NOTE TO USERS

uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Shen Wang

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Recovering Repetitive Sub-Functions from Observations in Distributed Systems

TITRE DE LA THÈSE / TITLE OF THESIS

Prof. Guy-Vincent Jourdan

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Prof. Samuel Ajila

Prof. Nejib Zaguia

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Recovering Repetitive Sub-Functions from Observations in Distributed Systems

Shen Wang

A Thesis

Submitted to the Faculty of Graduate and PostDoctoral Studies of the University of Ottawa in Partial Fulfillment of the Requirements for the Degree of Masters in Computer Science.*

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario

---

* The Masters program in Computer Science is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Your file Votre référence
ISBN: 978-0-494-49289-5
Our file Notre référence
ISBN: 978-0-494-49289-5

Abstract

Since many distributed systems are developed without complete or consistent design documents, some relevant designs need to be recovered from the executable software itself. One important objective of reverse engineering is to synthesize meaningful high-level design abstractions from the observations of a subject system. Algorithms, which, given a set of observations of an individual functionality of an existing distributed system, construct, under specific assumptions, a high-level design abstraction represented by an MSC-graph where repetitive sub-functions are identified, have already been proposed. In this thesis, we propose a new algorithm (based on the previous algorithms) that waives the strongest assumption made in the previously published work, and it is capable of recovering several repetitive sub-functions at once. Consequently, it is easier to generate observations for constructing a high-level design abstraction represented by an MSC-graph by using our proposed method. The resulted MSC-graph can then be used to facilitate the system maintenance or evolutionary development.

Acknowledgment

I would like to express my deepest gratitude to my supervisor, Dr. Guy-vincent Jourdan, who provided invaluable help throughout the preparation of this thesis. Without his insight suggestions, and extremely patient review, this thesis could never have been completed.

I would like to give my most sincere thanks to Professor Hasan Ural for his invaluable guidance and support throughout my study, especially during the preparation period of this thesis.

Finally, I would like to express my thanks to my husband, for his endless love and support.

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

Since many existing distributed systems are written without complete or consistent design documents, some relevant designs need to be recovered from the executable software itself for maintenance or evolutionary development purpose. One important objective of reverse engineering is to synthesize meaningful higher level design abstractions from the observations of a subject system [Chi90]. Given an observation obtained by the execution of an individual function of an existing distributed system with repetitive sub-functions, we can construct a Message Sequence Chart (MSC) representing the functionality executed. This thesis proposes an algorithm which, given a set of observations, constructs under specific assumptions an MSC-graph where repetitive sub-functions of the distributed system are identified. This algorithm makes fewer assumptions than previously published work, and thus requires fewer and easier to generate observations to construct the design model represented by an MSC-graph. The constructed MSC-graph can then be used to facilitate the system maintenance or evolutionary development. For example, the resulted MSC-graph may be used as input to existing synthesis algorithms that can refine and extend the system model.

1.1 Background

Software maintenance is a very important phase of software development cycle. However, the maintainers of most software systems were not their designers, so they need many resources to learn, understand and examine the system, but the existing design documents are usually not enough for that purpose. "Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction" [Chi90]. Consequently, reverse engineering of a software system can increase the overall comprehensibility of the system, and the resulted documents can be used to aid maintenance and support evolutionary development. By reducing the time required to gain a sufficient design-level understanding of software (including the time lost to misunderstanding), reverse engineering may greatly reduce the overall cost of software development [Chi90].

Reverse engineering generally involves building system representations in another form at a same abstraction level or at a higher abstraction level, and these representations are usually less implementation-independent [Chi90]. "Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself "[Chi90]. Design recovery requires methods and techniques for generating higher level

abstractions that can be utilized to support software maintenance and evolutionary development. System design abstractions help developers understand the system, make proper changes, such as optimizing deployed software, and remedying defects, as well as support new development. However, many existing software is written without using formal methods and their design documents are either incomplete or inconsistent with the existing software implementation. Consequently, some relevant designs need to be recovered from the executable software itself. In the context of design recovery, several attempts appeared in the literature to recover the design of an implementation from a given set of observations. Some works [Raj91] [Sal96] [Sal96b] [Sal99] [Che02] recover system design from observations in the context of communicating finite state machines, which is a traditional high-level design model. Some works [Ura04] [Jou05] consider recovering MSC-graphs from observations, which is the immediate basis of our research.

Message Sequence Charts (MSCs) are a commonly used visual descriptions of design requirements for distributed systems [ITU99] [Rud96], and have been incorporated into software design notations such as Unified Modeling Language (UML) [Rum99]. In a distributed system, two or more processes perform their functionalities by communicating among themselves via message exchanges. Depictions of individual intended behaviors of a distributed system can be described by MSCs [Rud96], and an individual MSC corresponds to a single (partial-order) execution of the system. MSCs can be used for requirement specification, validation, test case specification and documentation of distributed systems

[ITU99]. MSC is a formal language, and MSC supports structured design to specify multiple scenarios [ITU99]. MSC-graphs allow basic MSCs to be combined using operations such as concatenation, choice and repetition to form more complete specifications of a distributed system. Therefore, MSC-graphs can be viewed as a high-level model of the system, and it can be subjected to formal analysis such as detecting non-local choice [Ben97], pattern matching [Mus98], model checking [Alu99], and checking realizability [Alu00] (revised version appeared as [Alu03]).

1.2 Motivation and Objectives of the Thesis

In this thesis, we consider recovering high-level design abstraction of an individual functionality in a distributed system. The recovering process is based on a given set of observations that are obtained from actual execution traces of the functionality. Given such an observation, we can construct a Message Sequence Chart (MSC) representing the functionality executed. Each individual functionality of a distributed system can include a number of sub-functions. However, we do not attempt to identify all sub-functions one by one, and we only consider constructing an MSC-graph as a high-level design abstraction of an individual functionality, where repetitive sub-functions and their relative ordering are identified. The language of the MSC-graph derived consists of all the MSCs corresponding to the given observations and the inferred observations. Therefore, the constructed MSC-graph is considered as a high-level abstraction of the functionality of the system. Note that, some methods [Ura04] [Jou05] have already been developed to solve the problem. They are the

immediate basis of our research. We adopt some of the assumptions, but not all of the assumptions from [Ura04] [Jou05]. We assume that we are given a set of observations, each observation O being an arbitrary linearization of an MSC m from a set of MSCs that are not given. We assume that an observation w corresponds to a complete execution of a single function of an implementation of a distributed system, and the functions are assumed to start and end at the initial system state, without going through the initial system state. And among the observations, there is an initial observation without any repetitive sub-functions. The initial observation, and each repetitive sub-function having nested repetitive sub-functions, has a non-empty, repetitive sub-function free prefix and a non-empty, repetitive sub-function free suffix. Furthermore, we also assume the executions are monitored by someone who is familiar with the functionality of the system and can log the execution traces properly. Our additional assumptions will be introduced later. Table 4.1, listing the assumptions used in this thesis, is given in chapter 4.

We then propose a new algorithm to construct an MSC-graph where repetitive sub-functions and their relative ordering implied by the given set of observations are identified. The resulting MSC-graph can be used in the system maintenance or evolutionary development. For example, it can be checked for safe realizability using existing algorithm [Alu03], that is, if the behaviors in the resulting MSC-graph can be generated by some deadlock-free implementation or not. If the MSC-graph is not safely realizable, missing implied scenarios are given to help designers to refine and extend the specification [Alu03].

1.3 Contributions of the Thesis

In this thesis, the proposed method is developed based on some existing methods from [Ura04] and [Jou05]. In [Ura04], a method to recover repetitive sub-functions represented by an MSC-graph from observations is proposed, but the method requires several restrictive assumptions on the set of observations, including the following assumptions:

1. Repetitive sub-functions must be iterated the same number of times in each observation,

2. Repetitive sub-functions need to be introduced in a specific order,

3. The ordering of the sub-functions must be totally unambiguous,

4. Each sub-function must be "introduced" individually by an observation that contains only "known" sub-functions and this new sub-function.

In [Jou05], the authors introduce a new concept, the lattice of repetitive sub-functions, which is 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 from observations without several restrictive assumptions made in [Ura04]. In particular, the first three assumptions listed above are waived. However, the fourth assumption that is the strongest one is still required in the approach taken in [Jou05].

In this thesis, we eliminate that strongest assumption and provide an algorithm that is capable of recovering several repetitive sub-functions at once. For effective reason, a new assumption is introduced: repetitive sub-functions 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). Another assumption for the correctness of the method is explained in Chapter 4. We believe that our proposed method is a significant practical improvement over both previous methods [Ura04, Jou05] 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 for 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.

A paper associated with this thesis has been published at an international conference, Formal Techniques for Networked and Distributed Systems (FORTE) 2007.

1.4 Organization of the Thesis

Chapter 2 outlines the preliminaries needed to describe the proposed methods, including the notations and definitions of MSCs, MSC-graphs and observations, and a simple example to

explain the problem we are trying to solve in this thesis, as well as some more definitions used in our proposed method.

Chapter 3 reviews the existing methods in literature that have been proposed to recover the design of an implementation from a given set of observations, especially algorithms from [Ura04] and [Jou05], which are the basis of our research.

Chapter 4 describes our proposed method that waives an important assumption from previous method [Jou05]. In this chapter, we present the main idea of our proposed method, algorithms in pseudo code and corresponding explanation, the complexity of the algorithms and two illustrative examples, as well as the implementation details.

Chapter 5 presents the conclusions, with a summary of contributions and directions for future research.

Chapter Two

Preliminaries

In this thesis, we propose a new algorithm which, given a set of observations, constructs under specific assumptions an MSC-graph where repetitive sub-functions of the distributed system are identified. In this chapter, we give the preliminaries of our proposed method, and most of the notations and definitions that we will be using are directly adopted from [Alu03] [Ura04] [Jou05].

