Recovering Representations of Systems with Repetitive Subfunctions from Observations

Guy-Vincent Jourdan* and Husnu Yenigun†
*School of Information Tech. and Engineering
University of Ottawa, 800 King Edward Avenue
Ottawa, Ontario, CANADA K1N 6N5
gvj@site.uottawa.ca
†Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, TURKEY
yenigun@sabanciuniv.edu

Abstract

This paper proposes an algorithm for the construction of an MSC graph from a given set of actual observations of an existing concurrent system which has repetitive subfunctions. When a design representing the current functionality of the existing system is desired, such a graph can be checked for safe realizability and be used as input to existing synthesis techniques to construct the design for the system functionality.

I. Introduction

A concurrent system consists of two or more processes communicating among themselves via message exchanges. Each individual functionality (i.e., intended or actual behavior) of such a system can be viewed as a sequence of subfunctions. Often, depictions of individual intended behaviors of a concurrent system are given by designers as Message Sequence Charts (MSCs) [1], [2]. An individual MSC is a visual description of a series of message exchanges among communicating processes in a concurrent system where the local view of the message exchanges is a total order with respect to each process but the global view is a partial order. A tuple consisting of a local view for each process of the message exchanges depicted in an MSC uniquely determines that MSC. Thus, an MSC represents a partial order execution of a concurrent system which stands for a set of linearizations

(total order executions of the system) determined by considering all possible interleavings of concurrent message exchanges implied by the partial order.

Formal semantics associated with MSCs provides a basis for their analysis such as detecting timing conflicts and race conditions [3], non–local choices [4], model checking [5], and checking safe realizability [6]. Safe realizability is a property that characterizes whether behaviors represented by a given set of MSCs can be realizable by some deadlock-free implementation of communicating processes. [6] shows that if the given set of MSCs is safely realizable then an approach similar to existing synthesis algorithms can be used to synthesize a deadlock-free design. If it is not, then unspecified (and possibly unwanted) MSCs that are implied can be detected and fed back to the design process. While checking for safe realizability of a given set of MSCs that does not imply any repetitive system subfunctions can be done in polynomial time [6], that of a given bounded MSC graph is EXPSPACE–complete [7].

One of the aims of the reverse engineering [8], [9] is to recover the design of an existing system from the run time behavior of its implementation. In this paper, we consider the reverse engineering of designs of existing concurrent systems from given sets of observations of their implementations. Here, a given set of observations consists of individual linearizations of a set of MSCs that is not given. We propose an algorithm for constructing an MSC graph from a given set of observations of an existing concurrent system as a representation of the system's design.

We assume that every repetitive subfunction of the system (if any) is represented in the given set of observations at least twice: once with no occurrence or one occurrence, and once with two or more consecutive occurrences. This assumption stems from the fact that some repetitive subfunctions can be skipped during executions, whereas others cannot. However, in all the observations in which the repetitive subfunction occurs two or more times, it must occur exactly the same number of times to simplify the loop inference algorithms.

When the resulting graph is acyclic, it is guaranteed that the system's functionality is free from repetitive subfunctions. Otherwise, the resulting MSC graph is cyclic due to the existence of repetitive subfunctions in system's behavior. In either case, the resulting MSC graph may then be checked for safe realizability and, when found safely realizable, can be used directly as input to the existing automated synthesis techniques.

An earlier version of this work appeared as [10]. Some solutions for the same problem, under different assumptions, were presented in [11], [12].

The rest of the paper is organized as follows: Section II introduces the terminology and notation used throughout the paper. Section III gives the formal definition of the problem. Section IV presents the construction of

an MSC graph from a given set of observations. Section V discusses some open problems and gives the concluding remarks. The proofs and complexity analysis can be found in the appendices.

II. Preliminaries

The notation we will be using is directly adopted from [6]. A concurrent system $\mathbb{P}$ is a set of processes $\mathbb{P} = \{P_1, P_2, \ldots, P_n\}$, communicating with each other by passing messages from an alphabet Σ, over infinite slot (not necessarily FIFO) buffers. An event labeled as $snd(i, j, a)$ denotes the transmission of a message $a \in \Sigma$ by the process P_i to the process P_j. Similarly, an event labeled as $rcv(j, i, a)$ denotes the reception of a message a by the process P_j, which must have been sent by P_i.

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. Let $\hat{\Sigma}_i^s = \{snd(i, j, a) | j \in [n], a \in \Sigma\}$, $\hat{\Sigma}_i^r = \{rcv(i, j, a) | j \in [n], a \in \Sigma\}$, and $\hat{\Sigma}_i = \hat{\Sigma}_i^s \cup \hat{\Sigma}_i^r$ be the set of send event labels, the set of receive event labels, and the set of event labels of the process P_i, respectively. Then we define, $\hat{\Sigma}^s = \cup_{i \in [n]} \hat{\Sigma}_i^s$, $\hat{\Sigma}^r = \cup_{i \in [n]} \hat{\Sigma}_i^r$, and $\hat{\Sigma} = \hat{\Sigma}^s \cup \hat{\Sigma}^r$, as the set of send event labels, set of receive event labels, and the set of event labels, respectively.

A *word* w over an alphabet $\hat{\Sigma}$ is a finite sequence of elements from that alphabet. For two words w and w', juxtaposition of the two words, ww', denotes the concatenation of w and w'. w' is said to be a *prefix* of w, if there exists w'' such that $w = w'w''$. For an integer $k \geq 0$, $w^{(k)}$ denotes the concatenation of k copies of w, where $w^{(0)}$ is defined to be the empty word. We will use the notation $w^\star$ to denote concatenation of 0 or more copies, and w^+ to denote concatenation of 1 or more copies of the word w.

Given a word w over $\hat{\Sigma}$ and an event label $\alpha \in \hat{\Sigma}$, let $\#(w, \alpha)$ be the number of occurrences of α in w. w is said to be *well–formed* if $\forall$prefix w' of w, $\forall i, j \in [n]$ and $\forall a \in \Sigma$, $\#(w', snd(i, j, a)) - \#(w', rcv(j, i, a)) \geq 0$. In other words, every receive event must be preceded by a matching send event. w is said to be *complete* if $\forall i, j \in [n]$ and $\forall a \in \Sigma$, $\#(w, snd(i, j, a)) = \#(w, rcv(j, i, a))$. That is, every message a sent by P_i to P_j must be received by P_j, within the word.

Given a word w and a set K, we use $w|_K$ (projection onto K) to denote the word that is derived by removing all the elements in w that are not in K. We use the same notation to denote the restriction of the domain of a binary relation R onto a set K. That is, $R|_K$ is the projection of R onto K.

Again, we directly adopt the formal definition of an MSC as introduced in [6]. A Σ–labeled MSC M for a concurrent system $\mathbb{P}$ is composed of the

following components[1]:
(i) A finite set S of send events and a finite set R of receive events. Let $E = S \cup R$.
(ii) A mapping $l : E \rightarrow \hat{\Sigma}$ that maps each event to a label such that $l(S) \subseteq \hat{\Sigma}^S$ and $l(R) \subseteq \hat{\Sigma}^R$.
(iii) A bijection $f : S \rightarrow R$ mapping each send event e with its matching receive event such that if $l(e) = snd(i, j, a)$ then $l(f(e)) = rcv(j, i, a)$.
(iv) A mapping $p : E \rightarrow [n]$ such that if $l(e) = snd(i, j, a)$ then $p(e) = i$, and if $l(e) = rcv(i, j, a)$ then $p(e) = i$. p simply gives the process on which e occurs. Let $E_i = \{e \in E | p(e) = i\}$ be set of events of P_i for $i \in [n]$.
(v) For each $i \in [n]$, a total order $\leq_i$ on E_i, such that when the relation $\leq$ is defined to be

$$\leq \triangleq \cup_{i \in [n]} \leq_i \cup \{(s, f(s)) | s \in S\}$$

the transitive closure $\leq^\star$ of $\leq$ is a partial order on E.

