Recovering Repetitive Sub-Functions From Observations

Guy-Vincent Jourdan[1], Hasan Ural[1], Shen Wang[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
{gvj,ural,swang010}@site.uottawa.ca
[2] Faculty of Engineering and Natural Sciences
Sabancı University
Tuzla, Istanbul, Turkey 34956
yenigun@sabanciuniv.edu

Abstract. This paper proposes an algorithm which, given a set of observations of an existing concurrent system that has repetitive sub-functions, constructs a Message Sequence Charts (MSC) graph where repetitive sub-functions of the concurrent system are identified. This algorithm makes fewer assumptions than previously published work, and thus requires fewer and easier to generate observations to construct the MSC-graph. The constructed MSC-graph may then be used as input to existing synthesis algorithms to recover the design of the existing concurrent system.

1 Introduction

A concurrent system is a system with two or more processes that are communicating among themselves using message exchanges. Message Sequence Charts (MSCs) [1, 2] provide a visual description of a series of message exchanges among communicating processes in a concurrent system. MSCs are often used by designers to depict individual intended behaviors of the concurrent system. However, a collection of such MSCs can only be viewed as providing information on a representative sample of the intended behavior rather than a design representation of the system giving a complete description of the functionalities to be provided [3]. A design representation is useful not only for implementing the system, but also for maintaining it, for example to detect and eliminate errors, to adapt it to a different environment, or simply to better understand the system. It also helps reusing parts of the system in new developments. Unfortunately, complete, up-to-date designs of evolving existing systems are seldom available.

Consequently, one of the aims of reverse engineering [4–6] is to recover the design of an existing concurrent system through an analysis of its runtime behavior. Such an analysis requires a finite set of *observations* of the running system. Each observation is a serialization of the events occurring possibly concurrently

during a system run. Due to the possible interleavings of these concurrent events, there are other serializations for the same run, all of which can be derived from the given serialization [4]. Each such observation can be seen as a word, which is made of the events being observed, belonging to the language of the system. From one word (observation), it is possible to derive other words corresponding to all remaining interleavings of the concurrent events in that word. If we are given a set of observations, we can thus infer a set of words as a union of the subsets of words where each subset corresponds to all possible interleavings of the events in each of these observations. However, this set is only a representative subset of the complete language of the system. Our aim is to derive, under some assumptions, an MSC-graph [7] that represents the complete language of the system from which a design of the system can be constructed using existing synthesis algorithms [4].

Since many concurrent systems have repetitive sub-functionality, some evidence for such sub-functions should at least be implicitly given in the set of observations. For a complete and accurate recovery of the design of a concurrent system, the given set of observations must provide evidence for each repetitive sub-function and must imply its relative position among other repetitive sub-functions of the system. This places some constraints on the nature of the observations which need to be taken into consideration when the set of observations are formed.

Existing methods to infer repetitive sub-functions require several restrictive assumptions on the set of observations. For example, the method presented in [8] requires (among others) the following assumptions:

i. Repetitive sub-functions must be iterated the same number of times in each observation,
ii. Repetitive sub-functions need to be introduced in a specific order,
iii. The ordering of the sub-functions must be totally unambiguous,
iv. Each sub-function must be "introduced" individually by an observation that contains only "known" sub-functions and this new sub-function.

In [9], the authors introduce a new concept, the *lattice of repetitive sub-functions*, a structure that provides all possible selections of n repetitive sub-functions. Using that lattice, they are able to infer the set of repetitive sub-functions of an application from a set of observations waiving several of the assumptions made in [8]. In particular, the first three assumptions listed above are waived. However, the fourth and the strongest assumption is still required by the approach taken in [9].

In this paper, we eliminate that assumption and provide an algorithm that is capable of recovering several repetitive sub-functions at once under a new assumption that repetitive sub-functions have a single initiator. We believe this to be a significant practical improvement over both previous methods [8, 9] since it relieves the user from the requirement of isolating each repetitive sub-function within its own observation, which could be fairly difficult in practice, and sometimes simply impossible if two or more repetitive sub-functions are tied together in the design of the system. The new assumption regarding the unique initiator

to repetitive sub-functions does not seem too constraining, since a repetitive sub-function is primarily a function and thus is usually initiated by a single process. In addition, the assumption is introduced for efficiency only and can be waived at the cost of increased complexity.

The paper is organized as follows: in Section 2, we introduce the concepts and definitions required. In Section 3, we review and discuss the assumptions that are made about the system and the observations. The proposed algorithms are described and analyzed in section 4, and in Section 5 we illustrate our approach on an example. We conclude in Section 6, where an implementation of the solution is also described.