2.1 Message Sequence Charts

A Message Sequence Chart (MSC), describes the message flow among communicating processes in a system, and one MSC describes a partial behavior of the system [ITU99]. The transmission and consumption of messages are two asynchronous events, but a message must first be sent before it is consumed. For each communicating process covered by an MSC there is an instance axis. Along each instance axis the time runs from top to bottom, and a global clock is assumed for one MSC. If no coregion (the region on a instance axis in which the events of the processes are not ordered) or inline expression (composition of event structures could be defined inside of an MSC by means of inline operator expressions) is introduced, a total time ordering of events is assumed along each instance axis. Therefore, the local view of the message exchanges is a total order with respect to each instance, but the global view is a partial ordering on the set of events being contained. Partial order is a binary

relation, which is transitive, anti-symmetric and irreflexive. A partial order which is total (for all x and y on a set X of a binary relation R, it holds that xRy or yRx) is called a total order or a linear order.

A sample Message Sequence Chart is shown in Figure 2.1(a). Vertical lines in the chart correspond to asynchronous processes. A message exchanged between these processes is represented by an arrow, and the arrow can be drawn either horizontally or sloping downwards, but not upwards. The message can be split into two events. The tail of the arrow corresponds to the event of sending the message (labeled by $snd(m_i)$ where m_i is the message being sent) while the head corresponds to the event of receiving the message (labeled by $rcv(m_i)$ where m_i is the message being received). The set of send and receive events in an MSC can be partially ordered according to causality. We define the causal relationship as follows:

Definition 1. We say that two events e_1 and e_2 of an MSC M are causally related, which we denote $e_1 < e_2$ 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 $e_1 < e_3 < e_2$.

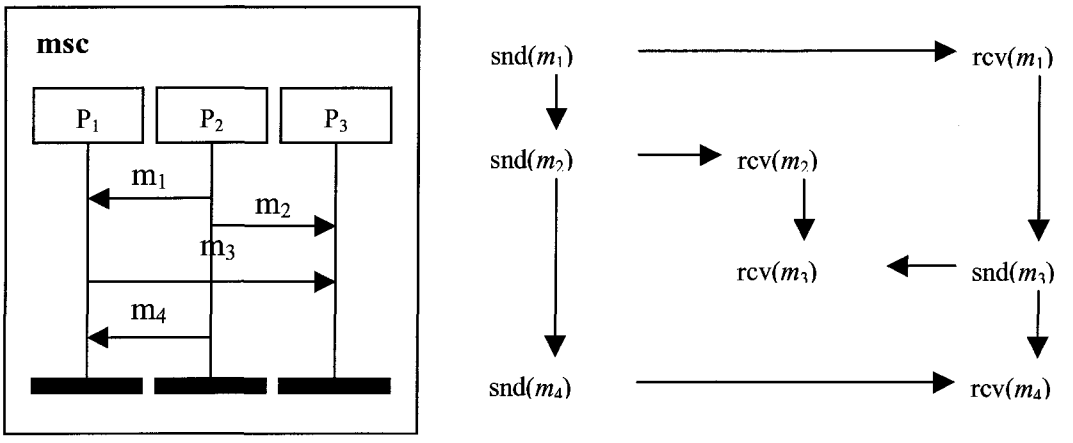

Figure 2.1 Message Sequence Chart and corresponding connectivity graph

For the Message Sequence Chart in Figure 2.1(a), on a set of send and receive events { $snd(m_1)$, $rcv(m_1)$, $snd(m_2)$, $rcv(m_2)$, $snd(m_3)$, $rcv(m_3)$, $snd(m_4)$, $rcv(m_4)$}, we can derive the following ordering relation $snd(m_1) < rcv(m_1)$, $snd(m_2) < rcv(m_2)$, $snd(m_3) < rcv(m_3)$, $snd(m_4) < rcv(m_4)$, $rcv(m_1) < snd(m_3) < rcv(m_4)$, $snd(m_1) < snd(m_2) < snd(m_4) < rcv(m_4)$, $rcv(m_2) < rcv(m_3)$ together with the transitive closure (the transitive closure of a binary relation R on a set X is the smallest transitive relation on X that contains R.). Here, the transitive closure of above ordering relation is the causality relation "<". Figure 2.1(b), a directed graph, describes this partial ordering, and the transitive closure is represented explicitly.

In the following four sub-sections, we give the formal notations and definitions of an MSC, and linerizations, projections, prefixes, and suffixes of an MSC, which are directly adopted from [Alu03] [Ura04] [Jou05].

2.1.1 The Notations and Definitions of an MSC

In this sub-section, we introduce the formal notations and definitions of an MSC.

A distributed system P is a set of processes $P = \{P_1, P_2, \ldots, P_n\}$, communicating with each other by exchanging messages from an alphabet Σ, over infinite slot buffers (not necessarily FIFO). All the MSCs mentioned will be Σ –labeled and defined on P. We adopt the same assumption from [Ura04] [Jou05] that Σ and P are assumed to be fixed. An event labelled as $s.m_{a,i,j}$ denotes the transmission of a message $a \in \Sigma$ by the process P_i to the process P_j. Similarly, an event labelled as $r.m_{a,i,j}$ 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 $\{1, 2, \ldots, n\}$. We define $\hat{\sum}^s = \{ s.m_{a,i,j} \mid i, j \in [n], a \in \Sigma \}$ as the set of send event labels , define $\hat{\sum}^r = \{ r.m_{a,i,j} \mid i, j \in [n], a \in \Sigma \}$ as the set of receive event labels, and define $\hat{\sum} = \hat{\sum}^s \cup \hat{\sum}^r$ as the set of event labels.

Definition 2 [Alu03] [Ura04]. A Σ-labeled MSC M for a concurrent system P is given by:

1. A finite set E which is divided into two sets: a finite set S of send events and a finite set R of receive events.

2. A mapping p: $E \rightarrow [n]$ that maps each event to a process on which it occurs. Let $E_i = \{e \in E \mid p(e) = i\}$ be set of events of P_i for $i \in [n]$.

3. A mapping l: $E \rightarrow \hat{\sum}$ that maps each event to a label such that $l(S) \subseteq \hat{\sum}^S$ and $l(R) \subseteq \hat{\sum}^R$. For consistency of labels, for all $s \in S$, if $l(s) = s.m_{a,i,j}$, then $p(s)=i$, and if

$l(r) = r.m_{a,i,j}$, then $p(r)=j$.

4. A bijection $f: S \rightarrow R$ that maps each send event e to its matching receive event such that if $l(e) = s.m_{a,i,j}$ then $l(f(e)) = r.m_{a,i,j}$.

5. For each $i \in [n]$, a total order $\leq_i$ on E_i, such that the transitive closure $\leq^*$ of the relation $\leq$ is a partial order on E. The relation $\leq$ is defined as

$$\bigcup_{i\in[n]} \leq_i \bigcup \{(s, f(s)) \mid s \in S\}.$$

Note that the total order $\leq_i$ on E_i gives a temporal order of execution of the events of P_i, and events on different processes are ordered via the relation $s.m_{a,i,j} \leq r.m_{a,i,j}$. An MSC can be viewed as a set E of $\hat{\sum}$ –labelled events partially ordered by $\leq^*$.

2.1.2 Linearizations and Projections of an MSC

In this section, we introduce the notations and definitions of linearization and projections of an MSC, which are directly adopted from [Alu03] [Ura04].

A word w over an alphabet $\hat{\sum}$ is made of a finite sequence of elements from the alphabet, and it actually is a finite sequence of event labels. For a word $w = w_1 w_2$ ($w_1 w_2$ denotes the concatenation of w_1 and w_2), the word w_1 is said to be a prefix of w, and the word w_2 is said to be a suffix of w.

Given an MSC M with a set of events $E = \{e_1, e_2, \ldots, e_n\}$, a linear extension of the partial order $\leq^*$ of the events of M is a total order $<_1$ on the events of M such that $\forall\, i, j \leq n$,

$e_i \overset{*}{\leq} e_j \Rightarrow e_i <_L e_j$. In other words, the total order $<_L$ induced by the given permutation on E is consistent with the partial order $\overset{*}{\leq}$. If we replace each event in a linear extension by its label, we got a string over $\hat{\sum}$, which is a linearization of M. Let $|E|$ denote the cardinality of the set E, the definition of a linearization of an MSC is given below:

Definition 3 [Alu03]. A word $w = w_1 w_2 \ldots w_{|E|}$ on $\hat{\sum}$ is a linearization of an MSC M iff there exists a total order $<_L = e_1 e_2 \ldots e_{|E|}$ of the events in E such that

1). for $i, j \in [|E|]$, whenever $e_i <_L e_j$, we have $i \leq j$ and

2). for $1 \leq i \leq |E|$, $w_i = l(e_i)$.

Not all sequences of send events and receive events are valid linearizations of MSCs. For example, in a valid linearization of an MSC, the send events must proceed the corresponding receive events because a message received must have been sent. What characterizes the words that can arise as linearizations of MSCs? First, we give the definition of well-formedness and completeness of a word. A word w over an alphabet $\hat{\sum}$ is well-formed if all receive events have matching sends [Alu03].

Definition 4 [Alu03] [Ura04]. Let α be an event label from $\hat{\sum}$, and $\#(w, \alpha)$ be the number of occurrences of α in w. An element $x \in \hat{\sum}$ is possible after a word v over $\hat{\sum}$ if 1) $x \in \hat{\sum}^S$ or, 2) $x = r.m_{a,i,j}$ with $\#(v, s.m_{a,i,j}) - \#(v, r.m_{a,i,j}) \geq 0$. If, for every prefix vx of a word w, x is possible after v, then w is said to be well-formed.

A word w over an alphabet $\hat{\sum}$ is complete if all send events have matching receive

events. That is to say, every message a sent by P_i to P_j must be received by P_j, within the word.

Definition 5 [Alu03] [Ura04]. A word w is said to be complete if $\forall i, j \in [n]$ and $\forall a \in \Sigma$, $\#(w, s.m_{a,i,j}) = \#(w, r.m_{a,i,j})$.

It is obvious that all linearizations of an MSC are well-formed and complete, and that if a word w over the alphabet $\hat{\sum}$ is well-formed and complete, then it is a linearization of an MSC. Therefore, we can say that, "a word w over the alphabet $\hat{\sum}$ is a linearization of an MSC iff it is well-formed and complete" [Alu03].