The total order $\leq_i$ on E_i gives a strict execution order of the events of P_i as seen on the vertical process lines of P_i in the visual representation of the MSC. The pairs $(s, f(s)) \in \leq$ correspond, in the visual representation, to the message passing arrows from the process line of $p(s)$ to the process line of $p(f(s))$.

Throughout the paper, Σ and $\mathbb{P}$ are assumed to be fixed and all the MSCs mentioned will be Σ–labeled and defined on $\mathbb{P}$. Let Φ denote the set of all Σ–labeled MSCs for $\mathbb{P}$.

Let $|E|$ denote the cardinality of the set E. A permutation of the events E of an MSC as $e_1 e_2 \ldots e_{|E|}$ is *valid* when $\forall i, j \in [|E|]$, $e_i \leq^\star e_j$ implies $i \leq j$. In other words, the total order induced by the given permutation on E is consistent with the partial order $\leq^\star$. A word w on $\hat{\Sigma}$ is a linearization of an MSC M if there exists a valid permutation $e_1 e_2 \ldots e_{|E|}$ of M such that $w = l(e_1) l(e_2) \ldots l(e_{|E|})$. The language of an MSC M, denoted by $L(M)$, is the set of all linearizations of M. Two MSCs M and M' are considered to be equal, $M = M'$, iff $L(M) = L(M')$.

Note that, by definition, if $w \in L(M)$, then w is well–formed and complete. Further note that, $\forall w, w' \in L(M)$, $w|_{\Sigma_i} = w'|_{\Sigma_i}$. In other words, the projections of the words in $L(M)$ onto the event labels of a process is unique. This follows from the fact that all the valid permutations of M must respect the total order $\leq_i$ which is included in $\leq^\star$. In fact, this unique word, which will be denoted by $M|_i$, is the concatenation of the labels of the events that appear on the vertical line of P_i in the visual representation of M. Therefore, as shown in [6], given a well–formed and complete word

[1]Note that, this definition of MSCs does not include the notion of *co–region* (the region on a process line in which the events of the processes are not ordered) in MSCs which is also omitted in this paper.

w, there exists a unique MSC M, such that $w \in L(M)$, under a non–degeneracy assumption (that there is no message overtaking between same labeled events) which we also adopt in this paper. We will denote this unique MSC by $msc(w)$. We also let $\overline{w} = L(msc(w))$.

Due to this fact, an MSC can be characterized by the sequence of sequences of the event labels that appear on the processes, i.e. $M = \langle M|_i \mid i \in [n]\rangle$. Given such a sequence, the actual MSC can be constructed easily as explained in [6].

Proposition 1: Let M and M' be two MSCs. $M = M'$ iff for each process P_i, $M|_i = M'|_i$.

Given two MSCs M and M', with the set of events respectively E and E', M' is said to be a *prefix* (resp. *suffix*) of M, iff:
(i) $E' \subseteq E$.
(ii) $e \in E'$ implies $\forall e' \in E$, if $e' \leq^\star e$ then $e' \in E'$ (resp. $e \in E'$ implies $\forall e' \in E$, if $e \leq^\star e'$ then $e' \in E'$)
(iii) $e \in E' \cap S$ implies $f(e) \in E'$ (resp. $e \in E' \cap R$ implies $f^{-1}(e) \in E'$, where f^{-1} is the inverse of the bijection f)
(iv) $S' = S \cap E'$, $R' = R \cap E'$, $l' = l|_{E'}$, $f' = f|_{E'}$, $p' = p|_{E'}$, and $\forall i \in [n]$, $\leq'_i = \leq_i |_{E'}$.

Let M, M_p and M_s be MSCs with the corresponding set of events E, E_p and E_s such that M_p is a prefix of M, M_s is a suffix of M, $E_p \cap E_s = \emptyset$ and $E = E_p \cup E_s$. Then, M is said to be the *sequential composition* of M_p and M_s, denoted by the juxtaposition of M_p and M_s as M_pM_s. Given two MSCs M' and M'', $L(M'M'') = \{w | w \in \overline{w'w''}, w' \in L(M'), w'' \in L(M'')\}$.

For an integer $k \geq 0$, $M^{(k)}$ denotes the sequential composition of k copies of M, where $M^{(0)}$ is defined to be the empty MSC, i.e. an MSC with the empty set of events. We will use the notation $M^\star$ to denote sequential composition of 0 or more copies, and M^+ to denote sequential composition of 1 or more copies of the MSC M.

While describing the scenarios that a concurrent system must perform, a set of MSCs can be used. However, when this set gets large, it is usually presented in a more structured way by using High Level MSCs, or HMSCs, which is formally equivalent to an MSC graph given below. An MSC graph is a labeled transition system $G = (V, v_0, v_f, T)$, where V is a finite set of nodes, $v_0, v_f \in V$ are the entry and exit nodes (respectively). The relation $T \subseteq V \times \Phi \times V$ gives the edges between the nodes with the labels from Φ. A *path* in G is a sequence of edges

$$(v_1, M_1, v_2)(v_2, M_2, v_3)(v_3, M_3, v_4) \cdots (v_m, M_m, v_{m+1})$$

Such a path is said to start at node v_1 and end at node v_{m+1}. The language of a path is given by the concatenation of the language of the MSCs that appear on the edges, $L(M_1)L(M_2) \cdots L(M_m)$. Given an ordered pair of

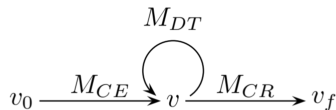

Fig. 1. An example MSC Graph

nodes (v, v'), the language of the pair (v, v'), denoted by $L(v, v')$, is the union of the languages of all the paths that start at v and end at v'. The language of a node v, denoted by $L(v)$, is $L(v, v_f)$. The language of an MSC graph G is defined as $L(G) = L(v_0)$. We will use the notation $v \xrightarrow{M} v'$ to denote $(v, M, v') \in T$.

III. Problem Definition

Each function implemented by a concurrent system can be viewed as a combination of some subfunctions. For example, in a file transfer function of a communication protocol, we may have connection establishment (CE), data transfer (DT), connection release (CR) subfunctions. If one could identify the subfunctions as they are being executed, then a typical execution would consist of the following steps:

$$\text{CE, DT, DT, ..., DT, CR} \quad (1)$$

Based on the size of the data being transferred, the subfunction DT would be executed repeatedly, as many times it is required to transfer the amount of data at hand. If we consider how one would start describing such a function at an abstract level when the system was first built, it is not unreasonable to imagine that an MSC graph similar to the one given Figure 1 had been used.

In the context of reverse engineering, several attempts appeared in the literature to recover the design of an implementation from a given set of observations [13], [14], [15], [16], [17], [18]. However, if the sequence given in (1) is reverse engineered with the current techniques, the existence of repetitions of DT will cause problems. In general, it is not possible to decide if the repetitions of DT in (1) are due to a repetitive subfunction (say a *loop*) or due to the sequential appearance of DT in the design. Current techniques favor the latter and therefore, do not attempt to recover a design with the repetitive subfunctions. In this paper, we will introduce a method that will help recover designs *with* repetitive subfunctions.