2 Preliminaries

In this section, we introduce the concepts and notations required. We mostly reuse the notions and notations of [9], adapting them as needed.

Figure 1, left, shows an MSC of three processes exchanging a total of five messages. The message m_1 is sent by the process P_2 and received by the process P_3, which is represented by an arrow from P_2 to P_3 and labeled m_1. Each message exchange is represented by a pair of *send* and *receive* events. The local view of the message exchanges of a process (send and receive events of a process) is a total order, 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. Any of the linearizations of an MSC uniquely determines the MSC.

To describe a functionality that is composed of several sub-functionalities, an MSC-graph (a graph with a source and a sink node where edges are labeled by MSCs) can be used. An MSC corresponding to the concatenation of MSCs along a path from the source node to the sink node in an MSC-graph is said to be in the language of the MSC-graph. In the following, M^k means that M is repeated k times, and M^* means any number of repetitions of M. Figure 1, right, shows an MSC-graph where the MSC M_p is followed by an arbitrary number of iterations of the MSC M, followed by the MSC M_s, which defines the language $M_p.M^*.M_s$. In this paper we assume that an MSC in the language of an MSC-graph represents a system functionality from the initial state to the final state, without going through the initial state again during the execution.

Formal semantics associated with MSCs provides a basis for their analysis such as detecting timing conflicts and race conditions [10], non–local choices [11], model checking [12], and checking safe realizability [13, 4].

In this paper, we consider the reverse engineering of designs of existing concurrent systems from given sets of observations of their implementations. We assume that we are given a set Ω of observations, each observation $O \in \Omega$ being an arbitrary linearization of an MSC m from a set of MSCs that is not given.

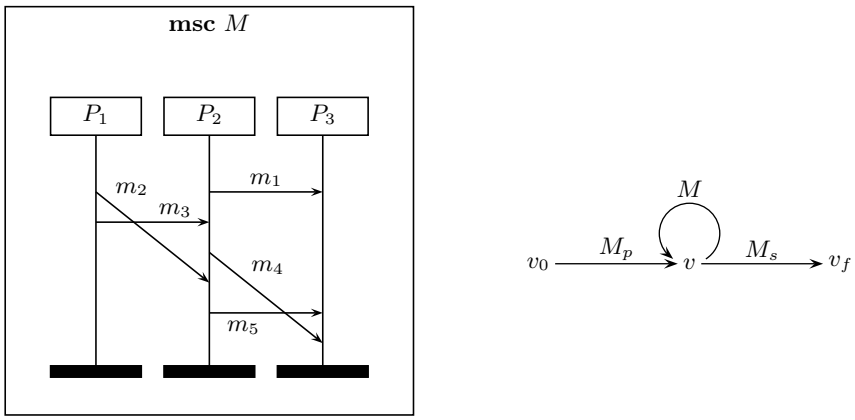

Fig. 1. An MSC of three processes (left) and an example MSC-graph (right)

We use $m(O)$ to denote the MSC m implied by an observation O. Some of the sub-functions of the system can be *repetitive*, in which case they can be called consecutively a different number of times in different runs of the system. As in [8, 9], we assume that a repetitive sub-function does not start (resp. end) at the initial (resp. final) state, and that every repetitive sub-function of the system (if any) is represented in the given set of observations at least twice: once with no occurrence, and once with two or more consecutive occurrences.

A sub-function that is repeated in an observation will create a repeated pattern in the MSC corresponding to that observation. However, a simple pattern repetition is not enough. In order to deduce the existence of a repetitive sub-function, we need to have an evidence such as different number of iterations of the pattern within the same context.

Definition 1. *[9] An MSC M is* 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.*

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. M_p and M_s are what we call a prefix of M and a suffix of M, respectively. If an MSC M' is the concatenation of three non empty MSCs M_p, M_m and M_s (i.e. $M' = M_p.M_m.M_s$), we say that M_m occurs *within the context* M_p–M_s, that is, M_m occurs after M_p and is followed by M_s.

Definition 2. *[9] 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:*

1. *M does not have a basic repetitive MSC,*

2. $M_1 = M_p.M^k.M_s$ *for some* $k \geq 2$*, with* M_p *and* M_s *non-empty and* $M_2 = M_p.M_s$*,*
3. M *is not a suffix of* M_p *and* M *is not a prefix of* M_s*.*

Definition 3. *[9] A* common prefix (resp. suffix) *of two MSCs* M_1 *and* M_2*, is an MSC* M*, such that* M *is a prefix (resp. suffix) of both* M_1 *and* M_2*. The* maximal common prefix (resp. suffix) *of* M_1 *and* M_2 *is a common prefix (resp. suffix)* M *of* M_1 *and* M_2 *with the largest number of events.*

The set of send and receive events in an MSC can be partially ordered according to *causality*. We define the causal relationship as follows: two events e_1 and e_2 of an MSC M are *causally related*, which we note $e_1 < e_2$ if and only if

1. e_1 is a *send* event and e_2 is the corresponding *receive* event, or
2. e_1 and e_2 are events of the same process and e_1 happens before e_2 on that process, or
3. there exists an event e_3 in M such that $e_1 < e_3 < e_2$.