Given an MSC M, the projection of M on the ith process, denoted $M|_i$, is a total ordered sequence of labels of events occurring at process i in the MSC M. Similarly, given a word w, which is a linearization of M, we project w on the ith process, denoted $w|_i$. $w|_i$ is a sub-sequence of w, which involves the labels of events of process P_i in the MSC M.

In addition, we require our MSCs to satisfy a non-degeneracy condition, which is also assumed in [Alu03] [Ura04] [Jou05]. We will say an MSC is degenerate if two identical messages sent by some process P_i are received by another process P_j in a reversed order. In an non-degenerate MSC M, for any two send-events e_1 and e_2, if $l(e_1) = l(e_2)$ and $e_1 < e_2$, then $f(e_1) < f(e_2)$. Under this non-degenerate condition, an MSC M over $\{P_1, \ldots, P_n\}$ is uniquely determined by the sequences $M|_i$, $i \in [n]$. M does not depend on the actual interleaving of messages in a linearization w. Thus, we may equate $M \cong \langle\ M|_i \mid i \in [n]\ \rangle$. Because MSCs are fully characterized by their projections onto the event labels in the processes, An MSC

M is equal to an MSC M', if and only if for $i \in [n]$, $M|_i = M'|_i$. Likewise, a well-formed and complete word w over $\hat{\sum}$ uniquely characterizes an MSC M_w given by $\langle\ M|_i \mid i \in [n]\ \rangle$. Given a word w that is a linearization of an MSC, the actual MSC can be constructed easily by establishing the mapping between each send event and matching receive event. The procedure is first to project w on every process in order to construct $\langle\ M|_i\ i \in [n]\ \rangle$; and then, scan each $M|_i$ starting from the beginning, and match a send event with a label $s.m_{a,i,j}$ with the first not–yet–matched receive event with a label $r.m_{a,i,j}$ in $M|_j$. Therefore, given a well–formed and complete word w over $\hat{\sum}$, we can infer a unique MSC M, which we denote $msc(w)$. For an MSC M, define $L(M)$ to be the set of all linearizations of M. For a set $\mathcal{M}$ of MSCs, the language $L(\mathcal{M})$ is the union of languages of all MSCs in $\mathcal{M}$. Two MSCs M and M' are considered to be equal, if and only if $L(M) = L(M')$. Thus, an MSC can be viewed as a set of partially ordered labelled events on $\hat{\sum}$, and it stands for a set of linearizations (total order) determined by considering all possible inter-leavings of concurrent message exchanges implied by the partial order.

2.1.3 Prefixes and Suffixes of an MSC

In this sub-section, we introduce the definitions of prefix and suffix of an MSC.

A prefix or suffix of a string is a subset of the symbols in a string, where the order of the elements is preserved. It is much easier to understand the concept of prefix or suffix of a linear sequence than that of MSC. The prefix of an MSC is more complicated in structure, and it should be an MSC too. Let us begin from the visual representation of an MSC M as

shown in Figure 2.2. We draw a hierarchical line through M by crossing each process line exactly once, without crossing the message arrow between each process line (without separating matching send and receive events). This line cuts the MSC M into two parts. The part of MSC above the cutting line is what we call a prefix of MSC M, and the part below the cutting line is what we call the suffix of MSC M. The prefix and suffix of M can be shown to be MSCs again.

Definition 6 [Ura04]. Given an MSCs M with a set E of events, and an MSC M', with a set E' of events, M' is said to be a prefix (resp. suffix) of M, if and only if:

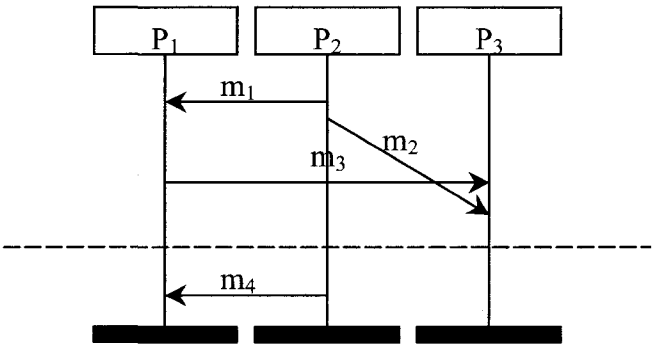

Figure 2.2 The prefix and suffix of an MSC M

(i) $E' \subseteq E$.

(ii) If $e \in E'$, $e' \in E$, and $e' \leq^* e$ then $e' \in E'$ (resp. *If* $e \in E'$, $e' \in E$, and $e \leq^* e'$ then $e' \in E'$)

(iii) If $s \in E' \cap S$ then $f(s) \in E' \cap R$ (resp. if $r \in E' \cap R$ then $f^{-1}(r) \in E' \cap S$, 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. If M_P is a prefix of M, M_S is a suffix of M, $E_P \cap E_S = \phi$; 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 $M_P M_S$.

Definition 7 [Jou05]. 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.

It is trivial to find the maximal common prefix of two sequences of labels by scanning and comparing the labels of two sequences starting from the beginning and stopping at the difference. 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. The maximal common prefix of an MSC can be constructed by the method given below, as explained in [Ura04].

Given an MSC M, let us consider $M_1|_i$ and $M_2|_i$ to represent ordered sequence of labels of events occurring at process i in the MSC M_1 and M_2, and w_i be the maximal common prefix of $M_1|_i$ and $M_2|_i$. However, $w = \langle\ w_i \mid i \in [n]\ \rangle$, which is a set of sequences of the event labels, may not characterize an MSC. The reason is that w may contain single send (resp. receive) event without matching receive (resp. send). Therefore, we can remove some of the suffixes of w_i's to get rid of each single send or receive event. However, the important thing is that

when we remove an event e from w_i, we must remove the events that are after e in w_i as well, and remove the corresponding receive (resp. send) event if it exists. The solution to construct maximal common prefix is that, first, calculate $w=\langle\ w_i \mid i\in[n]\ \rangle$, second, mark matching pairs of send and receive events, and at last, follow the MSC and remove all the event after the first unmarked event, and check that if removing marked event generate single send or receive events, then remove them. The algorithm of finding maximal common prefix of two MSCs is given in Appendix A.4, and finding maximal common suffix can be performed in a similar way.

2.2 Basic repetitive MSC

In this section, we give the definition of basic repetitive MSC.

Definition 8 [Ura04]. Given two MSCs M and M', M is said to be the basic repetitive MSC of M', if $M'=M^k$ for some $k \geq 2$ and there is no basic repetitive MSC of M.

The method to check the existence of a basic repetitive MSC M of a given MSC M' is also given in [Ura04] as well. It is a well-studied problem to check if a sequence of labels w' consists of repetitions of another sequence of labels w [Cro94]. If $w'=w^{(r)}$, then w is called a root of w' and r is called the power of w in w'. w is called as primitive if it cannot be written as a repetition of another word. Linear time algorithms exist to find the primitive root of a sequence of labels [Cro94]. 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 $r\geq 2$. Therefore, the solution of finding basic repetitive MSC M of a given MSC M' can start with

calculating w_i which is the primitive root of $M'|_i$, and r_i which is the power of w_i in $M'|_i$. Next, the greatest common divisor $r=gcd(\{r_i\})$ is calculated. If r is greater or equal than 2, then there exists a basic repetitive MSC M of M', and $M = \langle\ M_i = \text{first} \mid M'|_i \mid /r$ events of $M'|_i \mid i \in [n]\ \rangle$. The corresponding algorithm is given in Appendix A.5.

2.3 MSC-graphs

When the scenarios represented by a set of MSCs are big and complicated, it is better to use MSC-graphs to represent the scenarios in a more structured way. It allows MSCs to be combined using operations such as concatenation, choice and repetition [Alu99]. For the purpose of this thesis, we only consider recovering an individual functionality with repetitive sub-functions but without choice, and choice will not be interpreted in this definition of MSC-graph. In this section we introduce the formal notations and definitions of MSC-graphs, which are adopted from [Ura04] directly.

Definition 9 [Alu99] [Ura04]. An MSC-graph is a tuple $G = (V, v_0, v_f, T)$, where

--V is a finite set of nodes,

--$v_0 \in V$ is the entry node,

--$v_f \in V$ is the exit node,

--$T \subseteq V \times M \times V$ is a set of edges that connect nodes and are labeled from a set of MSCs M.

We use the notation $v \xrightarrow{M} v'$ to denote $(v, M, v') \in T$, and if $v=v'$, then we denote M as a basic repetitive MSC. In this thesis, we only focus on recovering basic repetitive MSCs, so that the MSC-graph we will use only contains two kinds of edges. One is "transition" edge labeled with "transition" MSC, and $v \neq v'$. And ε edge is a special kind of "transition" edge whose "transition" MSC is empty. Another one is the loop edge labeled with basic repetitive MSC, and $v=v'$. For a single loop of a basic repetitive MSC, it could be separated with other loops by none null "transition" MSC, and we call it single loop. For several loops concatenated together, we call them concatenated loops. In order to establish the relative ordering of two or more concatenated loops in an MSC-graph, we connected these loops with ε edge.

A path in G is a sequence of edges $(v_0, M_1, v_1)(v_1, M_2, v_2)(v_2, M_3, v_3) \cdots (v_m, M_m, v_f)$. The language of a path is given by the concatenation of the language of the MSCs that appear on the edges, $L(M_1)L(M_2) \cdots L(M_m)$. The paths starting from entry node v_0 and ending at exit node v_f represents the finite executions of the system modeled by the MSC-graph. A sample MSC-graph is depicted in Figure 2.3. It shows an MSC-graph that defines language

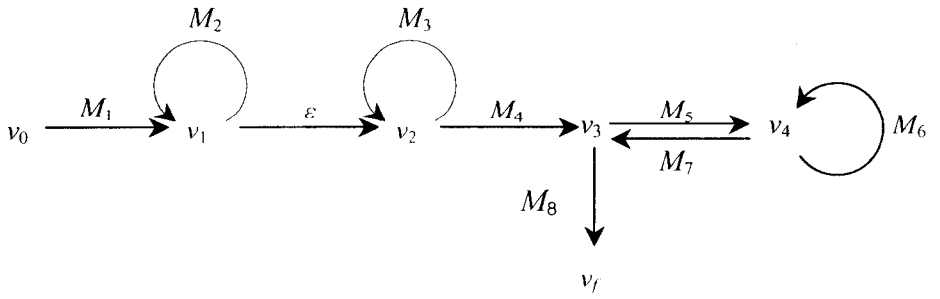

Figure 2.3. An example MSC-graph G_0

$M_1.M_2^*.M_3^*.M_4.(M_5.M_6^*.M_7)^*.M_8$. Here, M^* denotes sequential composition of 0 or more copies, or we can say that M^* *is* any number of repetitions of M. For an integer $k \geq 0$, M^k denotes the sequential composition of k copies of M, or we can say that M is repeated k times. In this MSC-graph, two loops labeled by the basic repetitive MSC M_1 and M_2, are concatenated together, and connected by an ε edge. After the transition MSC M_4, the third loop $((M_5.M_6^*.M_7)^*)$ begins and ends at node v_3, which include a nested loop labeled with the basic repetitive MSC M_6.

