Towards Design Recovery from Observations

Hasan Ural[1] and Hüsnü Yenigün[2]

[1] School of Information Technology and Engineering (SITE)
University of Ottawa, 800 King Edward Avenue
Ottawa, Ontario, Canada, K1N 6N5
[2] Faculty of Engineering and Natural Sciences
Sabancı University, Tuzla, Istanbul, Turkey 34956

Abstract. This paper proposes an algorithm for the construction of an MSC graph from a given set of actual behaviors of an existing concurrent system which has repetitive subfunctions. Such a graph can then be checked for safe realizability and be used as input to existing synthesis techniques.

1 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] (revised version appeared as [7]). 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. [7] 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

D. de Frutos-Escrig and M. Núñez (Eds.): FORTE 2004, LNCS 3235, pp. 133–149, 2004.

time [7], that of a given bounded MSC graph is in EXPSPACE [8], which is later shown to be EXPSPACE–complete [9].

Design representations are helpful not only for implementing software systems, but also for software maintenance, e.g. to detect and eliminate errors in a system, to extend the capability of a system, or to adapt a system to different operating environments. Further, the developers of a new software system whose functionality contains some of the functionality of an existing system can benefit by using the related part of the design of the existing system. However, up-to-date or complete designs of many existing systems may not always be available.

One of the aims of the reverse engineering [10, 11] 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.

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.

The rest of the paper is organized as follows: Section 2 introduces the terminology and notation used throughout the paper. Section 3 gives the formal definition of the problem. Section 4 presents the construction of an MSC graph from a given set of observations. Section 5 discusses some open problems and gives the concluding remarks.

2 Preliminaries

The notation we will be using is directly adopted from [7]. A concurrent system $\mathcal{P}$ is a set of processes $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, 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 [7]. A Σ–labeled MSC M for a concurrent system $\mathcal{P}$ 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.

[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.

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

Let $|E|$ denote the cardinality of the set E. A permutation of the events E of an MSC as $e_1 e_2 \dots e_{|E|}$ 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 $e_1 e_2 \dots e_{|E|}$ of M such that $w = l(e_1) l(e_2) \dots l(e_{|E|})$. 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 [7], given a well–formed and complete word 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 [7]. Roughly, the procedure is to scan each $M|_i$ starting from the beginning. During this scanning, a send event with a label $snd(i, j, a)$ is matched with the first not–yet–matched receive event with a label $rcv(j, i, a)$ in $M|_j$.

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

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

Consider the visual representation of an MSC M and imagine that we draw a line through M by crossing each process line exactly once, and without crossing any message arrows. Such a line divides M into two parts M_p (the part above the cutting line) and M_s (the part below the cutting line). M_p and M_s can be shown to be MSCs again. In fact, M_p and M_s are, what we are going to call below, a prefix of M and a suffix of M, respectively.

Formally, 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 $\forall i \in [n]$, $\leq'_i = \leq_i |_{E'}$.

The imaginary cutting line mentioned in the intuitive explanation above is the one that crosses P_i's line right below (resp. right above) the largest (resp. the smallest) event $e \in E'_i$ with respect to the total order $\leq_i$.

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 $M_p M_s$. 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 $T \subseteq V \times \mathbb{M} \times V$ gives the edges between the nodes with the labels from $\mathbb{M}$. 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 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 $(v, M, v') \in T$.

3 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

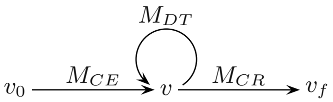


Fig. 1. An example MSC Graph

(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:

$$\text{CE, DT, DT, ..., DT, CR} \tag{1}$$

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 [12, 13, 14, 15, 16, 17]. 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 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 loops. In this paper, we will introduce a method that will help recover designs *with* loops.

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 2, 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 $\mathcal{M} = \{msc(w) \mid w \in O\}$, and we will consider an observation to be an MSC from now on unless stated otherwise.