As observations, we consider the execution logs (logs of message transmissions and receptions of the processes) of an implementation Imp of a concurrent system. In other words, if Σ is the set of messages used for the communication between the processes of Imp, then an observation is a well–formed and complete word over $\hat{\Sigma}$. We assume that an observation

$w \in \hat{\Sigma}^\star$ corresponds to a complete execution of a single function of Imp, and the functions are assumed to start from the initial system state, and end back at the initial system state, without going through the initial system state. That is, if w is an observation and the system is at the initial state, then after performing the message exchanges given in w (in the order they appear in w), the system goes back to the initial state right after the last member of w (which must be a reception since w is complete) is realized. Furthermore, at no point in w, the system must be in the initial system state, since otherwise the prefix of w up to that point would be considered as a separate observation.

Suppose that we are given a set of observations O. In Section II, it is shown that a well–formed and complete word corresponds to a unique MSC. Consider the MSC $msc(w)$ corresponding to an observation $w \in O$, and a word $w' \in L(msc(w))$ but $w' \notin O$. Since the individual processes are behaving, from their local point of view, exactly in the same way for both w and w', although not given as an observation, w' must also be an observation of Imp. Only the interleaving preferences between the processes change from w to w'. Therefore, the given set of observations O, together with these implied observations, actually corresponds to a wider set which is:

$$\overline{O} = \bigcup_{w \in O} \overline{w} = \bigcup_{w \in O} L(msc(w))$$

Since each given observation in $w \in O$ actually corresponds to an MSC $msc(w)$, we consider our input to be the set of MSCs $\mathbb{M} = \{msc(w) \mid w \in O\}$, and we will consider an observation to be an MSC from now on unless stated otherwise.

To be able to infer a loop in the design by looking at observations, we demand some evidence. We do not readily accept that a repeated pattern seen in a single observation is due to a loop. However, if the same pattern is seen different number of repetitions *within the same context*, then we assume this is a sufficient evidence for the existence a loop. Below is the formal definition of the notion of this evidence.

Definition 1: An MSC M is said to be *the basic repetitive MSC* of MSC M' if $M' = M^{(k)}$ for some $k \geq 2$ and there does not exist a basic repetitive MSC of M.

Definition 2: Two MSCs M_1 and M_2 are said to *infer M to be repetitive within the context M_p–M_s* if all the following are satisfied:
(i) M does not have a basic repetitive MSC;
(ii) $M_1 = M_p M^{(k)} M_s$ for some $k \geq 2$;
(iii) M is not a suffix of M_p;
(iv) M is not a prefix of M_s;
(v) either $M_2 = M_p M_s$ (in which case M is said to be *while–repetitive*) or $M_2 = M_p M M_s$ (in which case M is said to be *repeat–repetitive*).

Note that Definition 2 captures the essence of two different repetitive subfunctions, one which can be skipped, whereas the other cannot be skipped during the execution of the system. In order to differentiate these two different types, we call them as *while* and *repeat* repetitive respectively, by using an analogy to standard programming loop types.

What we require in observations to infer a loop is that, there must be an observation in which the loop is iterated $k \geq 2$ times, and there must also be another observation in which the same loop iterated the least possible number of times (which is 0 for a while loop, and 1 for a repeat loop). Furthermore, these two observations must have exactly the same prefix before the iterations of M, and exactly the same suffix after the iterations of M, that is they must appear within the same context. Under such an evidence, M will be assumed to be generated by a loop in the design, or more precisely, by the matching loops (sending and receiving matching messages) in the processes.

It is also important to note that, an iteration of a loop in an observation is allowed only to provide the required evidence to infer a loop. Moreover, each repetitive subfunction is inferred only once using the given observations. Further, an outer loop cannot start with an iteration of an inner loop, and cannot end with an iteration of an inner loop. Furthermore in order to establish the relative ordering of two or more loops $l_1, l_2, \ldots, l_n$ in an MSC graph, if in an observation there are $k \geq 2$ iterations of l_i, then for any l_j such that $j < i$, a nonzero iteration of l_j must also occur before $k \geq 2$ iteration of l_i in the same observation. In case of nested loops, in this order we assume that the outer loop starts before the inner ones. Note that, this requirement will be satisfied if l_j is a repeat loop, or if l_j is an outer loop of l_i. Therefore, this requirement must only be enforced if l_j is a while loop and l_i is not an inner loop of l_j.

IV. Problem Solution

When a pair of MSCs M_1 and M_2 is identified within the given set $\mathbb{M}$, such that M_1 and M_2 infers M to be repetitive within the context M_p–M_s, then M_1 and M_2 will be represented in the output MSC graph by using either one of the following templates, depending on whether it is while–repetitive (left) or repeat–repetitive (right).

$$v_0 \xrightarrow{M_p} v \xrightarrow{M_s} v_f \quad (\text{self-loop } M \text{ at } v) \qquad\qquad v_0 \xrightarrow{M_p} v \xrightarrow{M} v' \xrightarrow{M_s} v_f \quad (\text{self-loop } M \text{ at } v')$$

The MSC M representing a repetitive functionality will be called *the loop body* (or body of the loop). We will also split the context M_p–M_s into two, and call M_p *the left–context* and M_s *the right–context* of the loop.

An algorithm that will find such pairs of MSCs, must identify the context part, i.e. common prefix M_p and the common suffix M_s, and must also check if the part remaining in the middle has a basic repetitive MSC. Before presenting such an algorithm, we need to introduce the following notions on MSCs. A *common prefix* of two MSCs M_1 and M_2, is an MSC M, such that M is a prefix of both M_1 and M_2. The *maximal common prefix* of M_1 and M_2 is a common prefix M of M_1 and M_2 with the largest number of events. Similarly, M is said to be a *common suffix* of M_1 and M_2 if it is a suffix of both M_1 and M_2. The common suffix of M_1 and M_2 with largest number of elements is called the *maximal common suffix* of M_1 and M_2.

Suppose that $M = M_p M_s$. Given M and M_p, it is trivial to find M_s, by simply removing all the events in M_p from the first part of M. Similarly, when we are given M and M_s, removing all the events in M_s starting from the tail of M will give M_p. In both of these algorithms, we need to match the labels of the events. Let us assume that these algorithms are called as "remove_prefix" and "remove_suffix", respectively.

We can now present an algorithm that can check if two given MSCs identify a repetitive MSC. Without loss of generality, we assume that M_1 has more events than M_2.

Recall that, in order to infer M to be repetitive from M_1 and M_2, we must have $M_1 = M_p M^{(k)} M_s$ and either $M_2 = M_p M_s$ or $M_2 = M_p M M_s$. The maximal common prefix of M_1 and M_2 will be consisting of M_p, followed by an optional single occurrence of M. In either case, M_2'' in Algorithm 1 given in Figure 2 must be empty (line 7). At line 10 and 11, we check if M_1'' has a basic repetitive MSC, by using the algorithm "basic_repetitive_MSC", which is explained in Section IV-A. For the time being, assume that it returns the basic repetitive MSC of its input MSC, if there exists one, or returns the empty MSC otherwise. If such an M does not exist, we may still infer a repetitive MSC. This corresponds to the case where $M_1 = M_p M^{(2)} M_s$ and $M_2 = M_p M M_s$. In this specific case, the maximal common prefix M_{mp} would be $M_p M$. Line 12 checks this singularity, and the correct left–context is calculated at line 13.

However, if M_1'' has a basic repetitive MSC M, then we decide if it is while–repetitive or repeat–repetitive, between the lines 18–23. Note that when M is found to be repeat–repetitive (line 19-20), M will be a suffix of the maximal common prefix of M_1 and M_2. In order to find the correct left–context, line 19 extracts this common suffix from M_{mp}.