2.4 Observations

An observation is a well–formed and complete word over a set of event labels $\hat{\sum}$, or we can say an observation is an arbitrary linearization of an MSC that is not given. However, the un-given MSC can be obtained from this arbitrary linearization (the observation) as we explained in section 2.1. An observation reflects an individual functionality of an implementation of a distributed system in terms of sequences of events that occur during the execution [Che02]. We assume that the individual functionality is periodic, and an observation $o \in \hat{\sum}^*$ corresponds to a complete execution of a single function of the system. The observations are assumed to begin from the initial system state, and end at the initial state, without going through the initial state. And we also assume that someone who is familiar with the system behavior monitors observation collection, thus the observations can be properly collected to reflect the individual functionality.

An observation is a sequence of events of message transmissions and receptions of processes ordered according to their time of occurrences, which can be observed (or deduced) by logging the behavior of the implementation of a distributed system during its execution. We can use a communication system as an example, which provides a specified service to a number of service users who access the system through many distributed upper service access points [sal96b]. Therefore, the execution traces can be collected locally on these upper service access points, and then the collected traces can be serialized to derive the global temporal ordering of the recorded events according to their time of occurrences. Such a set of globally ordered events composes a global observation. [sal96b] shows an architecture for collecting traces from executions of a communication service, which can be adapted to collect traces for our purpose. However, in this thesis, we don't focus on the collection architecture, and we assume we already have such global observations.

2.5 A Simple Example

In this thesis, we consider recovering design abstractions (with repetitive sub-functions) represented by an MSC-graph from a finite set of observations obtained by the execution of an individual functionality of an existing distributed system. In order to illustrate the problem, in this section, we give a simple example on data transfer service.

Connection-oriented data transfer services are usually grouped into three phases or three sub-functions: connection establishment, data transfer, and connection release. Suppose that

there is a data transfer service in which two processes communicate with each other by passing messages from $\Sigma = \{CR, CC, DT, DR, DC\}$, where *CR* stands for connection request, *CC* for connection confirmation, *DT* for data transfer, *DR* for connection disconnect request, and *DC* for connection disconnect confirmation. Suppose the following is an observation we obtained from the execution of such a simple connection-oriented data transfer service.

$O = s.m_{CR,1,2},\ r.m_{CR,1,2},\ s.m_{CC,2,1},\ r.m_{CC,2,1},\ s.m_{DT,1,2},\ r.m_{DT,1,2},\ s.m_{DT,1,2},\ r.m_{DT,1,2},\ s.m_{DT,1,2},$
$r.m_{DT,1,2},\ s.m_{DR,1,2},\ r.m_{DR,1,2},\ s.m_{DC,2,1},\ r.m_{DC,2,1}$

Given the observation *O*, we can obtain $O|_i$ by projection of *O* on process *i* to form $\langle\ O|_i\ i=1,2\ \rangle$ that uniquely determines an MSC *M*. And then by matching corresponding send and receive events in $\langle\ O|_i\ \ i=1,2\ \rangle$, we can derive the MSC *M* as below:

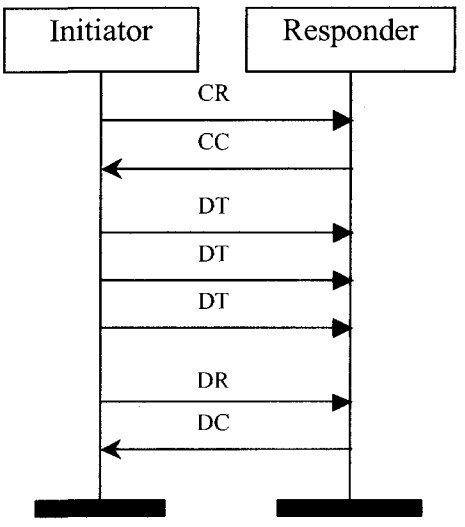

Figure 2.4 An Example MSC *M*

Given such MSC *M*, we can see in this functionality that connection is established first, and then data is transferred repeatedly from initiator to responder several times. From this run time behavior, there is a possibility that, in the abstract level, the MSC-graph in the Figure 2.5 was used at an early design time. However, we cannot guarantee that the repetitive pattern showed in *M* must be generated by iterations of a loop. It could possibly be the sequential appearance of *DT* in the design. Therefore, in order to infer repetitive sub-functions from run time behavior, we need more evidence. Next section, we give more definitions which will be used to recover designs with repetitive sub-functions. Then, in Chapter 4, we will propose a method to solve the problem based on the methods stated in [Ura04] and [Jou05].

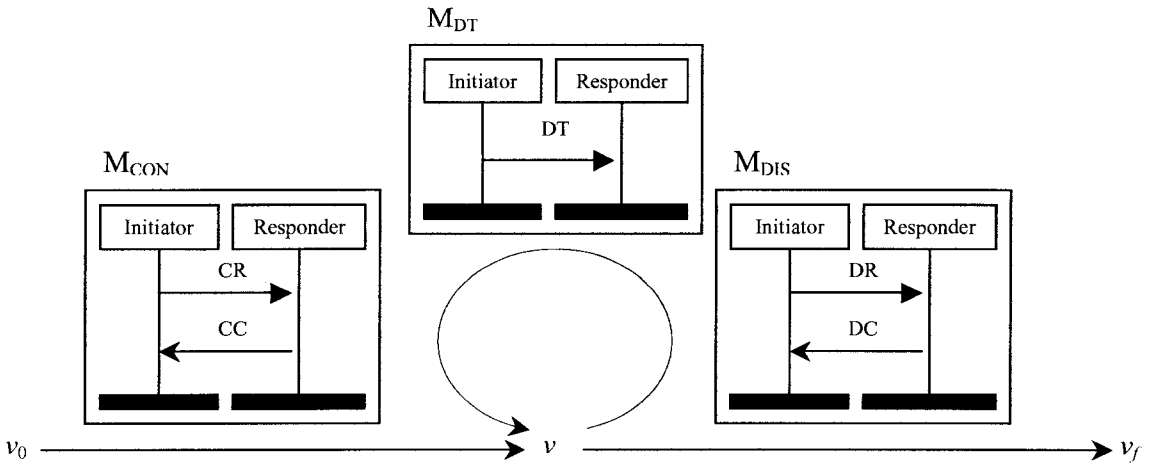

Figure 2.5 An Example MSC-graph G_1

2.6 More Definitions

We have given definitions of MSC (section 2.1), basic repetitive MSC (section 2.2), MSC-graph (section 2.3), as well as observation (section 2.4), and we continue to introduce more definitions that will be used in our proposed method.

We have mentioned that each individual function in a distributed system can be viewed as a combination of some sub-functions. Furthermore, we also consider that the behavior of a sub-function can be specified by an MSC [Ura04]. However, in general, it is not possible to identify all sub-functions represented by MSCs. In this thesis, we only focus on inferring those MSCs that represent repetitive sub-functions.

We also mentioned that, in an observation, a repeated pattern could be possibly generated by the iterations of the loops in the design, and it also could be the sequential appearance of some patterns. Therefore, only the repeated pattern itself is not enough to infer a loop, thus we need more evidences.

Definition 10 [Ura04] [Jou05]. Two MSCs M and M' are said to infer M_r to be repetitive within the context $M_p - M_s$ if all the following are satisfied:

1. $M = M_p . M_r^k . M_s$ for some $k \geq 2$, with M_p and M_s non-empty and $M' = M_p . M_s$,

2. M_r is not a suffix of M_p and M_r is not a prefix of M_s,

3. M_r does not have a basic repetitive MSC.

In this definition, we need at least two observations to infer a basic repetitive MSC (the body of a loop). One observation has two or more consecutive occurrences of M_r, and another one has no occurrences of M_r. Further, these two observations must have exactly the same prefix before the iterations of M_r, and exactly the same suffix after the iterations of M_r. That is they must appear within the same context. Under such evidences, the iterations of M_r are assumed to be generated by a loop in the design. Therefore, the system state right before the iterations of M_r is the same with the state right after the iterations of M_r, or in another word, the execution of M_r begin and end at the same system state. Further, any MSC in the form of M_p. M_r^*. M_s are realizable by the implementation of the system. Furthermore, since we assume that the observations begins and ends at the initial system state, but do not go through the initial system state, prefix M_p and suffix M_s must not be empty. For a nested repetitive basic MSC, it must have a non-empty, repetitive MSC free prefix and a non-empty, repetitive MSC free suffix.

Using the definition 10, we can only infer one basic repetitive MSC within the context $M_p - M_s$ at a time. If there are concatenation of more than one basic repetitive MSCs within the context $M_p - M_s$, for example $M = M_p M_1^{k_1} M_2^{k_2} \ldots M_n^{k_n} M_s$ for $n \geq 1$ and $k_1 \geq 2$, $k_2 \geq 2$, ... $k_n \geq 2$, then we can not handle it. Therefore, we need to adapt definition 1, so that we can infer several concatenated basic repetitive MSCs within the context $M_p - M_s$ at once.