In our view of a function being composed of subfunctions, we also consider a subfunction to be specified by an MSC. This can be justified by considering that all the messages sent within a subfunction will be consumed within the same subfunction. In an observation M, which is given as a single MSC, the MSCs corresponding to the subfunctions are not apparent. However, our purpose is not to identify the MSCs of all the subfunctions one by one, but rather to identify those MSCs that correspond to repetitive subfunctions.

Note that, if there are any loops in the design of Imp, and if an observation $M \in \mathcal{M}$ is generated by more than one iteration of some loop, then there must be some repeated pattern in M where the pattern being repeated is generated by the iterations of the loop. However, the converse is not correct in general, i.e., a repeated pattern seen in an observation is not necessarily due to a loop.

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.

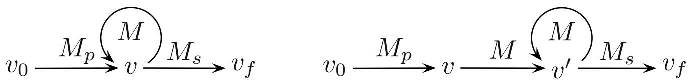
Suppose that M is found to be while–repetitive under the evidence of two observations M_1 and M_2 with the prefix M_p and suffix M_s, and suppose that M is indeed generated by a loop in the design. Then the state right before the execution of M, and right after the execution of M are the same. Hence, any MSC in the form $M_p M^\star M_s$ must be realizable by Imp. A similar argument can be applied to show that when M is found to be repeat-repetitive, any MSC in the form $M_p M^+ M_s$ must be realizable by Imp.

Since we assume that Imp does not go through the initial system state during the execution of an observation, M_p and M_s in Definition 2 must not be empty. This can be justified by the following observation. If M is repetitive, then the state just before and just after an iteration of M are the same. If M_p is empty, then M starts its execution from the initial state, since the observations start from the initial state. After the first iteration of M, the system will again be at the initial state. However, this is the definition of a function in our setting, hence M and M_s must be given as separate observations. Similarly, if M_s is empty, then the state after an iteration, hence before an iteration of M is the initial state, since observations are assumed to end at this state. In this case, M_p and M must be given as separate observations.

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. Further, each repetitive subfunction is inferred only once using the given observations. Moreover, in order to establish the relative ordering of two or more loops $l_1, l_2, \ldots, l_n$ in an MSC graph, $k \geq 2$ iterations of at least two loops l_i and l_j need to be given in the same observation such that the relative ordering of a pair of loops l_i and l_k can be determined from the relative ordering of two pairs of loops l_i and l_j, and l_j and l_k, $1 \leq i, j, k \leq n$, and by transitivity.

4 Problem Solution

When a pair of MSCs M_1 and M_2 is identified within the given set $\mathcal{M}$, 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).



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 $M = M_pM_s$. 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 last part 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 $M_1 = M_pM^{(k)}M_s$ and either $M_2 = M_pM_s$ or $M_2 = M_pMM_s$. 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, however, 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 4.1. 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 $M_1 = M_pM^{(2)}M_s$ and $M_2 = M_pMM_s$. In this specific case, the maximal common prefix M_{mp} would be M_pM. 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 common 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}.