For a while repetitive subfunction M_b within a context M_a–M_c, i.e. $M_a M_b^* M_c$ in the actual system, we assume that M_b and M_c do not have a nonempty common prefix, and M_b and M_a do not have a nonempty common suffix. Similarly, for a repeat repetitive subfunction M_b within a context M_a–M_c, i.e. $M_a M_b M_b^* M_c$ in the actual system, we assume that M_b and

```
1: M_mp = maximal_common_prefix(M_1, M_2);
2: M'_1 = remove_prefix(M_mp, M_1);
3: M'_2 = remove_prefix(M_mp, M_2);
4: M_s = maximal_common_suffix(M'_1, M'_2);
5: M''_1 = remove_suffix(M_s, M'_1);
6: M''_2 = remove_suffix(M_s, M'_2);
7: if M''_2 is not empty or M_mp is empty or M_s is empty then
8:     M_1 and M_2 do not infer a repetitive MSC
9: else
10:    M = basic_repetitive_MSC(M''_1);
11:    if M is empty then
12:        if M''_1 is a suffix of M_mp and M''_1 ≠ M_mp then
13:            M_p = remove_suffix(M_mp, M''_1);
14:            M_1 and M_2 infer M''_1 to be repeat–repetitive within the context
               M_p–M_s
15:        else
16:            M_1 and M_2 do not infer a repetitive MSC
17:        end if
18:    else if M is a suffix of M_mp and M ≠ M_mp then
19:        M_p = remove_suffix(M_mp, M);
20:        M_1 and M_2 infer M to be repeat–repetitive within the context
           M_p–M_s
21:    else
22:        M_1 and M_2 infer M to be while–repetitive within the context
           M_mp–M_s
23:    end if
24: end if
```

Fig. 2. Algorithm 1 – Checking if M_1 and M_2 infers an MSC M to be repetitive

M_c do not have a nonempty common prefix, and M_b and M_a do not have a nonempty common suffix. These assumptions are required to be able to infer the repetitive functionality uniquely. For example, when a while repetitive subfunction MM_b within the context M_a–MM'_c occurs in the actual system, there may be two observations to infer this repetitive subfunction in the form $M_aMM'_c$ and $M_a(MM'_b)\dots(MM'_b)MM'_c$. However based on such two observations, one can either infer MM'_b to be repetitive within the context M_a–MM'_c, or infer M'_bM to be repetitive within the context M_aM–M'_c.

We also assume that for a loop body M_b, there does not exist M such that $M_b = M^{(k)}$ for some $k \geq 2$.

Proposition 2: When $M_1 = M_a(M_b)^kM_c$ (for some $k \geq 2$) and $M_2 = M_aM_c$ are given as two observations to Algorithm 1 given in Figure 2, it

will infer M_b to be while repetitive within the context M_a–M_c.

Proposition 3: When $M_1 = M_a M_b (M_b)^k M_c$ (for some $k \geq 1$) and $M_2 = M_a M_b M_c$ are given as two observations to Algorithm 1 given in Figure 2, it will infer M_b to be repeat repetitive within the context M_a–M_c.

Proposition 4: If for two MSCs M_1 and M_2, Algorithm 1 given in Figure 2 infers M to be repetitive within the context M_p–M_s, then either:
(i) $M_1 = M_a M_b (M_b)^{(k)} M_c$ (for some $k \geq 1$) and $M_2 = M_a M_b M_c$; or
(ii) $M_1 = M_a (M_b)^{(k)} M_c$ (for some $k \geq 2$) and $M_2 = M_a M_c$
for some M_a, M_b, and M_c.

We explain four elementary functions and the algorithm basic_repetitive_MSC referenced in Algorithm 1 in the following subsections.

A. Finding the basic repetitive MSC

In this subsection, we explain how to check the existence of and find the basic repetitive MSC M of a given MSC M'.

Recall that $M|_i$ denotes the sequence of event labels of process P_i in the MSC M. If $M' = M^{(k)}$, it is obvious that for each process P_i, we have

$$M'|_i \quad = \quad \underbrace{M|_i M|_i \cdots M|_i}_{k \text{ times}}$$

In other words, we must see these repeating patterns in the events of the processes as well. Checking if a word w' consists of repetitions of another word w is a well–known and well–studied problem (e.g. see [19]) in pattern matching and text processing. If $w' = w^{(r)}$, then r is called *the power of* w *in* w' (we are only interested in integer powers, although the general theory of repetitions in words considers rational powers as well), and w is called a *root* of w'. If w cannot be written as a repetition of another word, then it is called as *primitive*. Linear time algorithms exist to find the primitive root of a word. Note that a word is always a root of itself with power 1.

Proposition 5: Given an MSC M', let r_i be the power of the primitive root of $M'|_i$, where $i \in [n]$, and let $r = \gcd(r_1, r_2, \ldots, r_n)$. M' has a basic repetitive MSC iff $r \geq 2$.

B. Functions on maximal common prefix–suffix, and prefix–suffix removal

Finding the maximal common prefix of two words is trivial, and based on scanning and comparing the elements of the words starting from the beginning and stopping when a difference is seen.

Finding the maximal common prefix of two MSCs is not as trivial as the case of words, since we require the prefix to be an MSC as well. Let M' and M'' be two MSCs. Consider again the sequence of event labels $M'|_i$ and $M''|_i$ of process P_i on M' and M''. Note that $M'|_i$ and $M''|_i$ are

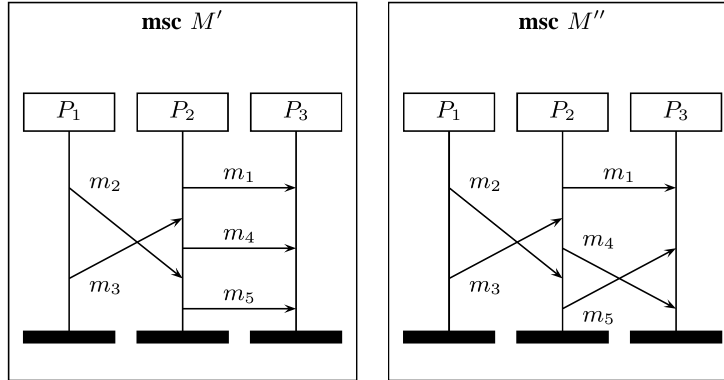

Fig. 3. Two MSCs

words. Let w_i be the maximal common prefix of the words $M'|_i$ and $M''|_i$. Note that the sequence of event labels $\langle w_i \mid i \in [n] \rangle$ does not necessarily characterize an MSC.

For example, when we consider the two MSCs M' and M'' given in Figure 3, we have

$$\begin{array}{lcl} w_1 & = & snd(1,2,m_2)snd(1,2,m_3) \\ w_2 & = & snd(2,3,m_1)rcv(2,1,m_3)snd(2,3,m_4)rcv(2,1,m_2)snd(2,3,m_5) \\ w_3 & = & rcv(3,2,m_1) \end{array}$$

Obviously, the sequence $\langle w_1, w_2, w_3 \rangle$ does not characterize an MSC. However, this problem can be solved by removing some of the suffixes of w_i's. Recall the procedure explained shortly in Section II, for building an MSC based on a sequence of event labels. This procedure can be adapted to eliminate the problematic suffixes in w_i's while finding the maximal common prefix in the following way. Initially, all the event labels in w_i's are unmarked. Then perform a scan on each w_i starting from the beginning. For each send event label of the form $snd(i,j,a)$ in w_i, find the first unmarked event label of the form $rcv(j,i,a)$ in w_j. If such an unmarked event could be found in w_j, mark both $snd(i,j,a)$ and $rcv(j,i,a)$ instances under consideration, and proceed to the next send event in w_i. When we mark two such events, they are called as marking pairs. It is necessary to remember this association since, if and when one of them is removed, the other will also be removed in the second phase of this procedure. If no such an unmarked event could be found in w_j, then leave $snd(i,j,a)$ and all the remaining events in w_i as unmarked. After this marking phase, we have an iterative suffix removal phase. For each w_i, the suffix of w_i that starts with the first unmarked event label is removed. Note that, some of the event