Definition 11. Two MSCs M and M' are said to infer a set of concatenated *MSCs* to be basic repetitive within the context $M_p - M_s$ if all the following are satisfied:

1. $M = M_p\, M_1^{k_1} M_2^{k_2} \ldots M_n^{k_n} M_s$, for $n \geq 1$ and $k_1 \geq 2$, $k_2 \geq 2$, $\ldots$ $k_n \geq 2$ with M_p and M_s non-empty and $M' = M_p.\, M_s$,

2. M_1 is not a suffix of M_p, M_n is not a prefix of M_s, and M_{i+1} is not a suffix of M_i, and M_i is not a prefix of M_{i+1} for $1 \leq i < n$,

3. none of $M_1, M_2, \ldots M_n$ has a basic repetitive MSC.

Considering the general form representing the selection of all n basic repetitive MSCs $M_1, M_2, \ldots M_n$, "$M_p.M_1^*.T_1.M_2^*.T_2.M_3^*\ldots M_{n-1}^*.T_{n-1}.\, M_n^*.M_s$, where $\forall\, i \leq n$, M_i is a non-empty MSC representing a repetitive sub-function, T_i is a possibly empty 'transition' MSC, M_p is the non-empty prefix and M_s is the non-empty suffix" [Jou05]. The definition of inferring basic repetitive MSCs are given in general form as below:

Definition 12. An MSC with no basic repetitive MSCs, $M = M_p.T_1.T_2\ldots T_{n-1}.M_s$, and an MSC $M' = M_p.\, M_1^{k_1}.T_1.\, M_2^{k_2}.T_2\ldots T_{n-1}.\, M_n^{k_n} M_s$ are said to infer a set of *MSCs* to be basic repetitive within the context $M_p - M_s$ if all the following are satisfied:

1 $M' = M_s.\, M_1^{k_1}.T_1.\, M_2^{k_2}.T_2,\ldots T_{n-1}.\, M_n^{k_n} M_s$, for $n \geq 1$ and $k_1 \geq 2$, $k_2 \geq 2$, $\ldots$ $k_n \geq 2$ with M_p and M_s non-empty, and M_1 is not a suffix of M_p, M_n is not a prefix of M_s,

2. for $2 \leq j \leq n$, $1 \leq i < j$, and the possibly concatenated basic repetitive MSCs M_i, $M_{i+1},\ldots M_j$: M_{r+1} is not a suffix of M_r, and M_r is not a prefix of M_{r+1}, for $i \leq r < j\text{-}1$,

3. none of $M_1, M_2, \ldots M_n$ has a basic repetitive MSC.

Definition 13. Let M be an MSC with k processes $\{P_1, P_2, \ldots, P_k\}$, and $E = \{ e_1, \ldots, e_n \}$ $(n>k)$, is a set of event of M. A cut c is a subset of E, represented by a tuple $\{e_1, \ldots, e_k\}$ that contains a set of up to k events, one per process.

For any send event e, we will define the set *Previous*(e) of elements, one per process, that happen before e or are not causally related with e, and that are maximal on their process with that property. Figure 2.6 illustrate the set *Previous*($s.m_{e,2,1}$)= $\{s.m_{d,1,3}, r.m_{c,3,2}, r.m_{d,1,3}\}$ in MSC M, where the events $s.m_{d,1,3}$ and $r.m_{d,1,3}$ are not causally related with the event $s.m_{e,2,1}$, and the event $r.m_{c,3,2}$ is happen before the event $s.m_{e,2,1}$. More formally:

Definition 14. 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, $e \not< e'' \Rightarrow e'' < e'$.

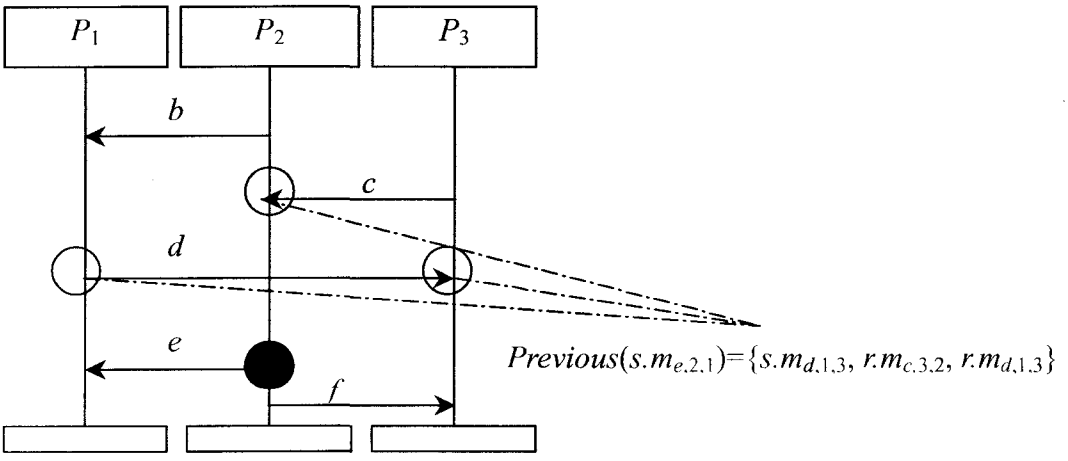

Figure 2.6 *Previous*($s.m_{e,2,1}$) in MSC M

Chapter Three

Previous Works

In the context of reverse engineering, the literature proposes various methods for the design recovery of distributed applications systems [sal96] [Sal96b] [Sal99] [Che02] [Ura04] [Jou05]. These methods typically infer a high-level abstraction of the distributed software from the code [sal96] or the software behavior based on selected observations [Sal96b] [Sal99] [Che02] [Ura04] [Jou05]. In [Sal96], authors propose a semi-automatic approach to recover an Estelle [ISO IS 9074] (extended state transition language) based protocol design from an existing *C* code of protocol implementation. Because recovery of design from code requires a complete and thorough understanding of the code, the full automation of the method is not possible in general [Sal96]. However, sometimes there is even no source code available for the implementation of the system. In this thesis, our concerns are the design recovery methods that utilize dynamic analysis of actual observations of the applications systems. In the dynamic approach, the advantage is that the information used to recover the system design is collected during the execution time, thus no false information is utilized in the recovery process. However, for lacking of source code, it is very difficult to recover the whole system design. Therefore, any efforts towards recovering one possible high-level abstraction of the system are regarded as reverse engineering. In the following sections, we will review several methods that recover systems design from observations based on communicating finite state machines (CFSM) or

MSC-graphs.

3.1 Recovering System Design based on CFSM

A communicating finite state machine (CFSM) can be formally defined as a quadruple $C=\{S, s_0, E, TF\}$ where S is a finite set of states, s_0 is the initial state, E is a finite set of events, and TF is the transition function $S\times E\rightarrow S$ [Sal96b]. CFSM, which can intuitively model the interactions between event-driven processes in a distributed system, is a very common high-level abstraction for the distributed system. Several methods [Sal96b] [Sal99] [Che02] have been proposed to construct a CFSM that represents the design of distributed software from the analysis of execution protocol traces.

In [Sal96b], Saleh gives an algorithm for obtaining CFSM-based protocol and service design from execution traces of cyclic (the initial and final states of the FSM are the same) and non-concurrent protocols. Then in [sal99], Saleh gives an algorithm for the design recovery of acyclic protocols taking into consideration the realistic protocol aspects of concurrency of events and their timing. These two methods assume that the presumed design does not include any unspecified receptions (a message at the front of some queue cannot be received due to a missing receive transition) and deadlocks (all channels are empty and none of the processes is ready to send a message). The methods start with execution traces collected from individual components of the existing system, serialize traces to derive the global temporal ordering of the recorded events, and then construct

protocol and service CFSM. Because there is no guarantee that the selected execution trace elements cover all possible transitions or behaviors of the system, the constructed protocol and service CFSM design is partial. Below, we give a more detailed review of the method to construct the partial protocol CFSM for each protocol entity in [Sal96b]. In the proposed method, an execution trace consists of a sequence of trace elements or records of events. Each trace element contains some information related to the corresponding event, including its name, its type (transmission or reception) and the location of its occurrence and an updated vector clock (the value of the vector clock of a trace element at the site of process P_i is an n-tuple for a system of n communicating processes and represents P_i's local view of the distributed system). The events are observed and recorded at one or more observation points of a communication system. These events could be either protocol messages, which are normally encapsulated within service primitives at lower service access points (a protocol entity interacts with medium through lower service access points) or service primitives at upper service access points (a protocol entity interacts with service user through upper service access points). Trace collection starts with the occurrence of an initial event, and stops with another occurrence of an initial event. Next, the traces collected at the different observation points of each protocol entity are serialized to derive the global temporal ordering of the recorded events according to the vector clock of each trace element. Furthermore, an algorithm is given to construct the partial protocol CFSM for each protocol entity. The algorithm starts from the initial state, and the next state reached by the CFSM is determined by executing an input event from the execution trace on the current state. If the

transition generated from last step is not available in the currently constructed CFSM, then a new state and a new transition function are created in the constructed CFSM. Then, the similar operation to determine next state and next transition function loops several times until the next input event in the execution trace is an initial event, which means that a cyclic trace sequence is over. At last, the final CFSM is obtained after equivalence reduction, which removes equivalent states to reduce the number of states of the constructed CFSM.

In [Chen02], the authors discuss the problem of the automated construction of designs from observations in a formal way. They assume the functionality of the implementation is periodic, and present a method to construct deadlock-free design from a given non-empty set of global observations that begin and terminate in initial global state without crossing the initial global state. The method starts with projecting the observations on each process, and then generating each initial constructed CFSM from the projection of each process. However, the initial constructed design may contain errors such as unspecified receptions and deadlocks even if the presumed design is error-free. In [Chen02], authors point out that non-determinism of constructed design and incompleteness of a given set of observations are causes of such errors in the constructed design. Chen proposes an algorithm to make the initial constructed tree-like CFSM deterministic, and this determinization rule removes the errors caused by non-determinism and introduces no new errors. Authors also point out that the incompleteness of observations can cause errors in the constructed design when there are two processes sending messages concurrently in the presumed design. Collecting more

observations is one of the solutions to eliminate such errors. Otherwise, the authors suggest a negation rule, which uses additional projections that are derived from the set of given projections on each process to modify the constructed design in order to eliminate all deadlocks in it. However, the constructed design modified by the negation rule could generate some new occurrences that cannot be generated by presumed design.