Note that, there may be multiple ways for dividing M_1 and M_2 to identify M_p, M and M_s. For example, let $M_1 = M_aM_bM_cM_bM_cM_bM_d$ and let $M_2 = M_aM_bM_d$. In this case, it is possible to infer M_bM_c as while–repetitive within the context M_a– M_bM_d. However, it is also possible to infer M_cM_b as

```
 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 $M_{mp}$ 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 $M_{mp}$ 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

while–repetitive within the context $M_a M_b$–M_d. By convention, we prefer to keep preamble of the loop as long as possible, hence use the latter alternative. Note that, this is only a convention as both $M_a M_b (M_c M_b)^\star M_d$ and $M_a (M_b M_c)^\star M_b M_d$ have the same language. Algorithm 1 implements this convention by extracting the maximal common prefix of M_1 and M_2 at line 1.

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

4.1 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

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

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 [18]) 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 2. *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 $r \geq 2$.*

Proof. 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 process wise projections are the same. 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 $r = gcd(r_1, r_2, \ldots, r_n)$. □

4.2 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 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. However, this problem can be solved by removing some of the suffixes of w_i's. Recall the procedure explained shortly in Section 2, for building an MSC based on a sequence of 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 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.

4.3 Forming the Final MSC Graph

Algorithm 2 given in Figure 4 is used to produce an MSC graph based on a given set of MSCs $\mathcal{M}$. It has two phases. The first phase considers every pair of MSCs M_1 and M_2 in $\mathcal{M}$ 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 3 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 3. We will call these subgraphs as paths below. These paths will be referenced using the labels of the edges. The loops in the labels of the paths will be represented by using $\star$. The second column of Figure 3 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

$$\begin{aligned} M_a &= M_1 M_2 M_2 M_3 M_4 M_5, \\ M_b &= M_1 M_2 M_2 M_3 M_4 M_4 M_5, \\ M_c &= M_1 M_3 M_4 M_5. \end{aligned}$$

M_a and M_b infer M_4 to be repeat–repetitive within the context $M_1 M_2^{(2)} M_3$–M_5. Hence

$$M_d = M_1 M_2^{(2)} M_3 M_4 M_4^{\star} M_5$$

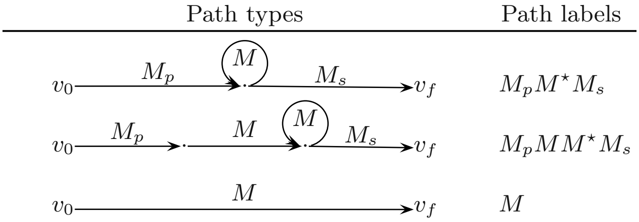


Fig. 3. Three different path types

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 loop left to be inferred. However, all possible relative positions of these loops 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 second phase of Algorithm 2 performs the merging of paths using Algorithm 3 given in Figure 5. 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 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.

```
 1: /* phase 1: infer loops and form $G$*/
 2: initially all the MSCs in $\mathcal{M}$ are unmarked
 3: generate the initial and the final nodes $v_0$ and $v_f$ in $G$
 4: for each pair $M_1, M_2 \in \mathcal{M}$ 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 $\Pi$ of the set of paths such that two paths $p, p'$ are in the same
    subset $P$ of $\Pi$ iff $\exists$ a sequence of paths $p_1, p_2, \ldots, p_k$ where $p_j \in P$, $1 \le j \le k$,
    $p = p_1$, $p' = p_k$, and for $1 \le i < k$, $src(p_i) \cap src(p_{i+1}) \ne \emptyset$
22: for all $P \in \Pi$ do
23:    insert $merge(P)$ into $G'$
24: end for
```

Fig. 4. Algorithm 2 – Building the MSC graph based on a set of MSCs $\mathcal{M}$

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 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_pM_s on the path 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 [8, 19]. For-