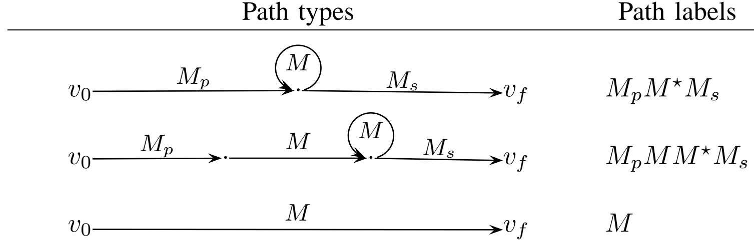

Fig. 4. Three different path types

labels in such a suffix may be marked. While removing such a marked event label, the mark of its marking pair (which must also be present in some w_j as marked) is removed. This iteration continues until all the event labels in all the w_i's are marked. The remaining event labels characterize an MSC which is the maximal common prefix. Finding the maximal common suffix of two MSCs can be performed in a similar way, by adapting the approach in the procedure explained above.

Given an MSC M and a prefix M' of M, removing the prefix M' can be performed by removing the event label sequence $M'|_i$ from the first part of $M|_i$ for each process P_i. Similarly, the removal of a suffix M' of M would be performed by removing the event label sequence $M'|_i$ from the last part of $M|_i$ for each process P_i.

C. Forming the final MSC graph

Algorithm 2 given in Figure 5 is used to produce an MSC graph based on a given set of MSCs $\mathbb{M}$. It has two phases. The first phase considers every pair of MSCs M_1 and M_2 in $\mathbb{M}$ and checks whether M_1 and M_2 infer a repetitive MSC. The output of the first phase is an MSC graph with a special structure. For each pair of MSCs that infer a repetitive MSC, a separate subgraph, that is disjoint with the rest of the graph except at v_0 and v_f, is created. Such a subgraph has a different structure depending on whether the inferred MSC is while— or repeat–repetitive, which are shown in Figure 4 at the top and in the middle, respectively. If an MSC M does not infer a repetitive MSC by pairing with another MSC, then a subgraph which consists of only (v_0, M, v_f) is created, as shown at the bottom in Figure 4. We will call these subgraphs *paths*. These paths will be referenced using their labels which are constructed by concatenation of the labels of the edges in the paths. The label M of a self-loop edge in such a path will be represented by using $M^\star$ in the path label. The second column of Figure 4 gives the label template associated with each path type.

As an example for the execution of the first phase, let us suppose that initially we have three MSCs
$M_a = M_1M_2M_2M_3M_4M_5$,
$M_b = M_1M_2M_2M_3M_4M_4M_5$,
$M_c = M_1M_3M_4M_5$.
M_a and M_b infer M_4 to be repeat–repetitive within the context $M_1M_2^{(2)}M_3$–M_5. Hence
$M_d = M_1M_2^{(2)}M_3M_4M_4^{\star}M_5$ is inferred as a path in G. Similarly, M_a and M_c infer M_2 to be a while-repetitive within the context of M_1–$M_3M_4M_5$. So
$M_e = M_1M_2^{\star}M_3M_4M_5$ is inferred as a path in G.

Note that the subgraph generated from M_1 and M_2 is guaranteed to represent both M_1 and M_2, i.e. the language of M_1 and M_2 are included in the language of the generated subgraph. Thus, M_1 and M_2 are marked for deletion since we generated a new path from them. However, if for an MSC M_1, there does not exist an MSC M_2 which infers a repetitive MSC, then M_1 will simply be put into G and left unmarked. At the end of the first phase, there is no other repetitive subfunction left to be inferred. However, all possible relative positions of these repetitive subfunctions must be represented in the final MSC graph G' which is constructed in the second phase of Algorithm 2.

The MSC graph G constructed by the first phase can be nondeterministic. In other words, there may be two different paths with a nonempty common prefix. In general, there is no guarantee that the system state is the same after the execution of the same prefix along these different paths, since the observations only give the message exchanges, and the local actions within the processes are hidden from the observer. There needs to be some evidence in the observations that allow some paths to be merged. Especially when a given observation M is used in the generation of two different paths p_1 and p_2 with labels M_1 and M_2 respectively, the execution of p_1 and p_2 also corresponds to the execution of M. Hence the system state along p_1 and p_2 that correspond to the execution of M must be same, and therefore p_1 and p_2 need to be merged.

The following is clear from Definition 2 and the construction of graph G.

Proposition 6: Algorithm 2 in Figure 5 infers all the repetitive subfunctions for which the required evidence is given in $\mathbb{M}$.

The second phase of Algorithm 2 performs the merging of paths using Algorithm 3 given in Figure 6. Since we need to know the actual observations from which a path p is generated, the first phase of the algorithm associates this information to the generated path p by $src(p)$.

In the example above, after the first phase, MSCs M_a, M_b and M_c are marked and thus removed, and we have the paths corresponding to M_d and

1: /* phase 1: infer repetitive subfunctions and form G*/
2: initially all the MSCs in $\mathbb{M}$ are unmarked
3: generate the initial and the final nodes v_0 and v_f in G
4: **for** each pair $M_1, M_2 \in \mathbb{M}$ **do**
5:   **if** M_1 and M_2 infers an MSC M to be repetitive within a context M_p–M_s **then**
6:     mark both M_1 and M_2
7:     **if** M is while–repetitive **then**
8:       generate a new path p in G given below, where v is a new node $p = \{(v_0, M_p, v), (v, M, v), (v, M_s, v_f)\}$
9:     **else**
10:       generate a new path p in G given below, where v and v' are new nodes $p = \{(v_0, M_p, v), (v, M, v'), (v', M, v'), (v', M_s, v_f)\}$
11:     **end if**
12:     let $src(p) = \{M_1, M_2\}$
13:   **end if**
14: **end for**
15: **for** each unmarked MSC M **do**
16:   generate a new path $p = \{(v_0, M, v_f)\}$
17:   let $src(p) = \{M\}$
18: **end for**
19: /* phase 2: merge paths and form G' */
20: let G' be an empty graph
21: obtain a partition Π of the set of paths such that two paths p, p' are in the same subset P of Π iff $\exists$ a sequence of paths $p_1, p_2, \ldots, p_k$ such that $p = p_1$, $p' = p_k$, and for $1 \leq i < k$, $src(p_i) \cap src(p_{i+1}) \neq \emptyset$
22: **for all** $P \in \Pi$ **do**
23:   insert $merge(P)$ into G'
24: **end for**

Fig. 5. Algorithm 2 – Building the MSC graph based on a set of MSCs $\mathbb{M}$

M_e added to G. In the second phase, there are two paths M_d and M_e in G. Since the sources of these two paths have a nonempty intersection, they will be in the same partition, which is actually the only partition of paths in this example. Hence, the for loop at lines 22–24 in Algorithm 2 will iterate only once, and will insert the merging of M_d and M_e into G'.