In this section, we have reviewed design recovery from observations in the context of CFSM that is a traditional high-level model for concurrent systems. We can think CFSM as a parallel composition of sequential machines, and then MSC-graph should be looked as a sequential composition of concurrent executions [Alu99]. Therefore, MSC-graph also plays an important role in design of concurrent systems. In the next section, we will review design recovery from observations in the context of MSC-graph, which is an immediately basis of our research.

3.2 Recovering System Design based on MSC-graph

MSC-graphs are used to structure multiple scenarios, which can be used to represent higher-level abstraction of a distributed system. In [Ura04], the authors propose an algorithm for the construction of an MSC-graph from a given set of observations of an existing concurrent system that has repetitive sub-functions. This algorithm is the basis of our research, which will be partly used in our new algorithm. Then in [Jou05], the authors introduce a concept of the lattice of repetitive sub-functions to waive several important assumptions made in [Ura04].

3.2.1 Towards Design Recovery from Observations

The method proposed in [Ura04], starts with a set of observations of an existing concurrent system, each observation O being an arbitrary linearization of an MSC m from a set of MSCs that are not given. For an observation O, there exists a unique MSC $msc(O)$ which can be constructed from O easily as explained in chapter 2.

3.2.1.1 Building the Initial MSC-graph

The solution proposed in [Ura04] mainly consists in two phases. The first phase builds an initial MSC-graph G consisting separate paths inferred by the given set of MSCs, and the second phase constructs the final MSC-graph G' by merging the separate paths in a same partition. The algorithms used in both phases are listed in appendix A and simple explanations are given below.

In order to construct initial MSC-graph, first, all the basic repetitive MSCs need to be recovered from the given set of MSCs. In the recovering procedure, each pair of MSCs M_1 and M_2 in a given set of MSCs is processed to see if the pair infers a repetitive MSC according to definition 10 given in the chapter 2 (the original algorithm is given in Appendix A.1). The algorithm starts with the calculation of the maximal common prefix M_p and maximal common suffix M_s of M_1 and M_2 (suppose M_1 is the shorter MSC), next removes the maximal prefix and suffix from M_1 and M_2. Let M_1" be the remaining part of M_1, and M_2" be the remaining part of M_2. If maximal common prefix or maximal common suffix is empty, or M_1" is not empty, then M_1 and M_2 do not infer a repetitive MSC. If not, then M_2" is

checked for the existence of a basic repetitive MSC M. If M is a basic repetitive MSC of $M_2^{"}$, then M_1 and M_2 infer a repetitive MSC M within the context M_p-M_s.

In the next step, for each pair of MSCs that infers a repetitive MSC, a separate sub-MSC-graph, which is called as a path, is generated and added to the initial MSC-graph G. For example, if M_1 and M_2 infer a repetitive MSC M within the context M_P-M_S, then a path $p = (v_0, M_P, v_1)\ (v_1, M, v_1)\ (v_1, M_S, v_f)$ is generated. We use $src(p)=\{ M_1, M_2 \}$ to indicate the actual observations from which the path p is generated. After checking each combination of two MSCs, which verifies whether the pair infers a repetitive MSC, if a MSC M has never inferred any repetitive MSC, then a path that only includes $p= (v_0, M, v_f)$ is created.

Let us give an example for the execution of the first phase. Suppose that there are four MSCs corresponding to a given set of four observations as input.

$M_a=M_1\ M_4\ M_6$, $M_b=M_1\ M_3\ M_3\ M_4\ M_6$,

$M_c=M_1\ M_4\ M_5\ M_5\ M_5\ M_6$, $M_d=M_1\ M_2\ M_2\ M_3\ M_3\ M_4\ M_6$.

First of all, according to Definition 10 and the algorithm given in Appendix A.1, we check the pair M_a and M_b to see whether the pair infers a basic repetitive MSC or not. Therefore, we first need to calculate the max common prefix and suffix of M_a and M_b, which are M_1 and M_4M_6. After removing the max common prefix and suffix from M_a and M_b, the remaining part of M_a is empty, and the remaining part of M_b is M_3M_3. Therefore, M_a and M_b infer M_3 to be basic repetitive within the context M_1-M_4M_6. Hence $p_1= M_1M_3^{*}M_4M_6$ is inferred as a path and $src(p_1)=\{ M_a, M_b\}$. Similarly, M_a and M_c infer M_5 to be basic

repetitive within the context M_1M_4-M_6. Hence $p_2= M_1M_4M_5^*M_6$ is inferred as another path and src(p_2)={ M_a, M_c}. M_b and M_d infer M_2 to be basic repetitive within the context M_1-$M_3M_3M_4M_6$. Hence $p_3= M_1M_2^*M_3M_3M_4M_6$ is inferred as another path and src(p_3)={M_b, M_d}. We can see that M_b and M_c, M_c and M_d, and M_a and M_d, infer no loop. Therefore, we have tried all combinations of two MSCs in the input set, and there is no other loop left to infer. At the end of first phase, the initial MSC-graph G (Figure 3.1) contains three separate paths p_1, p_2, and p_3, with src(p_1)={ M_a, M_b}, src(p_2)={ M_a, M_c}, and src(p_3)={ M_b, M_d}.

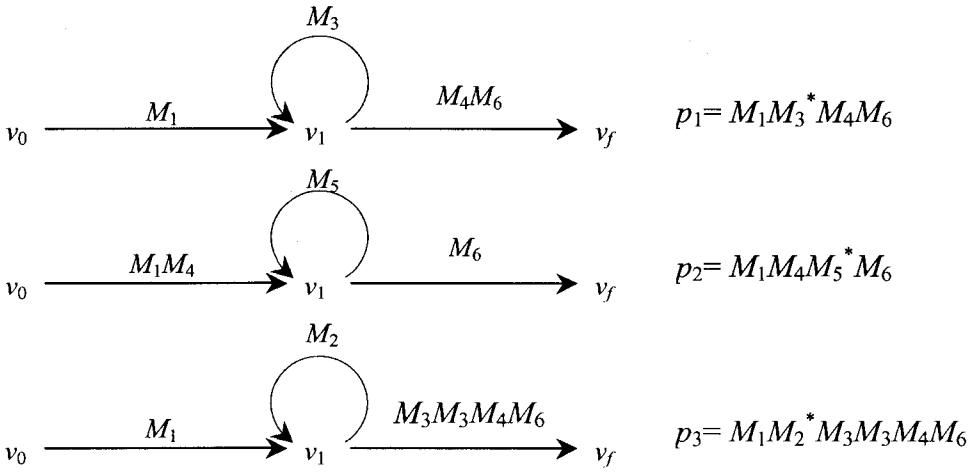

Figure 3.1 The separate paths obtained in the initial MSC-graph

3.2.1.2 Constructing the Final MSC-graph

The second phase constructs the final MSC-graph G' by merging the paths inferred from the first phase. It starts with obtaining "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 $p_1, p_2, \ldots, p_k$ where $p_j \in P$, $1 \leq j \leq k$,

$p = p_1, p' = p_k$, and for $1 \leq i < k$, $\text{src}(p_i) \cap \text{src}(p_{i+1}) \neq \phi$." Next, the paths in each subset P are merged into one new path (the algorithm of merging paths in a partition P is given in Appendix A.3), and all the new paths are inserted to the empty graph G' to construct the final MSC-graph. Note that, in the merging algorithm, every path is considered in the form of three edges: $(v_0, M_P, v_1)(v_1, M, v_1)(v_1, M_S, v_f)$. M_P, M, and M_S are referred as the prefix label, the loop label and the suffix label of the path, and p_m is the merged path accumulated so far. The merging algorithm is based on tracing M_P in p_m which is actually an MSC-graph. In this case, the tracing can be done in linear time due to some assumptions made in this paper. One assumption is that there are no two loops starting at the same node, which means for each node in p_m there are at most two outgoing edges. One of the edges corresponds to loop's body, and the other corresponds to the loop's right-context. Due to another assumption that a repetitive sub-function (loop) has no common prefix with the part of the MSC that succeeds it, there is only a unique edge to proceed when there is no concatenated basic repetitive MSCs. Actually, an assumption that concatenated basic repetitive MSCs don't have common prefix, should be added to make the merging algorithm correct. Because, without the assumption, if concatenated basic repetitive MSC in p_m have common prefix with M_P at the same time, then we don't know which loop to follow when we trace M_P in p_m and we need to try every possibilities. The algorithm of merging paths in a partition P is given in Appendix A.3.

Now, we continue our example above. After the first phase, the resulting initial MSC-graph G contains two paths p_1, p_2, and p_3. Since $\text{src}(p_1) \cap \text{src}(p_2) \neq \phi$ and $\text{src}(p_1) \cap \text{src}(p_3) \neq \phi$, p_1, p_2 and p_3 are in the same partition. Therefore, p_1, $p_2$and p_3 should be merged into one new path $p = M_1 M_2^* M_3^* M_4 M_5^* M_6$, and this new path p is inserted to the final MSC-graph G'. The final MSC-graph obtained is given in Figure 3.2.

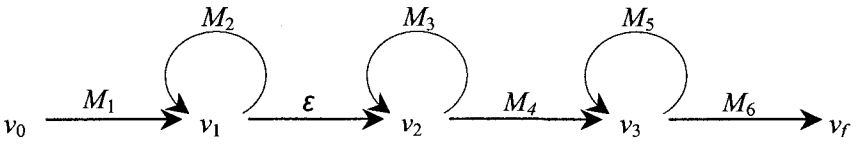

Figure 3.2 The final MSC-graph obtained after processing M_a, M_b, M_c and M_d

This solution also works for nested loops. Suppose that there are three MSCs corresponding to a given set of three observations as input.

$M_a = M_1 M_5$,

$M_b = M_1 M_2 M_4 M_2 M_4 M_5$,

$M_c = M_1 M_2 M_3 M_3 M_4 M_2 M_4 M_5$.

