Compositional Message Sequence Charts

Elsa L. Gunter[1], Anca Muscholl[2], and Doron A. Peled[1]

[1] Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974, USA
[2] Université Paris 7
2, place Jussieu, case 7014
75251 Paris Cedex 05, France

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. Specifically, an HMSC consists of a graph, where each node is a finite MSC with *matched* send and receive events, and vice versa. In this paper we demonstrate a weakness of HMSCs, which disallows one to model certain interactions. 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, e.g. in UML. In the recent years, we observe the development of a growing number of tools and algorithms for the manipulation of MSC based designs [1–3, 7, 11, 12].

The standard visual and textual notation [9] 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 MSC node in an HMSC must have only *matched* send and receive events, i.e., each MSC must be labeled by message arrows. We show examples where one cannot break a possibly infinite computation of a finite state system into finitely 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.

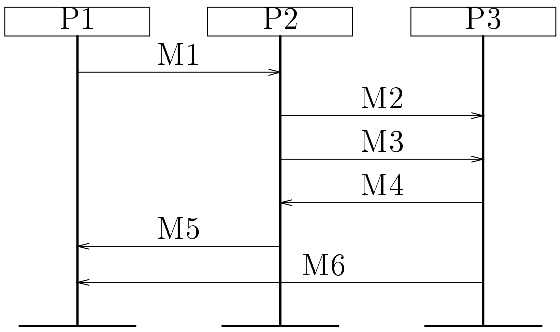
To circumvent the problem, we suggest an extension to the MSC standard, titled *compositional message sequence charts* (CMSC and HCMSC). This extension allows specifying MSCs with unmatched sends and receives. The semantics of the new construct prescribes how to combine such MSC nodes together. We use the extended notation to suggest an algorithm for the automatic generation of HCMSC representations for finite state systems. We show that basic properties of HCMSCs become undecidable, e.g. the question whether a message will be received in at least one computation. We propose to use a restriction of HCMSCs, called *realizable* HCMSCs. We show how to test whether an HCMSC is realizable in an efficient way. The notion of realizable HCMSC is quite natural, as our algorithm for the HCMSC generation already yields HCMSCs of this kind.

The deficiency of the original MSCs was also recognized in [10]. 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 [10], as we study the effect of allowing communication between sequentially composed CMSCs. That is, a communication that starts in one CMSC and ends in a subsequent one. Notice that our solution is more canonical, since it does not make use of special message names for the purpose of binding by name identifiers, as in [10]. Further papers considering this issue are [8, 13]. These papers look at the problem of checking whether a finite state protocol can be translated into an HMSC. In the first of these papers, it is shown that this question is decidable, whereas the second paper shows that for a natural class of finite state protocols one can efficiently check whether the translation into an equivalent HMSC is possible.

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




```
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;
```

Fig. 1. Visual and textual representation of an MSC

- V *is a (finite or infinite) 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.*
- $m \subseteq V \times V$ *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 matched 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.,

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

An MSC with a finite (an infinite, respectively) set of events is called a finite (infinite, respectively) MSC.

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 [9]). Clearly, the visual ordering can be defined equivalently as the transitive closure of the relation $\longrightarrow$. 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 $(V, <)$.

Example 1. Let us denote in the example MSC given in Figure 1 by e_i the send event and by f_i the receive event of message Mi, $1 \le i \le 6$. Then we have $V = \{e_1, \ldots, e_6, f_1, \ldots, f_6\}$, $\mathcal{P} = \{P1, P2, P3\}$, $\mathcal{N} = \{M1, \ldots, M6\}$ and $N(e_i) = N(f_i) = Mi$ for all i. The events located on $P1$ are $\{e_1, f_5, f_6\} = L^{-1}(P1)$, with $T(e_1) = \mathrm{s}$, $T(f_5) = T(f_6) = \mathrm{r}$, and $e_1 < f_5 < f_6$. This ordering is the time ordering of events on $P1$. We also have $m(e_i, f_i)$ and $e_i < f_i$ for all i (message ordering). In particular, $e_1 < f_1 < e_2 < f_2$ and e_1 is the minimal event of the MSC w.r.t. visual ordering.

A *type* is a triple (i, j, C), including two processes P_i and P_j, and a message name $C \in \mathcal{N}$. Each *send* or *receive* event has a type, according to the origin and destination of the message, and the label of the message. Matching events have the same type.