For any *send* event e, we will define the set *Previous*(e) of elements, one per process, that do not happen *after* e and that are maximal on their process with that property. More formally:

Definition 4. *Let* M *be an MSC with* k *processes* $\{p_1, p_2, \ldots, p_k\}$*, and let* e *be a send event of* M*.* *Previous*(e) *is a set of up to* k *events such that* $\forall j \in \{1, \ldots, k\}$*, for all event* e' *of* $p_j, e' \in$ *Previous*(e) *if and only if* $e \not< e'$ *and for all events* $e'' \neq e'$ *of* $p_j, e \not< e'' \Rightarrow e'' < e'$*.*

A *linear extension* of the events of an MSC is a *total ordering* of the events that respects the (partial) causal ordering:

Definition 5. *Let* M *be an MSC with* n *events* $\{e_1, e_2, \ldots, e_n\}$*. A linear extension of the causal order* $<$ *of the events of* M *is a tota order* $<_L$ *on the events of* M *such that* $\forall i, j \leq n, e_i < e_j \Rightarrow e_i <_L e_j$*.*

3 Assumptions

As mentioned earlier, previous work [8, 9] have been published on the same problem, [9] making fewer assumptions than [8] about the system being reverse engineered. In this paper, we are waiving one of the strongest assumptions made in [9], namely that each repetitive sub-function is introduced by a particular observation. We in turn make a couple of less restrictive assumptions for efficiency reasons.

To recap, the most important assumptions made in [8] were the following:

1. There is one observation without any repetitive sub-functions. This observation is called the *initial* observation; it will be the shortest of all the provided observations and every other observation will be made of that initial observation plus a number of iterations of a number of repetitive sub-functions.

2. The initial observation, and each repetitive sub-function having nested repetitive sub-functions, have a non empty, repetitive sub-function free prefix and a non empty, repetitive sub-function free suffix.
3. Repetitive sub-functions have no common prefix with the part of the MSC that starts just after them and no common suffix with the part of the MSC that leads to them.
4. Repetitive sub-functions starting at the same point do not alternate.
5. Repetitive sub-functions must be iterated the same number of times in each observation,
6. Repetitive sub-functions need to be introduced in a specific order,
7. The ordering of the sub-functions must be totally unambiguous,
8. Each sub-function must be "introduced" individually with an observation that contains only "known" sub-functions and this new sub-function.

In [9], the assumptions 5, 6, and 7 are waived, but the strong assumption 8 is kept. In this paper, we waive assumption 8. However, we do introduce the following two new assumptions:

9. Sub-function have a single initiator. That is, there is always a unique *send* event at the source of a repetitive sub-function (and this send event is thus repeated at the beginning of each iteration of the sub-function).
10. Repetitive sub-functions repeat at least twice.

We will see that assumption 9 speeds up our algorithm. This assumption seems fairly reasonable, since functions have a single starting point.

Assumption 10 is there to avoid a particular case, where a set of repetitive sub functions "hide" each other, for example an initial observation $P.S$, and two other observation $P.A.B^{k_1}.S$ and $P.A^{k_2}.B.S$ for $k_1 > 1$ and $k_2 > 1$. The single occurrence of A in the second observation prevents B to be recognized as repetitive while the single occurrence of B in the third observation prevents A to be recognized as repetitive. Note that if a fourth observation allows A or B to be recognized then the problem disappears, so this assumption can be weakened to prevent only the problematic pattern. We have used a larger assumption for the sake of readability.

3.1 Main Algorithm

The main idea behind our algorithm is the following: at any given time, we have already built a particular "knowledge" of the system, the initial knowledge being the initial observation. We gradually enhance this knowledge by uncovering information about repetitive sub-functions. Given the current knowledge, say *current*, and an observation, say O, we attempt to "enhance" our knowledge by identifying in $m(O)$ portions that are coherent with *current* (that is, portions that are compliant with what *current* describes of the system), while the parts of $m(O)$ that do not match *current* are made exclusively of repetitive sub-functions.