Therefore, M_a and M_b infer M_2M_4 to be basic repetitive within the context M_1-M_5. Hence $p_1 = M_1(M_2M_4)^*M_5$ is inferred as a path and $\text{src}(p_1) = \{ M_a, M_b \}$. M_b and M_c infer M_3 to be basic repetitive within the context M_1M_2-$M_4M_2M_4M_5$. Hence $p_2 = M_1M_2(M_3)^*M_4M_2M_4M_5$ is inferred as another path and $\text{src}(p_2) = \{ M_b, M_c \}$. We can see that M_a and M_c infer no loop. Therefore, we have tried all combinations of two MSCs in the input set, and there is no

other loop left to infer. Since src(p_1) ∩src(p_2)$\neq \phi$, p_1 and p_2 are in the same partition. Therefore, p_1 and p_2 should be merged into one new path p= $M_1(M_2\ M_3{}^*M_4)^*M_5$, and this new path p is inserted to the final MSC-graph G'. The final MSC-graph obtained is given in Figure 3.3.

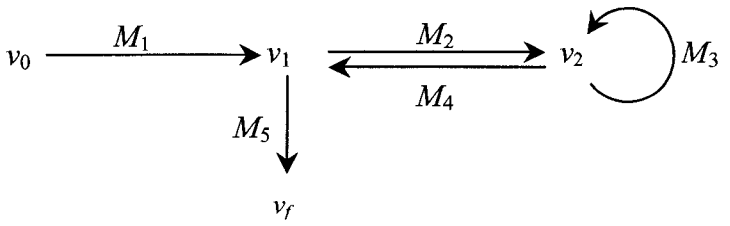

Figure 3.3 The MSC-graph obtained after processing M_a, M_b and M_c

In [Ura04], we need at least two observations to infer a loop in the design. One observation has two or more consecutive occurrences of a basic repetitive MSC, and another one has no occurrences of the basic repetitive MSC. Therefore, every comparison between two MSCs at most infers one basic repetitive MSC. Or we can say, the new basic repetitive MSC must be "introduced" individually [Jou05].

Furthermore, the algorithm proposed in [Ura04] can only work under some restrictive assumptions. In the example of building initial MSC-graph (sub-section 3.2.1.1), there arc four MSCs M_a=$M_1\ M_4\ M_6$, M_b=$M_1\ M_3\ M_3\ M_4\ M_6$, M_c=$M_1\ M_4\ M_5\ M_5\ M_5\ M_6$ and M_d=$M_1\ M_2\ M_2\ M_3\ M_3\ M_4\ M_6$. Originally, M_b and M_d infer M_2 to be basic repetitive within the context M_1-$M_3M_3M_4M_6$. However, if the basic repetitive M_3 iterate three times instead of two times

in M_d, which is $M_d = M_1\ M_2\ M_2\ M_3\ M_3\ M_3\ M_4\ M_6$, then, according to the algorithm, M_b and M_d do not infer any basic repetitive MSC. Therefore, in order to recover the repetitive sub-function represented by M_2 using the algorithm given in [Ura04], the repetitive sub-function represented by M_3 has to be iterated the same number of times in the two observations represented by M_b and M_d. Furthermore, in the merging algorithm, basic repetitive MSCs need to be merged in a specific order, and the ordering of those basic repetitive MSCs need to be totally unambiguous. If not, the algorithm cannot record this ambiguity and waive it later.

3.2.2 Recovering the Lattice of Repetitive Sub-functions

In order to waive some of the above restrictive assumptions, a concept of a lattice of repetitive sub-functions is introduced in [Jou05], which is a structure that provides all possible combinations of n repetitive sub-functions. Using the lattice, the assumptions that repetitive sub-functions have to be iterated the same number of times in each observation; repetitive sub-functions need to be introduced in a specific order; and the ordering of the sub-functions need to be totally unambiguous, are waived.

First, we consider the general case of n repetitive sub-functions without nested repetitive sub-functions. As we explained in the previous section (section 3.2.1), if two MSCs M_a and M_b infer M to be basic repetitive within the context M_P-M_S, then $p_1 = M_P M^* M_S$ is inferred as a path. However, the regular expression $M_P M^* M_S$ could also be seen as a language whose alphabet is the set of MSCs $\{ M_P, M, M_S \}$. Considering the general form representing the

selection of all n repetitive sub-functions, "$M_P.M_1^*.T_1.M_2^*.T_2.M_3^*...M_{n-1}^*.T_{n-1}.M_n^*.M_S$, where $\forall i \leq n$, M_i is a non-empty MSC representing a repetitive sub-function, T_i is a possibly empty 'transition' MSC, M_P is the non-empty prefix and M_S is the non-empty suffix," is also a language whose alphabet is the set of MSCs used in the regular expression $M_P.M_1^*.T_1.M_2^*.T_2.M_3^*...M_{n-1}^*.T_{n-1}.M_n^*.M_S$. If n repetitive sub-functions exist in the presumed design, then 2^n possible combinations of those repetitive sub-function calls could appear in the observations. Therefore, the process of recovering n repetitive sub-functions from a given set of observations is equal to constructing a hypercube of size n with 2^n different languages. The bottom "language" (the bottom of the lattice) is the given initial observation without repetitive sub-function calls, in the form of $M_P.\ T_1.T_2...T_{n-1}.M_S$. All other observations except initial observation should "introduce" a new repetitive sub-function individually. Some languages in the lattice are directly deduced from the observations, and the others are inferred by combining the deduced languages. Therefore, the top "language" (the top of the lattice) includes selections of all repetitive sub-functions, which can be transformed to an MSC-graph as our final constructed design.

In [Jou05], authors propose an algorithm to construct the lattice of repetitive sub-functions (we also call a repetitive sub-function as a loop). First, consider the case without nested repetitive sub-functions, and the ordering of the repetitive sub-functions is never ambiguous. The algorithm starts from the bottom, and then constructs the lattice round by round. A variable *topLabel* is used to store the current top of the lattice, or we can

say the current recovered model of the system. In the first round, letting *topLabel* to be the MSC of the shortest observation of the whole set, compare every observation with *toplabel* to infer a new repetitive sub-function, and then the inferred languages are recorded as successors of *topLabel*. Once all such repetitive sub-functions are deduced, the portion of the lattice is closed by combining all the found successors of *topLabel,* and the top of the lattice is assigned to *topLabel*. Next, each observation that can infer just one repetitive sub-function in addition to the ones that have been already inferred is processed and the generated new languages are recorded as successors of *topLabel*. Then the lattice is closed again, etc. until all the observations are processed. This algorithm is given in Appendix A.6. Note that, the algorithm of checking if an observation (MSC) and *toplabel* infer a repetitive sub-function can be adapted from the algorithm of checking if two MSCs infers a repetitive MSC, which is presented in [Ura04]. However, we must be careful that after the first round, *topLabel* is a regular expression, and actually it is an MSC-graph. Therefore, the functions finding maximal common prefix and suffix presented in [Ura04] must be adapted to trace an MSC in an MSC-graph. In this tracing case, there is only a unique edge in the MSC-graph to proceed due to some assumptions made in this paper. One assumption is that there are no two loops starting at the same node, which means for each node in p_m there are at most two outgoing edges. One of the edges is correspond to loop's body, and the other is corresponds to the loop's right-context. Due to another assumption that a repetitive sub-function (loop) has no common prefix with the part of the MSC that succeeds it, there is only a unique edge to proceed in tracing, which is either follow the loop body or skip it all (we only consider the

case without nested repetitive sub-functions). The algorithm of constructing the lattice of repetitive sub-functions without nested repetitive sub-functions is given in Appendix A.6.

In the above case, we consider maximal common prefix and suffix must not finish inside a loop body. However, in order to infer nested loops, the algorithm should allow maximal common prefix and suffix finish inside a basic repetitive MSC (the loop body). The algorithm given in Appendix A.6, need to be iterated several more times to handle the case with nested loops. The first iteration of the algorithm is performed as before, until the first level basic repetitive MSCs are identified. In the second iteration, the max common prefix and suffix are allowed to stop inside the first level basic repetitive MSCs, until all the second level repetitive MSCs are identified. Then, we go on the iteration, until all the observation are processed. However, when we trace an MSC M in the *topLabel,* once we reach a point where there are several repetitive sub-functions starting there, which have common prefix (resp. suffix) themselves, and also have common prefix (resp. suffix) with the corresponding part of M, we do not know which loop should be traced. Therefore, all the possibility should be tried and follow the one that works. If both maximal common prefix and suffix stop at the same point, then we say this pair of maximal common prefix and suffix is the candidate for the context of nested repetitive sub-functions.

At the end of every round of closing lattice, several repetitive sub-functions may start at the same point. Therefore, the algorithm needs to be adapted to record ambiguity. For example, if repetitive sub-functions M_1 and M_2 start at the same point, $(M_1^* \mid M_2^*)$ is used in

the regular expression to allow alternative. In order to waive the ambiguity generated, the functions finding maximal common prefix and suffix need to be modified as we explained in the last paragraph because we may reach several concatenated repetitive sub-functions as tracing an MSC M inside *topLabel*. If ambiguous branches have been followed during finding maximal common prefix or suffix, then we must record the order in which we have followed these branches because this is the real order of the repetitive sub-functions starting from the same point. Therefore, that specific order moves the ambiguity recorded before.

Let us give an example now, and consider the following six MSCs, corresponding to a given set of four observations:

$M_a=M_1\ M_5\ M_7$, $M_b=M_1\ M_2\ M_3\ M_2\ M_3\ M_5M_7$,

$M_c=M_1\ M_2\ M_4\ M_4\ M_3\ M_2\ M_3\ M_5M_7$, $M_d=M_1\ M_5\ M_6\ M_6M_7$.