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 $\longrightarrow$ is shown. Notice for example that the *send* events of the two messages, $M5$ and $M6$, are unordered.

Definition 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 $V_1 \cap V_2 = \emptyset$ (we can always rename events so that the sets become disjoint), 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. The infinite concatenation of finite MSCs is defined in a similar way.

Definition 3. *Let M_1, M_2, $\ldots$, $\ldots$ be an infinite sequence of finite MSCs, $M_i = \langle V_i, <, \mathcal{P}, \mathcal{N}_i, L_i, T_i, N_i, m_i\rangle$. Then the infinite concatenation $M_1 M_2 \ldots$ is defined as the MSC $\langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m\rangle$ where $V = \cup_{i \ge 1} V_i$ is the disjoint union of the V_i, $\mathcal{N} = \cup_{i\ge 1}\mathcal{N}_i$, $L|_{V_i} = L_i$, $T|_{V_i} = T_i$, $N|_{V_i} = N_i$, $m = \cup_{i \ge 1} m_i$ and*

$$< \; = \; \bigcup_{i \ge 1} <_i \; \cup \{(p, q) \mid L_i(p) = L_j(q) \wedge p \in V_i \wedge q \in V_j \wedge i < j\}\,.$$

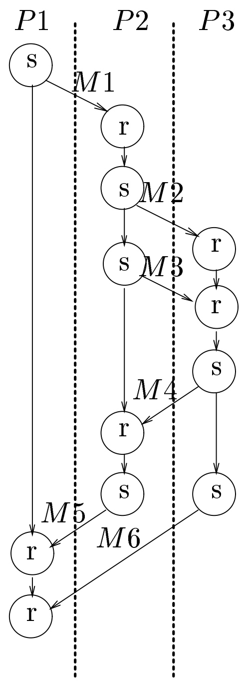


Fig. 2. The partial order between the events of the MSC in Figure 1.

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) that starts 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 4. *An HMSC N is a 4-tuple $\langle \mathcal{S}, \tau, s_0, c \rangle$ where $\mathcal{S}$ is a finite set of states, each labeled by some finite 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 $c(s)$. By $\tau \subseteq \mathcal{S} \times \mathcal{S}$ we denote the* edge relation *and the* initial *state is $s_0 \in \mathcal{S}$. An* execution *of N is a (finite or infinite) MSC $c(s_0)\,c(s_1)\,c(s_2)\,\ldots$ associated with a maximal path of N that starts with the initial state s_0.*

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

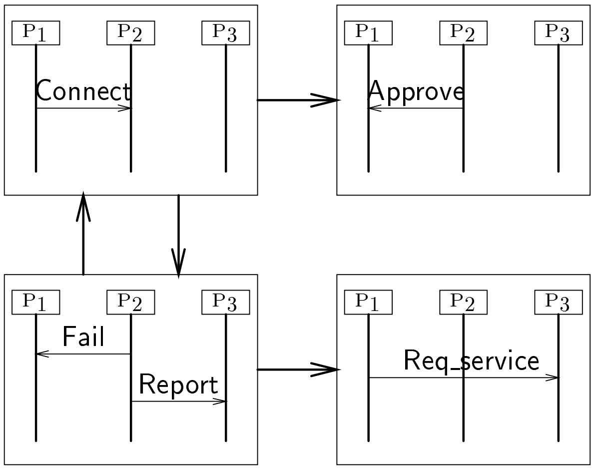


Fig. 3. An HMSC graph

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.

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 execution 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 infinite MSC whose prefix appears in Figure 4. We assume that $P1$ repeatedly sends a message m to $P2$, and $P2$ repeatedly sends m' to $P1$. We omit the message labels m, m' below. We can model for example each of the processes $P1$ and $P2$ by a finite state machine. Here, $P1$ starts by sending twice message m to $P2$, then he alternates between receiving m' from $P2$ and sending back m to $P2$. Process $P2$ alternates between sending m' to $P1$ and receiving m from $P1$. We show that this infinite MSC cannot be decomposed into a product of finite MSCs. We start with the *send* event e_1 and *receive* event f_1. Obviously, because of the compulsory matching in HMSCs, they must belong to the same MSC node. We have the *send* event g_1 preceding f_1, on the same process line, while its corresponding *receive* event h_1 succeeds the *send* e_1. Thus, the events g_1 and h_1 must be in the same node with e_1 and f_1. For the same reason, we have that e_2 and f_2 must belong to the same node with g_1, and h_1, and so forth.