We can sketch a first algorithm as follows: we first identify the longest common prefix of *current* and $m(O)$. After that common prefix, if O is not entirely

recognized yet then we must be looking at the beginning of a repetitive sub-function. That sub-function will iterate a certain number of times, after which $m(O)$ will either "reconnect" with *current* where it left off to go into the repetitive sub-function, or will enter into a second repetitive sub-function. In any case, it will eventually "reconnect" with *current*. The strategy is thus to first look for a possible "reconnection" point between $m(O)$ and *current*. When such a point is found, we check if the portion of $m(O)$ that has been skipped is made of one or more repetitive sub-functions. If that is not the case, we keep looking for another reconnection point further down in $m(O)$. If, on the other hand, what we have are repetitive sub-functions, then we have to see if we can complete the comparison starting from that reconnection point (and possibly find a number of additional repetitive sub-functions along the way). The simplest way to achieve this is to make a recursive call to the same algorithm, starting from that reconnection point. If the recursive call succeeds in finishing the comparison of *current* and $m(O)$, then we are done. If not, then we have to look for another reconnection point that would be further down in $m(O)$.

The above sketch achieves the expected result, but can be very inefficient when trying to find the next connection point. Indeed, after identifying the maximum common prefix of $m(O)$ and *current*, we know that the next connection point in $m(O)$ will have to match the next events on each process of *current*. If these events are not causally related (that is, these are independent events) then any combination of matching events on O can potentially be a connection point. If there are k processes involved and O has p matching events on each process, we will have to try up to p^k possible connections.

In order to avoid this combinatorial explosion, we can use Assumption 9 stating that repetitive sub-functions have a single initiator. The algorithm as described cannot benefit from such an assumption, since the connection point is searched at the end of the repetitive sub-function, on which no assumption is made. It is however possible to reverse the algorithm and go through *current* and O from the end to the beginning instead of from the beginning to the end. When going backward, the very same approach can be followed (find the longest *suffix*, then find the *previous* connection point, make sure that what was skipped on $m(O)$ is made of basic repetitive sub-functions and recursively call the same algorithm on the remaining part of *current* and O), except that with that strategy we know that the next connection point will be in $m(O)$ just *before* the beginning of a repetitive sub-function. Since each repetitive sub function has a single send event as initiator, it means that the only possible connection points correspond to the set $\mathit{Previous}(e)$ of a send event e. We thus simply have to try a number of candidates which are bounded by the number of send events in O.

Algorithm 1 performs the initialization and the loop that will "consume" the provided observations. The variable *current* holds the current knowledge of the system, initialized with the *initial* observation. The first loop is a phase of pre-computation on the set of observations: we calculate an ordering of the events which is compatible with the causal relation, and we pre-compute all possible $\mathit{Previous}(e)$. Both calculations will be used later in the main algorithm. Then,

the observations are compared with *current* one after the other, until they are all properly interpreted. It may be necessary to compare a given observation to *current* more than once, if the observation includes nested repetitive sub-functions, since *current* might not have inferred the sub-function containing the nested sub-function the first time around.

Algorithm 1 Initialization and Main Loop

```
 1: current = the MSC of the shortest observation
 2: Q = a queue of all other observations
 3: KeepGoing=true
    {Precomputation on the set of observations}
 4: for all observations O ∈ Q do
 5:    Compute linearExtension(m(O)), a linear extension of the events of the MSC
       m(O) induced by O
 6:    for all send event e in O do
 7:       Compute Previous(e)
 8:    end for
 9: end for
    {Main loop through the observations}
10: while Q ≠ ∅ AND KeepGoing==true do
11:    KeepGoing = false;
12:    for all observations O ∈ Q do
13:       if InferRepetitive(current, O) then
14:          remove O from Q
15:          KeepGoing=true
16:       end if
17:    end for
18: end while
    {If Q ≠ ∅, some observations were not handled}
19: if Q ≠ ∅ then
20:    ERROR: some observations were not processed
21: else
22:    SUCCESS: the system has been reversed engineered as current
23: end if
```

4 Repetitive Sub-Function Inference Algorithm

Algorithm 2 given below attempts to *trace O* in *current* and to infer new repetitive sub-functions. The call to *FindMaximumSuffix* traces the maximum possible suffix common to *current* and $m(O)$. The location (starting) of this suffix is returned in *cutCurrent* and *cutO*.

Algorithm 3 implements *FindNextConnectionPoint*. Due to the assumption of having a single initiator, we simply have to search backward on *linearExtension(m(O))* for a *send* event e so that *Previous*(e) matches *currentCut*.

