the transitivity of $\sqsubseteq$, we have the result. ∎

7.1 The Matching Algorithm

For each CMSC node M constructed by the above algorithm and for every possible deficit d for messages of type t on paths to M we create a HCMSC node M_d with the same events as M and match the events as follows.

1. Mark the first d *receive* events of type t in M_d as 'unmatched' (there may be fewer than d such messages).
2. Of the remaining *send* and *receive* events of type t, pair the ith *send* with the ith *receive*.
3. If there are *send* events of type t that are unpaired in the previous step, mark them 'unmatched'.

8 Conclusion and Implementation

HMSCs are a useful and standard notation for describing executions of communication protocols. We showed that the requirement of pairing up *send* and *receive* events in each MSC node prohibits the representation of a simple finite state protocol. We presented an extension of the HMSC notation, which we call HCMSC. This notation circumvents this problem. With the extension, we presented an algorithm for automatically generating the HCMSC structure for finite state communication protocols. We have implemented this algorithm as an extension of the PET system [6]. The implementation is written using 800 lines of SML/NJ code, and in addition exploits the C code of the MSC/POGA [9] system for generating the HCMSC visual structure.

Acknowledgement. We would like to thank Mihalis Yannakakis, who suggested the counterexample in Figure 4, which is simpler than our original counterexample.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd International Conference on Software Engineering, Limerick (Ireland)*, pages 304–313, ACM, 2000.

[2] R. Alur, G. H. Holzmann, and D. A. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory CONCUR'99, Eindhoven (The Netherlands)*, LNCS 1664, 1999. Springer.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, Massachusetts, 1999.

[5] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[6] E. Gunter and D. Peled. Path exploration tool. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Amsterdam, The Netherlands*, LNCS 1579, pages 405–419, 1999. Springer.

[7] L. Hélouët and P. Le Maigat. Decomposition of Message Sequence Charts. In *Proc. of the 2nd Workshop on SDL and MSC (SAM2000)*, pp. 46–60, 2000.

[8] J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P. Thiagarajan. On message sequence graphs and finitely generated regular msc languages. In *Proceedings of the 27th (ICALP'00), Geneva (Switzerland), 2000*, LNCS 1853, pages 675-686, 2000. Springer.

[9] G. Holzmann. Formal methods for early fault detection. In *Proc. of the 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, Uppsala (Sweden)*, 1996.

[10] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.

[11] S. Mauw and M. Reniers. High-level message sequence charts. In *SDL'97: Time for Testing - SDL, MSC and Trends. Proceedings of the 8th SDL Forum, Evry (France) 1997*, pages 291–306, 1997.

[12] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *Proceedings of the 24th Symposium on Mathematical Foundations of Computer Science (MFCS'99), Szklarska Poreba (Poland) 1999*, LNCS 1672, pages 81–91, 1999. Springer.

[13] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of the 1st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'98), Lisbon, Portugal, 1998*, LNCS 1378, pages 226–242, 1998. Springer.

[14] A. Muscholl and D. Peled. High-level message sequence charts and finite-state communication protocols. Submitted, 2000.

[15] D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Proc. of Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece*, LNCS 697, pages 409–423, 1993. Springer.

[16] A. Tanenbaum, Computer Networks, Prentice Hall, 1988.

[17] M. Fowler, K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.