In Algorithm 3, we consider every path as a set of three edges : (v_0, M_p, v_1), (v_1, M, v_1) and (v_1, M_s, v_f). M_p, M and M_s will be referred to as the prefix, loop and the suffix labels of the path, respectively. Note that, if the path is generated for a repeat repetitive subfunctionality, then we consider

1: let p be a path in P whose prefix label is the shortest among other paths in P
2: let $p_m = p$
3: let $src(p_m) = src(p)$
4: let $P = P - \{p\}$
5: **while** P is not empty **do**
6: let p be a path in P such that $src(p) \cap src(p_m) \neq \emptyset$ and the prefix label of p is the shortest among other such paths in P.
7: $src(p_m) = src(p_m) \cup src(p)$
8: $P = P - \{p\}$
9: trace the prefix label M_p of p in p_m (by skipping ε edges)
10: **if** during this trace M_p ends in the middle of an edge (v_1, M', v_2) in p_m **then**
11: remove the edge (v_1, M', v_2)
12: insert a new node v in p_m
13: insert the edges (v_1, M'_1, v) and (v, M'_2, v_2) such that $M' = M'_1 M'_2$, and trace of M_p ends in v
14: insert a new edge (v, M, v) in p_m where M is the label of the self loop edge of p
15: **else**
16: $\{$the trace ends at an already existing node in $p_m\}$
17: let v be the node at which the trace of M_p has ended
18: let (v, M', v') be the edge which is not used during the trace of M_p
19: remove the edge (v, M', v')
20: insert a new node v'' in p_m
21: insert a new edge (v'', M', v')
22: insert a new edge (v, ε, v'')
23: insert a new edge (v'', M, v'')
24: **end if**
25: **end while**
26: return (p_m)

Fig. 6. Algorithm 3 – Merging paths in a partition P

the first iteration of the loop as embedded in the prefix label.

Note that, G' produced by the second phase will effectively have the loops in G placed in their relative order, which also includes placing a loop into another, i.e. nesting of the loops. Deciding the relative orders of the loops in different paths in the merged path is based on having a common observation used in the generation of these paths. For instance, see the example given above.

Note that, Algorithm 3 depends on tracing M_p in p_m (which is actually an MSC graph) accumulated so far. Given an MSC M and an MSC graph G, it is known that deciding if $M \in L(G)$ is NP–complete [20], [21]. However, in our case this trace can be found in linear time. This is due to the fact each node in p_m corresponds to the start of a loop. Since we never allow two loops to start at the same node, for each node in p_m there will be exactly two outgoing edges. One of these edges will be corresponding to the (beginning of the) loop's body, and the other will be corresponding to the (beginning of the) loop's right–context. Since for a loop, the body and the right–context of the loop do not have a non–empty common prefix, we will always have a unique edge to proceed with the trace.

V. Conclusion

We have presented an algorithm to derive an MSC graph G' from a given set of observations of an existing implementation of a concurrent system. This algorithm is based on inferring repetitive subfunctions from a given set of observations. The language of the MSC graph G' derived consists of all the given observations and the inferred observations, in which the loops and their relative positions are all explored. And thus, the language of the MSC graph G' is a design representation of the existing system.

As a proof of concept, we have developed a prototype tool to implement the technique introduced in this paper. The tool accepts a set of observations of an existing implementation and produces an MSC graph by applying the algorithms proposed in this work.

An interesting problem that remains open is the following. When the evidences of some loops are missing in the given set of observations, generated subgraphs may provide these missing evidences. For example, assume that initially we have three MSCs $M_a = M_1M_2^{(2)}M_3M_5$, $M_b = M_1M_2^{(2)}M_3M_4^{(2)}M_5$, and $M_c = M_1M_3M_4^{(3)}M_5$. M_a and M_b infers M_4 to be while–repetitive within the context $M_1M_2^{(2)}M_3$–M_5. Hence $M_d = M_1M_2^{(2)}M_3M_4^{\star}M_5$ is generated. Although M_c does not infer any repetitive subfunctionality with M_a or M_b, it infers M_2 to be while–repetitive within the context M_1–$M_3M_4^{(3)}M_5$ when it is considered together with $M_e = M_1M_2^{(2)}M_3M_4^{(3)}M_5$ which is obtained by instantiating $\star$ by 3 in M_d. However, these new repetitive subfunctions must be confirmed by the observer. Hence the question is, can and how an MSC graph, which is a design representation of the existing system, be generated under such missing evidences. It will also be interesting to consider the effects of a) the number of occurrences of repetitive subfunctions in the given set of observations to be a fixed value, and b) some apriori knowledge of the structure of the communicating processes on the derivation of an MSC graph.

ACKNOWLEDGMENT

The authors wish to acknowledge many useful discussions with Jessica Chen. We thank Hasan Ural for his crucial input and constant help. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, under grant OGP 976.

REFERENCES

[1] ITU Telecommunication Standardization Sector, ITU-T Recommendation Z.120. Message Sequence Charts (MSC96) (May 1996).
[2] E. Rudolph, P. Graubmann, J. Gabowski, Tutorial on message sequence charts, Computer Networks and ISDN Systems–SDL and MSC 28.
[3] R. Alur, G. J. Holzmann, D. Peled, An analyzer for message sequence charts, Software Concepts and Tools 17 (2) (1996) 70–77.
[4] H. Ben-Abdallah, S. Leue, Syntactic detection of progress divergence and non–local choice in message sequence charts, in: 2nd TACAS, 1997, pp. 259–274.
[5] R. Alur, M. Yannakakis, Model checking of message sequence charts, in: 10th International Conference on Concurrency Theory, Springer Verlag, 1999, pp. 114–129.
[6] R. Alur, K. Etessami, M. Yannakakis, Inference of message sequence charts, IEEE Transactions on Software Engineering 29 (7) (2003) 623–633.
[7] M. Lohrey, Safe realizability of high–level message sequence charts, in: 13th International Conference in Concurrency Theory, CONCUR 2002, 2002, pp. 177–192.
[8] E. Chikofsky, J. Cross, Reverse engineering and design recovery, IEEE Software 7 (1990) 13–17.
[9] D. Lee, K. Sabnani, Reverse engineering of communication protocols, in: IEEE ICNP'93, 1993, pp. 208–216.
[10] H. Ural, H. Yenigun, Towards design recovery from observations, in: FORTE 2004, LNCS 3235, 2004, pp. 133–149.
[11] G.-V. Jourdan, H. Ural, H. Yenigun, Recovering the lattice of repetitive sub-functions, in: ISCIS 2005, LNCS 3733, 2005, pp. 956–965.
[12] G.-V. Jourdan, H. Ural, S. Wang, H. Yenigün, Recovering repetitive sub-functions from observations, in: FORTE '07: Proceedings of the 27th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 35–49.
[13] K. Koskimies, E. Makinen, Automatic synthesis of state machines from trace diagrams, Software–Practice & Experience 24 (1994) 643–658.
[14] M. Rajagopal, R. E. Miller, Synthesizing a protocol converter from executable protocol traces, IEEE Transactions on Computers 40 (1991) 487–499.
[15] P. Zafiropulo, C. West, H. Rudin, D. D. Cowan, D. Brand, Towards analyzing and synthesizing protocols, IEEE Transactions on Communications 28 (1980) 651–660.
[16] K. Saleh, A. Boujarwah, Communications software reverse engineering: A semi–automatic approach, Information & Software Technology 38 (1996) 379–390.
[17] K. Saleh, R. L. Probert, I. Manonmani, Recovery of communication protocol design from protocol execution traces, in: IEEE ICECCS'96, 1996, pp. 265–272.
[18] X. J. Chen, H. Ural, Construction of deadlock–free designs of communication protocols from observations, The Computer Journal 45 (2002) 162–173.
[19] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[20] R. Alur, K. Etessami, M. Yannakakis, Realizability and verification of MSC graphs, in: Automata, Languages and Programming, 28th International Colloquium, ICALP, LNCS 2076, 2001, pp. 797–808.
[21] A. Muscholl, D. Peled, Z. Su, Deciding properties of message sequence charts, in: Foundations of Software Science and Computation Structures, LNCS 1378, 1998, pp. 226–242.