Algorithm 2 BOOLEAN InferRepetitive(IN-OUT *current*, IN O)

```
 1: FindMaximumSuffix(current,O, cutCurrent, cutO)
 2: if both cutCurrent and cutO are at the beginning of their MSC then
 3:    return true
 4: else if one of cutCurrent or cutO is at the beginning of its MSC then
 5:    return false
 6: end if
    {A repetitive sub-function might end at cutO}
 7: startingCut = cutO
 8: while true do
 9:    FindNextConnectionPoint(cutCurrent, O, startingCut, connectionPoint)
10:    if connectionPoint== ∅ then
11:       return false
12:    end if
13:    if IsMadeOfBasicRepetitives (O, connectionPoint , cutO) then
14:       if InferRepetitive(current[CurrentCut], O[Previous(connectionPoint)]) then
15:          modify current to include the newly discovered repetitive sub-function(s)
16:          return true
17:       end if
18:    end if
       {What we have found wasn't good, either because it wasn't basic repetitive or
       because it did not allow us to finish trace O inside current. We keep looping.}
19:    startingCut = Previous(connectionPoint)
20: end while
```

4.1 Finding Basic Repetitive Sub-Functions

In Algorithm 2, we extract a segment S of O which is not present in *current*. We must now see if this segment is made of one or more repetitive sub-functions. In [8], *BasicRepetitiveMSC()*, a linear time algorithm is provided. This algorithm is used to decide whether or not a given MSC is the concatenation of two or more basic MSCs. This algorithm is based on the fact that if an MSC is basic repetitive, then the sequence of labels on each of its processes are also repetitive. Such a sequence of label forms a word w, and finding the shortest word w' such that $w = (w')^k$ for some $k > 0$ is a well studied problem for which we have linear time algorithms [14].

Under the present assumptions, the segment S of $m(O)$ can be the concatenation of more than one basic repetitive MSCs, that is, S could be of the form $M_1^{k_1} M_2^{k_2} \dots M_p^{k_p}$, for $p \geq 1$ and $k_1 \geq 2, k_2 \geq 2, \dots k_p \geq 2$. Therefore, the algorithm *BasicRepetitiveMSC()* must be adapted to the multiple basic repetitive case.

Our approach to address this problem is the following: starting from S, we try to find a *single* basic repetitive MSC on the longest possible prefix of this segment. If we do find such a basic repetitive MSC on a prefix P of S, we recursively call our algorithm on $S \setminus P$ to find more basic repetitive MSCs in S. Here again, we use Assumption 9 stating that repetitive sub-functions have a single initiator, which allows to speed up the search quite dramatically, since

Algorithm 3 FindNextConnectionPoint(IN *cutCurrent, O, startingCut*,OUT *connectionPoint*)

```
1: for all send event e ∈ O before startingCut, moving backward on linearExtension(m(O)) do
2:    if Previous(e) == currentCut then
3:       connectionPoint = e
4:       return
5:    end if
6: end for
   {Connection point not found}
7: connectionPoint== ∅
```

it allows to look only at the prefixes of S that end at *Previous(e)* for some *send* event e.

Algorithm 4 BOOLEAN IsMadeOfBasicRepetitives (IN *O, connectionPoint , cutO*)

```
1: if BasicRepetitiveMSC(O[connectionPoint, cutO]) then
2:    return true
3: end if
4: for all send event e ∈ O between connectionPoint and cutO, moving backward on linearExtension(m(O)) do
5:    if BasicRepetitiveMSC(O[connectionPoint, Previous(e)]) then
6:       if IsMadeOfBasicRepetitives(O, e, cutO) then
7:          return true
8:       end if
9:    end if
10: end for
11: return false
```

4.2 Complexity of the Solution

In this section, we evaluate the complexity of the proposed solution in the worst case. We must first evaluate Algorithm 4, *IsMadeOfBasicRepetitives*, which is called by Algorithm 2, *InferRepetitive*.

In the following, we assume that the system being reverse-engineered involves k independent processes, and that the observations that are provided contain up to n events. There are up to p observations, and the size of the reconstructed system is m events. Clearly, $m \in O(p.n)$.

Proposition 1. *Algorithm IsMadeOfBasicRepetitives can be implemented to run in* $O(n^3)$.

Proof. As pointed out in [8], Algorithm *BasicRepetitiveMSC* can be made to run in $O(n)$, and this algorithm is called up to n times in the *for* loop. In addition, one should note that it is not necessary to recursively call *IsMadeOfBasicRepetitives* with the same $e - cut$ argument twice, since it would always return the same result (and actually return false if it was about to be called a second time, since the algorithm terminates as soon as one such call returns true). It is thus possible to record the fact that a particular $e-cut$ was already used and avoid a recursive call when this is the case. This can be checked in $O(n)$ and will limit the number of recursive calls to a maximum of n.