```
let $p$ be a path in $P$ whose prefix label is the shortest among other paths in $P$
let $p_m = p$
let $src(p_m) = src(p)$
let $P = P - \{p\}$
while $P$ is not empty do
    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 $P$.
    $src(p_m) = src(p_m) \cup src(p)$
    $P = P - \{p\}$
    trace the concatenation of the prefix label $M_p$ and the suffix label $M_s$ of $p$ in $p_m$
    (by skipping $\varepsilon$ edges)
    if during this trace $M_p$ ends in the middle of an edge in $p_m$ then
        insert a new node $v$ in the middle of that edge in $p_m$
        insert a new edge $(v, M, v)$ in $p_m$ where $M$ is the loop label of $p$
    else
        let $v$ be the node at which the trace of $M_p$ has ended
        if during the trace of $M_p$, $v$ is visited exactly once then
            insert a new edge $(v, M, v)$ in $p_m$ where $M$ is the loop label of $p$
        else
            let $(v, M', v')$ be the edge which is not used during the trace of $M_p$
            remove the edge $(v, M', v')$
            insert a new node $v''$ in $p_m$
            insert a new edge $(v'', M', v')$
            insert a new edge $(v, \varepsilon, v'')$
            insert a new edge $(v'', M, v'')$
        end if
    end if
end while
return $(p_m)$
```

Fig. 5. Algorithm 3 – Merging paths in a partition P

tunately, in principle, tracing of M_pM_s on p_m corresponds to the case given in Theorem 6 of [8] with a time complexity of $O(|p_m| \times |M_pM_s|)$.

5 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.

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 $M_a = M_1 M_2^{(2)} M_3 M_5$, $M_b = M_1 M_2^{(2)} M_3 M_4^{(2)} M_5$, and $M_c = M_1 M_3 M_4^{(3)} M_5$. M_a and M_b infers M_4 to be while–repetitive within the context $M_1 M_2^{(2)} M_3$–M_5. Hence $M_d = M_1 M_2^{(2)} M_3 M_4^{\star} M_5$ 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–$M_3 M_4^{(3)} M_5$ when it is considered together with $M_e = M_1 M_2^{(2)} M_3 M_4^{(3)} M_5$ 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.

Acknowledgement

The authors wish to acknowledge many useful discussions with Jessica Chen. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, under grant OGP 976.

References

[1] ITU-T Recommendation Z.120. Message Sequence Charts (MSC96) (1996) 133

[2] Rudolph, E., Graubmann, P., Gabowski, J.: Tutorial on message sequence charts. Computer Networks and ISDN Systems–SDL and MSC **28** (1996) 133

[3] Alur, R., Holzmann, G. J., Peled, D.: An analyzer for message sequence charts. Software Concepts and Tools **17** (1996) 70–77 133

[4] Ben-Abdallah, H., Leue, S.: Syntactic detection of progress divergence and non–local choice in message sequence charts. In: 2nd TACAS. (1997) 259–274 133

[5] Alur, R., Yannakakis, M.: Model checking of message sequence charts. In: 10th International Conference on Concurrency Theory, Springer Verlag (1999) 114–129 133

[6] Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: 22nd International Conference on Software Engineering. (2000) 304–313 133

[7] Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering **29** (2003) 623–633 133, 134, 135, 136

[8] Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. In: Automata, Languages and Programming, 28th International Colloquium, ICALP, LNCS 2076. (2001) 134, 146, 147

[9] Lohrey, M.: Safe realizability of high–level message sequence charts. In: 13th International Conference in Concurrency Theory, CONCUR 2002. (2002) 177–192 134

[10] Chikofsky, E., Cross, J.: Reverse engineering and design recovery. IEEE Software **7** (1990) 13–17 134

[11] Lee, D., Sabnani, K.: Reverse engineering of communication protocols. (In: IEEE ICNP'93) 208–216 134

[12] Koskimies, K., Makinen, E.: Automatic synthesis of state machines from trace diagrams. Software–Practice & Experience **24** (1994) 643–658 138

[13] Rajagopal, M., Miller, R. E.: Synthesizing a protocol converter from executable protocol traces. IEEE Transactions on Computers **40** (1991) 487–499 138

[14] Zafiropulo, P., West, C., Rudin, H., Cowan, D. D., Brand, D.: Towards analyzing and synthesizing protocols. IEEE Transactions on Communications **28** (1980) 651–660 138

[15] Saleh, K., Boujarwah, A.: Communications software reverse engineering: A semi–automatic approach. Information & Software Technology **38** (1996) 379–390 138

[16] Saleh, K., Probert, R. L., Manonmani, I.: Recovery of communication protocol design from protocol execution traces. (In: IEEE ICECCS'96) 265–272 138

[17] Chen, X. J., Ural, H.: Construction of deadlock–free designs of communication protocols from observations. The Computer Journal **45** (2002) 162–173 138

[18] Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press (1994) 143

[19] Muscholl, A., Peled, D., Su, Z.: Deciding properties of message sequence charts. In: Foundations of Software Science and Computation Structures. (1998) 146