APPENDIX

A. Proofs

Proof of Proposition 1:
The proof follows from the fact that MSCs are fully characterized by their projections onto the event labels in the processes as explained above. ■

Proof of Proposition 2:
Based on the assumption that a loop body cannot have a nonempty common prefix with the right–context of the loop, M_b and M_c cannot have a nonempty common prefix.

Therefore, the maximal common prefix of M_1 and M_2 is M_a, hence the algorithm will find $M_{mp} = M_a$, $M_1' = (M_b)^k M_c$, $M_2' = M_c$. The maximal common suffix of M_1' and M_2' is then obviously M_c, and hence the algorithm will find $M_s = M_c$, $M_1'' = (M_b)^k$, and $M_2'' =$ empty.

Then the algorithm will arrive at line 10, and it will find $M = M_b$ since M_b, a loop body, cannot have a basic repetitive MSC. The algorithm will then check if M_b is a suffix of M_a or not at line 18. However, due to our assumption that a loop body and the left–context of the loop do not have a non–empty common suffix, this check will fail. Hence the algorithm will reach to line 22 and infer M_b to be while repetitive within the context M_a–M_c, i.e. $M_a(M_b)^\star M_c$. ■

Proof of Proposition 3:
Based on the assumption that a loop body cannot have a nonempty common prefix with the right–context of the loop, M_b and M_c cannot have a nonempty common prefix. Therefore, the maximal common prefix of M_1 and M_2 is $M_a M_b$, hence the algorithm will find $M_{mp} = M_a M_b$, $M_1' = (M_b)^k M_c$, $M_2' = M_c$. The maximal common suffix of M_1' and M_2' is then obviously M_c, and hence the algorithm will find $M_s = M_c$, $M_1'' = (M_b)^k$, and $M_2'' =$ empty.

Then the algorithm will arrive at line 10. There may be two cases depending on k:
Case 1 ($k = 1$): Then the algorithm will find M as empty. However, since $M_1'' = M_b$ is a suffix of $M_{mp} = M_a M_b$, it will infer (at line 14) M_b to be repeat repetitive within the context M_a–M_c, i.e. $M_a M_b (M_b)^\star M_c$.
Case 2 ($k > 1$): Then the algorithm will find $M = M_b$, and since it is a suffix of $M_{mp} = M_a M_b$, it will infer (at line 20) M_b to be repeat repetitive within the context M_a–M_c, i.e. $M_a M_b (M_b)^\star M_c$. ■

Proof of Proposition 4:
Algorithm 1 will infer a repetitive MSC only at lines 14, 20, 22. Line 10

has to be reached for all these inference lines. When the algorithm reaches to line 10, we must have: $M_1 = M_{mp}M_1''M_s$, and $M_2 = M_{mp}M_s$.
Case 1 (inference is made at line 14): In order to reach line 14 from line 10, M_1'' must not have a basic repetitive MSC, but it must be a suffix of M_{mp}. Hence in this case $M_1 = M_pM_1''M_1''M_s$, and $M_2 = M_pM_1''M_s$, which is case (i) with $k = 1$ given in the proposition.
Case 2 (inference is made at line 20): In order to reach line 20 from line 10, we must have $M_1'' = M^{(k)}$ for some $k \geq 2$, and $M_{mp} = M_pM$. Hence in this case $M_1 = M_pM(M)^{(k)}M_s$, and $M_2 = M_pMM_s$, which is case (i) with $k > 1$ given in the proposition.
Case 3 (inference is made at line 22): In order to reach line 20 from line 10, we must have $M_1'' = M^{(k)}$ for some $k \geq 2$. Hence in this case $M_1 = M_p(M)^{(k)}M_s$, and $M_2 = M_pM_s$, which is case (ii) with $k \geq 2$ given in the proposition. ■

Proof of Proposition 5:
Assume that M' has a basic repetitive MSC, i.e. $M' = M^{(k)}$ for some M and $k \geq 2$. Since the projections onto the processes must be the same, $M'|_i = M|_i^{(k)}$. Therefore, $r_i = kr_i'$ where r_i' is the power of the primitive root of $M|_i$ (note that $M|_i$ is not necessarily primitive). Therefore k is a common divisor of $r_1, r_2, \ldots, r_n$, and hence $r = gcd(r_1, r_2, \ldots, r_n) \geq k \geq 2$.
For the proof of the reverse direction, assume that $r \geq 2$. Consider an MSC M, where $M|_i$ is the first $|E_i'|/r$ event labels in $M'|_i$. Note that $M' = M^{(r)}$, since processwise projections are the same, and also note that M formed in this way must be a (well–formed) MSC, since otherwise M will have a send (receive) event which is not matched by a receive (send) event, which in turn would imply that M' also has unmatched events, hence M' would not be well–formed. It remains to show that M does not have a basic repetitive MSC. In fact this must be true, since if $M = M''^{(r')}$ for some $r' \geq 2$, then we must have $M' = M''^{(rr')}$. However in this case, rr', which is strictly greater than r would be a common divisor of $r_1, r_2, \ldots, r_n$, contradicting with the fact that $r = gcd(r_1, r_2, \ldots, r_n)$. ■

B. Analysis of the Algorithms

1) Maximal common prefix and maximal common suffix of two MSCs:
Analysis of Algorithm 4: At line 2, given two words w' and w'', it takes $O(min(|w'|, |w''|))$ time to find the maximal common prefix of w' and w''. Since this line is iterated for each process, the total time taken by the loop on lines 1–3 can be considered as

$$\begin{aligned}\sum_{P_i \in \mathbb{P}} O(\min(M'|_i, M''|_i)) &= O(\sum_{P_i \in \mathbb{P}} \min(M'|_i, M''|_i)) \\ &= O(\min(|M'|, |M''|))\end{aligned}$$

```
1: for all P_i do
2:    w_i = maximal_common_prefix (M'|_i, M''|_i)
3: end for
4: let G be an empty graph
5: for all w_i do
6:    for all events e in w_i do
7:       insert a node n_e into G as an unmarked node
8:    end for
9: end for
10: for all w_i do
11:    for all events e in w_i do
12:       if e is not the last event in w_i then
13:          insert the edge n_e → n_e' in G where e' is the event in w_i that
             immediately follows e
14:       end if
15:    end for
16: end for
17: for all w_i do
18:    for all events e in w_i in the order they appear in w_i do
19:       if e = snd(i, j, a) then
20:          let e' be the first event in w_j such that e' = rcv(j, i, a) and n_e'
             is unmarked
21:          if e' can be found then
22:             mark n_e and n_e'
23:             insert the edges n_e → n_e' and n_e' → n_e in G
24:          end if
25:       end if
26:    end for
27: end for
28: unmark all the nodes in G that are reachable from unmarked nodes
29: remove the unmarked nodes in G
```

Fig. 7. Algorithm 4 – Finding maximal common prefix of M' and M''

Between the lines 4–27, we form a graph which actually corresponds to the partial MSC represented by w_i's. The only difference to a standard partial MSC is that, the message arrows are bidirectional. Forming this graph can be performed in $O((\sum w_i)^2) = O((\min(|M'|, |M''|))^2)$ time.

Line 28 is a standard graph reachability problem which can be solved in $O(V + E)$ time, where V and E are number of nodes and edges in the graph. In our graph, we have $\min(|M'|, |M''|)$ nodes. Since each node has at most two outgoing edges (one going along the process line, and

1: **for all** P_i **do**
2: w_i = primitive_root $(M|_i)$
3: $r_i = |E_i|/|w_i|$
4: **end for**
5: $r = \text{gcd}(\{r_i\})$
6: **if** $r \geq 2$ **then**
7: **for all** P_i **do**
8: $M'|_i$ = first $\frac{|E_i|}{r}$ events of $M|_i$
9: **end for**
10: **else**
11: **return** empty MSC
12: **end if**

Fig. 8. Algorithm 5 – Finding basic repetitive MSC M' of a given MSC M

1: **for all** P_i **do**
2: w_i = remove_prefix($M_p|_i$, $M|_i$);
3: **end for**

Fig. 9. Algorithm 6 – Remove a prefix M_p of an MSC M

optionally – if it is matched – another one going to another process line), $E = O(\min(|M'|, |M''|))$. Therefore, line 28 takes $O(\min(|M'|, |M''|))$ time as well.