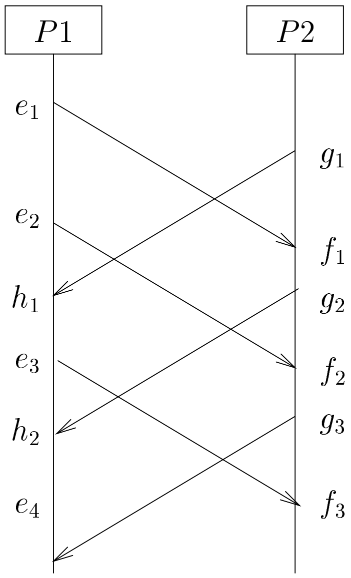


Fig. 4. A prefix of an MSC execution that cannot be decomposed .

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 [15], where messages and acknowledgments 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 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 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. CMSC arrows where both the send and the receive are unmatched events are forbidden. Moreover, we also disallow an unmatched *receive* event to be followed by a matched *receive* event of the same type in the same CMSC node. Similarly, we disallow an unmatched *send* event to be preceded by a matched *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.

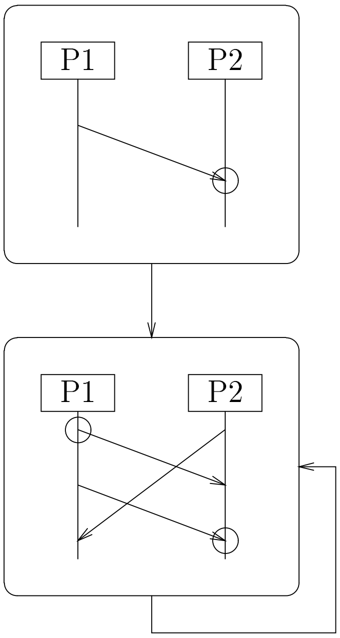


Fig. 5. A decomposition of the execution in Figure 4.

Before defining the concatenation of CMSCs let us denote a CMSC as *left-closed*, if it does not contain unmatched *receive* events.

Definition 5. *The* concatenation M_1M_2 *of two CMSCs* $M_1 = \langle V_1, <_1, \mathcal{P}, \mathcal{N}_1, L_1, T_1, N_1, m_1\rangle$ *and* $M_2 = \langle V_2, <_2, \mathcal{P}, \mathcal{N}_2, L_2, T_2, N_2, m_2\rangle$ *over disjoint events sets, is defined when the following conditions hold:*

1. *M_1 is left-closed.*

2. *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.*
3. *If M_2 contains a matched send event of type t then the number of unmatched $\tilde{r}$eceive events of type t in M_1 is equal to the number of unmatched send events of type t in M_1.*

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 ith unmatched send in M_1 is paired up with the ith unmatched receive event of the same type in M_2. Notice that the function m is uniquely defined.

The concatenation M_1M_2 is then defined as $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2 \cup m\rangle$, where

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

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_1M_2 also does. This follows from the last requirement in the definition. Note that this requirement is consistent with our fifo definition, which applies only to messages with the same name. Thus, if we require instead that the fifo condition is satisfied by *all* messages from one process to another means that we have to modify the last requirement of the definition of the concatenation accordingly. Infinite concatenation and HMSCs are defined in an analogous way to Definitions 3, 4.

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, where some simple properties can be checked, see e.g., [12], in HCMSC one cannot decide even the trivial property of whether a particular message can be sent or received in at least one computation. The undecidability proof will be a reduction 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 there is some integer $n > 0$ and some sequence of indexes $i_1, i_2, \ldots, i_n$ such that $v_{i_1} v_{i_2} \ldots v_{i_n} = w_{i_1} w_{i_2} \ldots w_{i_n}$. We require in addition that the PCP solution is such that $i_n = 1$. This is not a restriction, since we can use a suitable encoding so 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 need this variant of PCP for technical reasons which will become clear in the proof below.