We can now evaluate the complexity of Algorithm 2.

Proposition 2. *Algorithm InferRepetitive can be implemented to run in* $O(n^5.m^k + k.n^2.m^{k+1})$.

Proof. Clearly, the algorithm *FindNextConnectionPoint* can be implemented to run in $O(n)$. Moreover, the algorithm will exit from the *while true* loop after at most n iterations. Proposition 1 tells us that *IsMadeOfBasicRepetitives* runs in $O(n^3)$, so the only missing information is the number of recursive calls to *InferRepetitive*. To do so, one should notice that it is not necessary to call *InferRepetitive* twice with the same pair of parameters. The first parameter corresponds to *Previous(e)* for some *send* event e, so the number of possibilities in bounded by n. If we consider that any cut in *current* is a possibility, there are at most m^k choices for the second parameter, and thus there are $O(n.m^k)$ possible pairs of parameters. Finding out if a given pair has already been used can be done in $O(n + k.m)$, so each complete run of one call of *InferRepetitive* (excluding recursive calls) can be completed in $O(n^4 + k.n.m)$.

Theorem 1. *The method proposed in this paper can be made to run in* $O(p^2.n^5.m^k + p^2.k.n^2.m^{k+1})$.

Proof. Immediate from propositions 1 and 2.

5 An Example

Let us illustrate our solution on a simple example. Assume that we are observing a system with three process p_1, p_2 and p_3. We note $s.m_{x,i,j}$ the sending of message m_x by p_i to p_j and $r.m_{x,i,j}$ the reception of message m_x by p_j from p_i. We are provided with the following two observations (omitting on-process ordering information, which is assumed to be preserved in the provided lists, that is, events of the same process are listed in the order they occur on that process):

$O_1 = s.m_{a,2,1}, s.m_{b,2,3}, s.m_{d,2,1}, r.m_{a,2,1}, s.m_{c,1,3}, r.m_{d,2,1}, r.m_{b,2,3}, r.m_{c,1,3}$
and

$O_2 = s.m_{a,2,1}, s.m_{a,2,1}, s.m_{a,2,1}, r.m_{a,2,1}, r.m_{a,2,1}, s.m_{c,1,3}, r.m_{a,2,1}, r.m_{c,1,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{d,2,1}, s.m_{b,2,3}, s.m_{d,2,1}, s.m_{b,2,3}, s.m_{b,2,3}, r.m_{d,2,1}, r.m_{b,2,3},$

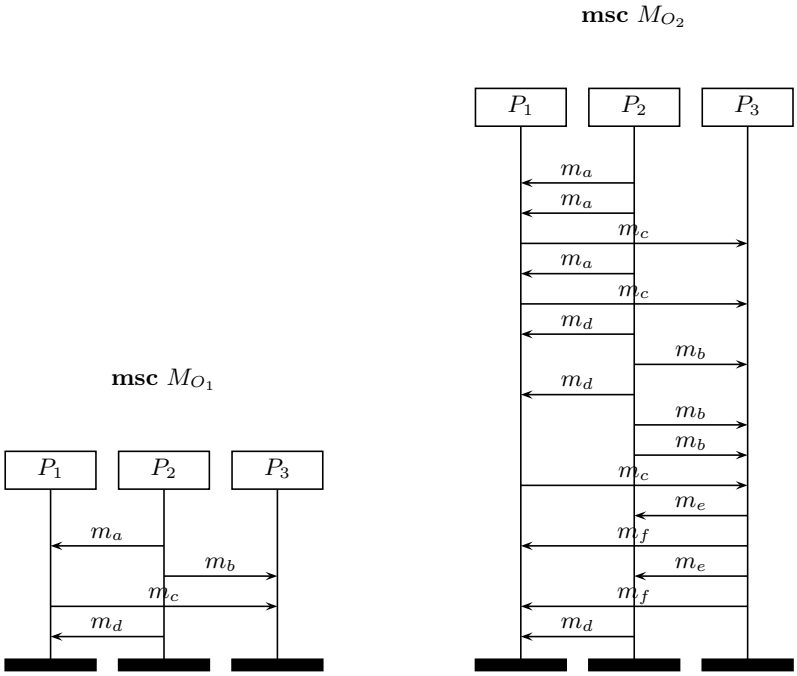

Fig. 2. MSCs infered by O_1 and O_2.

$r.m_{d,2,1}, r.m_{b,2,3}, r.m_{b,2,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{e,3,2}, s.m_{f,3,2}, s.m_{e,3,2}, s.m_{f,3,2}$, $s.m_{d,2,1}, r.m_{e,3,2}, r.m_{f,3,2}, r.m_{e,3,2}, r.m_{f,3,2}, r.m_{d,2,1}$.