Line 29 can also be handled in $O(\min(|M'|, |M''|))$ time.

The overall execution time for Algorithm 4 is therefore, $O((\min(|M'|, |M''|))^2)$.

Note that a very similar algorithm can be used to find the maximal common suffixes. The only difference would be to take the maximal common suffixes at line 2, and the edges inserted at line 13 would be reversed. Therefore finding maximal common suffix of two MSCs M' and M'' takes $O((\min(|M'|, |M''|))^2)$ time too.

2) Basic repetitive MSC of a given MSC: Analysis of Algorithm 5: Finding the primitive root of a word w at line 2 takes $O(|w|)$ time. The total time spent between lines 1–4 will be $O(|M|)$ (where $|M|$ is the total number of events in the MSC M), since the loop will be iterated for each process. In the worst case, $r = 2$, hence we will need to copy $|M|/2$ events in total at line 8. Hence the total time for Algorithm 5 is $O(|M|)$.

3) Removing a prefix and a suffix of an MSC: Analysis of Algorithm 6: Since $M_p|_i$ events will be removed from the beginning of $M|_i$, the total time spent in this algorithm will be $O(|M_p|)$.

Note that, removing a suffix can be performed by using a very similar algorithm. The only difference will be that we will remove $|M_p|_i|$ elements at the end of $M|_i$. Hence it will also take $O(|M_p|)$ time.

1: $M = M_{in}$, $v = v_{in}$
2: **while** M is not empty **do**
3: let $v \stackrel{M'}{\rightarrow} v'$ be the outgoing edge from v such that M and M' has a common prefix
4: **if** $M' == M$ **then**
5: **return** v'
6: **else if** M' is a prefix of M **then**
7: $M = M - M'$
8: **else**
9: /* M is a prefix of M' */
10: insert a new edge v''
11: remove the edge $v \stackrel{M'}{\rightarrow} v'$
12: insert the edged $v \stackrel{M}{\rightarrow} v''$ and $v'' \stackrel{M'-M}{\rightarrow} v'$
13: **return** v''
14: **end if**
15: **end while**

Fig. 10. Algorithm 7 – Given an MSC graph p_m, a node v_{in} in p_m, and an MSC M_{in}, find the node v in p_m whose label is equal to M

4) Analysis of Algorithm 1: Note that we assume $|M_1| \geq |M_2|$ for Algorithm 1
Line 1: $O((\min(|M_1|, |M_2|))^2) = O(|M_2|^2)$
Line 2: $O(|M_{mp}|) = O(\min(|M_1|, |M_2|) = O(|M_2|)$
Line 3: $O(|M_{mp}|) = O(\min(|M_1|, |M_2|) = O(|M_2|)$
Line 4: $O((\min(|M_1'|, |M_2'|))^2) = O((\min(|M_1|, |M_2|))^2) = O(|M_2|^2)$
Line 5: $O(|M_s|) = O(\min(|M_1'|, |M_2'|) = O(\min(|M_1|, |M_2|) = O(|M_2|)$
Line 6: $O(|M_{mp}|) = O(\min(|M_1|, |M_2|) = O(|M_2|)$
Line 10: $O(|M_1''|) = O(|M_1|)$
Overall: $O(|M_2|^2 + |M_1|)$

5) Tracing an MSC within an MSC graph: **Analysis of Algorithm 7:** It is known that the problem of "deciding whether a given MSC M is in the language of a given MSC graph" is NP–complete. However, since we assume that at each branching point there will be a single branch that can be followed, the complexity of Algorithm 7 is $O(|M_{in}|)$.

6) Analysis of Algorithm 3: Let $|P|$ be the number of paths in the subset of the partition being handled.
Let e_P be the total size of the paths (the total number of events in all the paths in P).
Line 1: $O(|P|)$
Line 2,3,4: constant
Line 6: For all iterations of the enclosing while loop, it takes $O(|P|^2)$ time

Line 9: $O(|M_p|)$, where $|M_p|$ is the size of M_p. Since the line will be executed for all the MSCs in the subset of the partition being handled, it takes time linear with the total size of the MSCs, i.e. $O(e_P)$.
Other lines in the while loop: constant
Overall: $O(|P|^2 + e_P)$

Note that the Algorithm 3 could be modified and its complexity reduced by changing the way we construct the graph. Under our assumptions, it is possible to create a total order on the paths, from the shortest to the longest, each path having a strictly shorter prefix than its successor in the order. Thus, instead of tracing the prefix from the beginning each time we select a new path (line 9) we could simply continue our tracing from the current node by selecting line 6 the next path in this total order. This improvement would however not change to final complexity of the system, as shown in Appendix B7.

7) Analysis of Algorithm 2: **Lines 1–14:** Assume that we have sorted the MSCs with respect to their sizes in $O(k \lg k)$ time (where $k = |\mathbb{M}|$), hence we have

$$|M_1| \geq |M_2| \geq \cdots \geq |M_k|$$

For line 5, we will be executing Algorithm 1 for each combination of MSCs in $\mathbb{M}$. Hence the total time taken at line 5 will be

$$O(\textstyle\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} (|M_j|^2 + |M_i|)) \quad = \quad O(\sum_{i=1}^{k-1} (k-i)|M_i| + \sum_{i=1}^{k-1} i|M_{i+1}|^2)$$

Other lines within the for loop between line 4 and line 14 takes constant time, hence the total time taken by this for loop is:

$$O(k^2 + \sum_{i=1}^{k-1} (k-i)|M_i| + \sum_{i=1}^{k-1} i|M_{i+1}|^2) = O(k^2 + ke_{\mathbb{M}}^2)$$

where $e_{\mathbb{M}}$ is the total number of events in all the MSCs in $\mathbb{M}$.

Line 21 can be handled by producing a graph in which we have a node for each path, and there is an undirected edge between the nodes of two paths p_1 and p_2 iff $src(p_1) \cap src(p_2) \neq \emptyset$. Then, the partition wanted is the set of connected components. With this approach, line 21 can be handled in quadratic time in the number of paths. Since, under our assumptions, the number of paths is comparable to k, line 21 will be done in $O(k^2)$.

The for loop on the last three lines makes a call to Algorithm 3, whose complexity for a partition subset P is $O(|P|^2 + e_P)$, where $|P|$ is the number of paths in P, and e_p is the total number of events in all the paths in P. Therefore the total time will be

$$\begin{aligned} \textstyle\sum_{P \in \Pi} O(|P|^2 + e_P) &= O(\textstyle\sum_{P \in \Pi} |P|^2 + \sum_{P \in \Pi} e_P) \\ &= O(k^2 + e_{\mathbb{M}}) \end{aligned}$$

Overall:
Initial sorting: $O(k \lg k)$
Lines 4–14: $O(k^2 + ke_{\mathbb{M}}^2)$
Lines 15–18: $O(k)$
Line 21: $O(k^2)$
Lines 22-24: $O(k^2 + e_{\mathbb{M}})$
And finally: $O(k \lg k + k^2 + ke_{\mathbb{M}}^2 + k + k^2 + k^2 + e_{\mathbb{M}}) = O(k^2 + ke_{\mathbb{M}}^2)$