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 HCMSC N has the form $F(E_1 + \cdots + E_m)^*(E'_1 + \cdots + E'_m)^* E'_1 F'$. That is, N 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, any number of times. Then one can take any number of nodes of the form E'_i, followed by the nodes E'_1, F'. The sink node F' contains a message from P_2 to P_5, 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 sequence $w_{i_1} \cdots w_{i_n}$ corresponding to the unmatched *send* events on the path in N is a prefix of $v_{i_1} \cdots v_{i_n}$, corresponding to the unmatched *receive* events, and $i_n = 1$. Under our assumption about PCP words this means equality, i.e., $v_{i_1} \dots v_{i_n} = w_{i_1} \dots w_{i_n}$, and we obtained a solution. Notice that we might have unmatched sends on P_1 and P_3 in the CMSC associated with the path in N. This explains why we obtain only the prefix relation and why we need the particular PCP encoding. Thus, the message from P_1 to P_5 is received if and only if there is a nonempty solution to the PCP instance.

6 Realizable HCMSCs

The way we defined HCMSCs makes that not all executions correspond to CMSC scenarios. We define *realizable HCMSCs*, a subclass where all maximal executions define left-closed CMSC. Note that we explicitely allow executions with unmatched *send* events. For example, the HCMSC of Figure 5 is such that every finite execution is a left-closed CMSC with unmatched sends. However, the (unique) infinite execution corresponds to an infinite MSC.

Definition 6. *An HCMSC is* realizable *if the execution of every finite path starting with the initial state is a left-closed CMSC.*

We will show that one can efficiently test whether an HCMSC is realizable. Consider the CMSC $M = c(s_0)c(s_1) \cdots c(s_n)$ associated with a finite path $\chi = s_0, s_1, \dots, s_n$ of the HCMSC N with 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 realizable is that $D_t(\chi) \geq 0$ for every loop χ and every type t. More generally, an HCMSC $N = \langle \mathcal{S}, \tau, s_0, c \rangle$ is realizable if and only if every node s which is accessible from the initial node satisfies the following conditions. Assume that node s contains x unmatched receives of type t. Then $D_t(\chi) \geq x$ for all paths χ from s_0 to s' with $(s', s) \in \tau$. Moreover, if node s also contains a matched *send* of type t, then $D_t(\chi) = x$ for all paths χ from s_0 to s' with $(s', s) \in \tau$.

We describe below the algorithm for checking that an HCMSC N is realizable. 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 $\gamma(s', s) = d_t(s')$. That is, each edge is labeled by the t-deficit of its source node. Then all we have to do is the following:

1. Check that $G_t(N)$ has no cycle with negative weight.
2. Check for all states s, s' such that $(s', s) \in \tau$: the minimal weight d of a path from s_0 to s' satisfies $d \geq x$, where x is the number of unmatched receives of type t in s.
3. Check for all states s, s' such that $(s', s) \in \tau$ and s' contains a matched *send* of type t: the *maximal* weight d of a path from s_0 to s' satisfies $d \leq x$, where x is the number of unmatched receives of type t in s.

For the first two items above we can apply a dynamic programming algorithm (Warshall's algorithm) for computing the shortest paths between all pairs of nodes in time $O(|\mathcal{S}|^3)$. That is, assuming that $\mathcal{S} = \{s_1, \ldots, s_n\}$ we compute the minimal weight of paths from state s_i to state s_j by allowing as intermediate nodes $\emptyset$, then $\{s_1\}$, $\{s_1, s_2\}$ up to $\mathcal{S}$. 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). Combining the second and the third item above we need to check for all states s containing a matched *send* of type t and all nodes s' where $(s', s) \in \tau^*$ that all paths from s_0 to s' have the same t-deficit, say $D(s')$. This means that we first compute the t-deficits along one path χ from s_0 to s. Let $D(s) = D_t(\chi)$. Then we compute backwards, from *state* s on, the deficits $D(s')$ for all nodes s' belonging to paths from s_0 to s. It remains to check for each pair s', s'' of nodes between s_0 and s with $(s', s'') \in \tau$ that we have $D(s') + d_t(s'') = D(s'')$. The last step can be done edge by edge. The overall complexity is in $O(|\tau|)$. Doing all this for all graphs $G_t(N)$ yields an $O(|\mathcal{P}|^2|\mathcal{S}||\tau|)$ algorithm for checking whether N is realizable.