These two observations induce the MSCs M_{O_1} and M_{O_2} respectively, as depicted in Figure 2. The shortest observation, and thus the initial one, is O_1. The algorithm *InferRepetitive*(O_1, O_2) is thus invoked.

The longest common suffix is the single-message MSC $M_1 = (m_{d,2,1})$. The reconnection point on O_1 is thus the reception of m_c on p_3, the sending of m_c on p_1 and the sending of m_b on p_2, which can be found in O_2 as *Previous*$(s.m_{e,3,2})$. The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{e,3,2}, m_{f,3,1}$, $m_{e,3,2}, m_{f,3,1}$, which infer the two-message MSC $M_2 = (m_{e,3,2}, m_{f,3,1})$ to be basic repetitive.

A recursive call to *InferRepetitive* is thus made on the MSCs leading up to the last occurrence of $m_{c,1,3}$ on both O_1 and O_2. This time, the maximum common suffix is the two-message MSC $M_3 = (m_{b,2,3}, m_{c,1,3})$, and the reconnection point is simply $m_{a,2,1}$. It is first found on O_2 as *Previous*$(s.m_{c,1,3})$, and *IsMadeOfBasicRepetitives* is then called on the segment $m_{c,1,3}, m_{a,2,1}, m_{c,1,3}, m_{d,2,1}, m_{b,2,3}$, $m_{d,2,1}, m_{b,2,3}$.

This call will fail identifying basic repetitive, and thus another connection point on O_2 will be searched for. It is found as *Previous*$(s.m_{a,2,1})$, and *IsMade-*

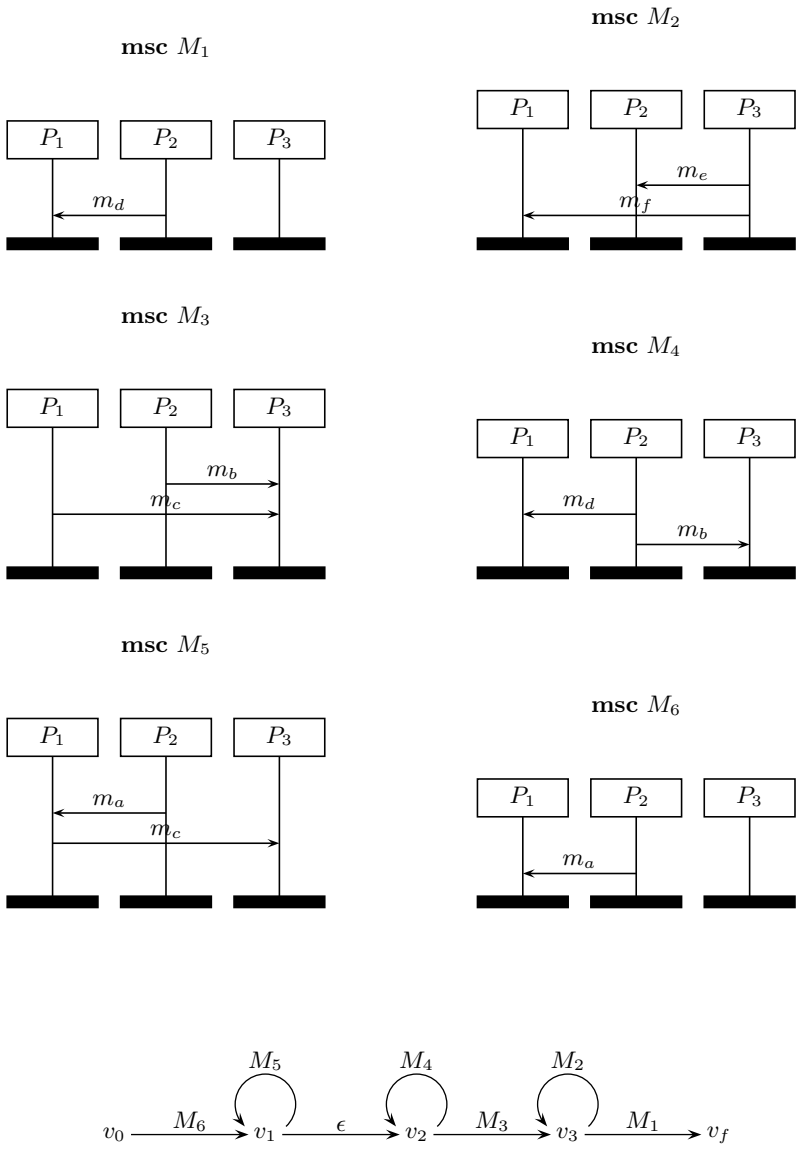

Fig. 3. Six MSCs obtained when processing M_{O_1} and M_{O_2}, and the final MSC-graph of the system as reverse engineered.