We conclude this section with a remark on the regularity of the set of executions of an HCMSC. Note that a realizable HCMSC N has bounded message queues if and only if $D_t(\chi) = 0$ for every loop χ in N and every type t. It is not difficult to see that bounded message queues do not ensure 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, 11]. 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. There is an edge from process P to process Q if P sends a message 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 target process Q. As for HMSCs we require that the communication graph of any loop in the HCMSC is strongly connected.

Proposition 1. *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. 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 (see [3, 11]). 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 the fact that the deficit $D_t(\chi)$ of any path χ 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 with asynchronous message passing to (realizable) HCMSC.

We are given a finite state space $G = (S, S_0, E, \Sigma)$, with states S, initial states $S_0 \subseteq S$, and edges $E \subseteq S \times \Sigma \times S$, labeled 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 label of a node e is the label of e in G.
- The *initial nodes* $N_0 \subseteq N$ of H correspond to the edges of G that exit from an initial state of 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.

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

1. Minimize the number of unmatched events appearing in the individual CMSC nodes, if possible obtaining an HCMSC without any unmatched events (however, recall from Section 3 that this is not always achievable).
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.

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. *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 [14] 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 at the end of the section 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.

function $expand\_node(s, sleep)$;
local explored, working_set, new_sleep, fixed;
 $explored := \emptyset$;
 $fixed := false$;
 if $en(s) = \emptyset$ then return $true$ fi;
 $working\_set := en(s) \setminus sleep$;
 while $working\_set \neq \emptyset$ do
  $\alpha :=$ biggest action in $working\_set$ according to '$\prec$';
  $working\_set := working\_set \setminus \{\alpha\}$;
  $s' := \alpha(s)$;
  $new\_sleep := (sleep \cup explored) \setminus dep(\alpha)$;
  $explored := explored \cup \{\alpha\}$;
  if $s' \in Z$ orelse s' is terminal orelse $exists\_node(s', new\_sleep)$
    orelse $expand\_node(s', new\_sleep)$ then
   $fixed := true$;
   $create\_edge((s, sleep), \alpha, (s', new\_sleep))$ fi;
  fi
 end while;
 if fixed then $store\_node\_in\_hash(s, sleep)$;
 return $fixed$;
end $expand\_node$.

Fig. 6. A reduced state space generation algorithm

Properties of the Algorithm. 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

[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.

transitions and f, e are individual transitions and $f \notin dep(e)$. Let $\stackrel{*}{\longrightarrow}$ be the transitive and reflexive closure of $\longrightarrow$.

The relation '$\sqsubseteq$' between strings over Σ is such that $v \sqsubseteq w$ when

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

Lemma 1. *If $v \sqsubseteq w$, then a CMSC with a linearization v is a prefix of a CMSC with a linearization w.*

Sketch of proof. We can show that the transitions of each process in v are a prefix of the transitions of the same process in w. ∎

Lemma 2. *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 3. *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. ∎

The Matching Algorithm. Consider an CMSC node M constructed by the above algorithm. By construction, each path from the initial node to M has the same t-deficit, for every type t (since the states of the original finite state machine refer to the contents of queues). Notice also that every loop in the HCMSC thus generated has zero t-deficit, for any type t. Suppose now that the t-deficit of paths from the initial MSC M_0 to the predecessors of M is equal to d. Then we match the events in M as follows.

1. Mark the first d *receive* events of type t in M 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 [2] system for generating the HCMSC visual structure.

Acknowledgment. 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 *Proc. of the 22nd Int. Conf. on Software Engineering*, pp. 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 *Proc. of CONCUR'99*, LNCS no. 1664, 1999.
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 *Proc. of ICALP'00, 2000*, LNCS no. 1853, pp. 675-686, 2000.
9. ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.
10. S. Mauw and M. Reniers. High-level message sequence charts. In *SDL'97: Time for Testing - SDL, MSC and Trends. Proc. of the SDL Forum'97*, pp. 291–306, 1997.
11. A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *Proc. MFCS'99*, LNCS no. 1672, pp. 81–91, 1999.
12. A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS no. 1378, pp. 226–242, 1998.
13. A. Muscholl and D. Peled. High-level message sequence charts and finite-state communication protocols. Submitted.
14. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Proc. of CAV '93*, LNCS no. 697, pp. 409–423, 1993.
15. A. Tanenbaum, Computer Networks, Prentice Hall, 1988.