OfBasicRepetitives is called on the segment $m_{a,2,1}, m_{c,1,3}, m_{a,2,1}, m_{c,1,3}, m_{d,2,1}, m_{b,2,3}, m_{d,2,1}, m_{b,2,3}$, which this time is recognized as the concatenation of the basic repetitive MSC $M_5 = (m_{a,2,1}, m_{c,1,3})$ followed by the basic repetitive MSC $M_4 = (m_{d,2,1}, m_{b,2,3})$.

A recursive call to *InferRepetitive* is thus made on what is left of the traces, namely the first message $m_{a,2,1}$, which is immediately recognized as the single-message MSC $M_6 = \{m_{a,2,1}\}$ and the algorithm finishes on a success, with the system reverse engineered as $M_6.M_5^k.M_4^k.M_3.M_2^k.M_1$.

Figure 3 shows the six MSCs obtained as well as the final MSC-graph (the graph as an ϵ transition between v_1 and v_2, meaning that nothing happens when moving from v_1 to v_2). As expected, both O_1 and O_2 can be obtained from that graph: O_1 comes from $v_0.(M_6).v_1.(\epsilon).v_2.(M_3).v_3.(M_1).v_f$, and O_2 comes from $v_0.(M_6).v_1.(M_5).v_1.(M_5).v_1.(\epsilon).v_2.(M_4).v_2.(M_4).v_2.(M_3).v_3.(M_2).v_3.(M_2).v_3.(M_1).v_f$

6 Conclusion

We have introduced a reverse-engineering method to infer the presence of repetitive sub-functions in an application from which only a set of execution traces are provided. The method is much less restrictive than the previously published ones and is therefore much more practical. Our algorithm is capable of identifying repetitive patterns and repetitive sub-patterns (without limitations in the number of nested levels) that are appearing when comparing different executions of the same application being reverse-engineered, and build an MSC-graph from these patterns that "summarize" the knowledge of the design of the application.

The method described in this paper has been implemented in C++. The resulting tool is a 2000 lines program that takes an arbitrary number of execution traces and builds the corresponding MSCs and infer the MSC-graph in accordance to Algorithm 1. In our tests, the application was able to analyze 100 execution traces totaling over 40,000 message exchanges and infer the corresponding MSC-graph, uncovering 130 repetitive subfunctions in less than 10 seconds on a MS Windows© based computer with 1 Gigabyte of RAM and a 3.4 GigaHertz Intel pentium processor.

Details, documentation and source-code download are available on *http://www.site.uottawa.ca/FAST*.

References

1. ITU Telecommunication Standardization Sector: ITU-T Recommendation Z.120. Message Sequence Charts (MSC96). (1996)
2. Rudolph, E., Graubmann, P., Gabowski, J.: Tutorial on message sequence charts. Computer Networks and ISDN Systems–SDL and MSC **28** (1996)
3. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In: 9th European Software Engineering Conferece and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01). (2001)

4. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering **29** (2003) 623–633
5. Chikofsky, E., Cross, J.: Reverse engineering and design recovery. IEEE Software **7** (1990) 13–17
6. Lee, D., Sabnani, K.: Reverse engineering of communication protocols. In: IEEE ICNP'93. (1993) 208–216
7. Braberman, V., Oliveto, F., Blaunstein, S.: Scenario-based validation and verification for real-time software: On run conformance and coverage for msc-graphs. In: 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003. (2003)
8. Ural, H., Yenigun, H.: Towards design recovery from observations. In: FORTE 2004, LNCS 3235. (2004) 133–149
9. Jourdan, G.V., Ural, H., Yenigun, H.: Recovering the lattice of repetitive subfunctions. In: ISCIS 2005, LNCS 3733. (2005) 956–965
10. Alur, R., Holzmann, G.J., Peled, D.: An analyzer for message sequence charts. Software Concepts and Tools **17** (1996) 70–77
11. Ben-Abdallah, H., Leue, S.: Syntactic detection of progress divergence and non–local choice in message sequence charts. In: 2nd TACAS. (1997) 259–274
12. Alur, R., Yannakakis, M.: Model checking of message sequence charts. In: 10th International Conference on Concurrency Theory, Springer Verlag (1999) 114–129
13. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: 22nd International Conference on Software Engineering. (2000) 304–313
14. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press (1994)