Figure 3.4 illustrate the algorithm step by step, where an arrow point to the current *toplabel*. The black dots are deduced from observations or from comparisons between observations and *toplabel*, and the white dots are the languages inferred by inclusion. Initially, topLabel=$M_1M_5M_7$(1). In the first pass through, *topLabel* and M_b find $M_1\ (M_2M_3)^*\ M_5M_7$(2), *topLabel* and M_d find $M_1\ M_5\ M_6{}^*M_7$(3). The lattice, a square, is closed, and $M_1(M_2M_3)^*M_5M_6{}^*M_7$(4) is inferred as top. In the second pass, *topLabel*= $M_1(M_2M_3)^*M_5M_6{}^*M_7$ with M_c find $M_1(M_2\ M_4{}^*M_3)^*M_5M_6{}^*M_7$(5), which include a nested loop. The lattice of size 3, a cube, is closed. $M_1(M_2M_4{}^*M_3)^*M_5M_6{}^*M_7$ is inferred as a final top, and the construction of the lattice of repetitive sub-functions is finished.

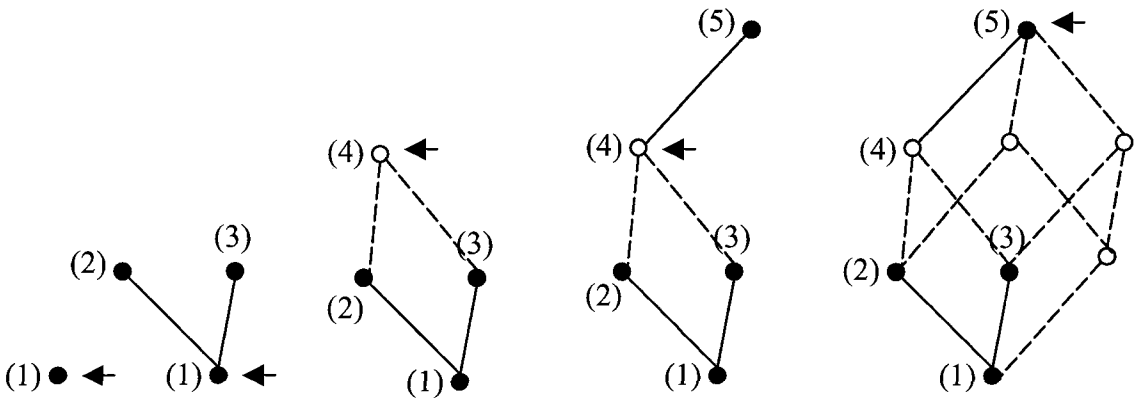

Figure 3.4 Construct the lattice step by step

[Jou05] waived several restrictive assumptions that are used in [Ura04]. However, the assumption that every repetitive sub-function must be introduced "individually", which is the strongest one, still remains.

Chapter Four

The Proposed Solution

In the previous work [Ura04] [Jou05], the proposed methods can recover at most one repetitive sub-function from an observation, so they are not efficient and require a great amount of combinations of repetitive sub-function calls in observations. In this thesis, we revisit the problem and provide an algorithm that is capable of recovering several new basic repetitive sub-functions at once from one given observation by comparing it with currently recovered system model represented by an MSC-graph. The strongest assumption that every repetitive sub-function is introduced "individually" by at least one observation is waived in our proposed method. Therefore, this new method is more efficient and convenient than previous one.

The assumptions used for our proposed method are listed in Table 4.1. Assumptions 1, 2 and 3 are directly adopted from the previous work [Ura04] [Jou05]. However, we do introduce two new assumptions. Assumption 4 is introduced for the efficiency of our proposed algorithm, and assumption 5 is for the correctness of the algorithm. In our new algorithm, we are looking for the beginning and the ending of a repetitive sub-function call by sequentially trying all possibilities until we hit the one that works. Assumption 4, sub-function has a single initiator, can simplify the case of identifying the beginning of a repetitive sub-function call. The assumption is only for efficiency, and the multi-initiator

case can be solved by increasing the complexity. We will explain assumption 4 further in the next section. However, assumption 4 seems fairly reasonable, since a repetitive sub-function is primarily a function, it is usually initiated by a single process, and consequently, it has a single starting point.

Assumption No.	Description
1.	The initial observation (without repetitive sub-function calls), and each repetitive sub-function having nested repetitive sub-functions, has a non-empty, repetitive sub-function free prefix and a non-empty, repetitive sub-function free suffix.
2.	A repetitive sub-function has no common prefix with the part of the MSC that succeeds it and no common suffix with the part of the MSC that precedes it.
3	Repetitive sub-functions starting at the same point do not alternate.
4	Sub-function has 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).
5	A repetitive sub-functions repeat at least twice in its observation.

Table 4.1 Assumptions used for the proposed method

Assumption 5 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

$P.A.B^k.S$ and $P.A^i.B.S$ for $k > 1$ and $i > 1$. 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 readability purpose.

In the following, we first present the main idea of our proposed method, the algorithms given in pseudo code and the corresponding explanation. We then analyze the complexity of the algorithms, and illustrate our approach on an example. Finally the implementation details are described, including the data structure and class graph used in the implementation.

4.1 Repetitive Sub-Function Inference Algorithms

In this section, we propose an algorithm to derive a high-level model represented by an MSC-graph G from a given set of observations of an existing implementation of a distributed system. We consider that each individual function in a distributed system can be viewed as a combination of some sub-functions, and the behavior of a sub-function can be specified by an MSC [Ura04]. Our research focuses on inferring those MSCs that represent repetitive sub-functions. In this whole Chapter, we use O to represent an observation which

is a linearization of an MSC that is not given, and *msc*(*O*) to represent that MSC derived from *O*. We can construct *msc*(*O*) directly from *O* as explained in the Chapter 2.

The algorithm starts with an initial observation *O* that is an observation without any repetitive sub-functions. The initial observation is the shortest of all the provided observations and every other observation is made of the initial observation plus a number of iterations of a number of repetitive sub-functions. We gradually build the system model represented by an MSC-graph, from the rest of a given set of observations. Given the current model represented by an MSC-graph, say *current*, and an MSC *msc*(*O*) that is derived from an observation *O*, we attempt to identify portions in *msc*(*O*), which are coherent with *current* by tracing *msc*(*O*) in *current*, so we can find the parts of *msc*(*O*) that do not match *current,* which are made exclusively of basic repetitive MSCs. Therefore, several basic repetitive MSCs could be inferred at once through comparison of an observation with current model. The final model represented by an MSC-graph is obtained after all the observations are processed.

Tracing *msc*(*O*) in *current* could be very complicated if concatenated basic repetitive MSCs in *current* have common prefix among them. For example, Suppose *current* is an MSC-graph *G'* given in Figure 4.1, $M_2=M_a\,M_b$, $M_3=M_a\,M_c$, $M_4=M_a\,M_d$ that is, *current* = $M_1(M_aM_b)^*(M_aM_c)^*(M_aM_d)^*M_7$. When we reach the node v_1 in *current* where several basic repetitive MSCs M_2, M_3, and M_4 concatenated together, and these MSCs have common

prefix M_a, then we do not know which basic repetitive MSCs to follow because each one can be followed or skipped.

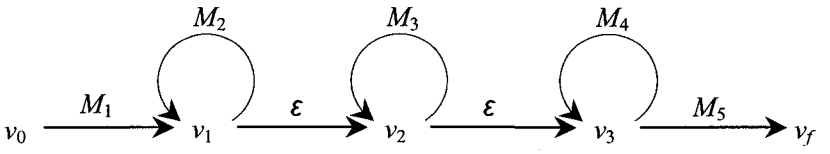

Figure 4.1. An example MSC-graph *G'* with concatenated loops

Thus, for readability purpose, we first consider the case under a temporary assumption: concatenated basic repetitive sub-functions have no common suffix among them. Here, we use common suffix, but not common prefix, because we trace the *current* backward from the exit node to the entry node in our method (the reason of tracing *current* backward is explained in the section 4.1.1). The temporary assumption is used to simplify tracing *msc*(*O*) in *current*, which will be waived later in this chapter.

4.1.1 Main idea

In this section, we explain how basic repetitive MSCs are recovered from *msc*(*O*) by comparing it with current inferred MSC-graph *current*. For readability purpose, first a sketch of our initial idea is described, which traces *msc*(*O*) in *current* from beginning to end. Then we modify our approach a little bit by tracing *msc*(*O*) in *current* from end to beginning, so that the efficiency of the algorithm could be improved by utilizing assumption 4 (sub-function has a single initiator).

The initial sketch is shown as following. We first identify the longest common prefix of *current* and *msc*(*O*) by tracing *msc*(*O*) in *current*. If *msc*(*O*) is already a member of *current*, then the tracing process is terminated, and we can recover nothing from this observation. Otherwise, if *msc*(*O*) is not entirely recognized yet, then we must be looking at the beginning of a repetitive sub-function. The repetitive sub-function will iterate a certain number of times, after which *msc*(*O*) will either "reconnect" with *current* or will enter into a second repetitive sub-function. In any case, it will eventually "reconnect" with *current*. Figure 4.2 illustrate tracing an *msc*(*O*) in an MSC-graph *current*, where *current*=$M_1M_3^*M_4M_7$, and *msc*(*O*)=$M_1M_2M_2M_3M_3M_4M_5M_5M_6M_6M_7$. In this case, the max common prefix M_1 between *current* and *msc*(*O*) stops before M_3^* (just succeeds M_1) in *current*, and before M_2 in *msc*(*O*). The strategy is thus to first look for a possible "connection cut" on *msc*(*O*), which is actually composed of a set of events, one per process and should match the next events (a cut) that follow the max common prefix of *current*, one per process. In the example given in Figure 4.2, the cut that follow the max common prefix M_1 on *current* is {A, B, C} (at the beginning of M_3 in *current*), and then we search for the first possible "connection cut" somewhere on *msc*(*O*), which matches the cut {A, B, C} on *current*. In this whole chapter, we use *cutO* to represent the possible connection cut on *msc*(*O*), and use *cutCurrent* to represent the connection cut on *current*. Once such a cut is found, we should then see if the portion of *msc*(*O*) after max common prefix (M_1) and before possible "connection cut" ({A, B, C}) is made of one or more repetitive sub-functions. If that is not the case, then the possible connection cut does not allow to

conclude and we have to look for another one that would be further down in *msc*(*O*). In the example, we assume that the first cut {A, B, C} fails recovering any basic repetitive MSCs. Therefore we continue searching for the next cut {A, B, C} on *msc*(*O*), which actually allow to recover a basic repetitive MSC M_2.

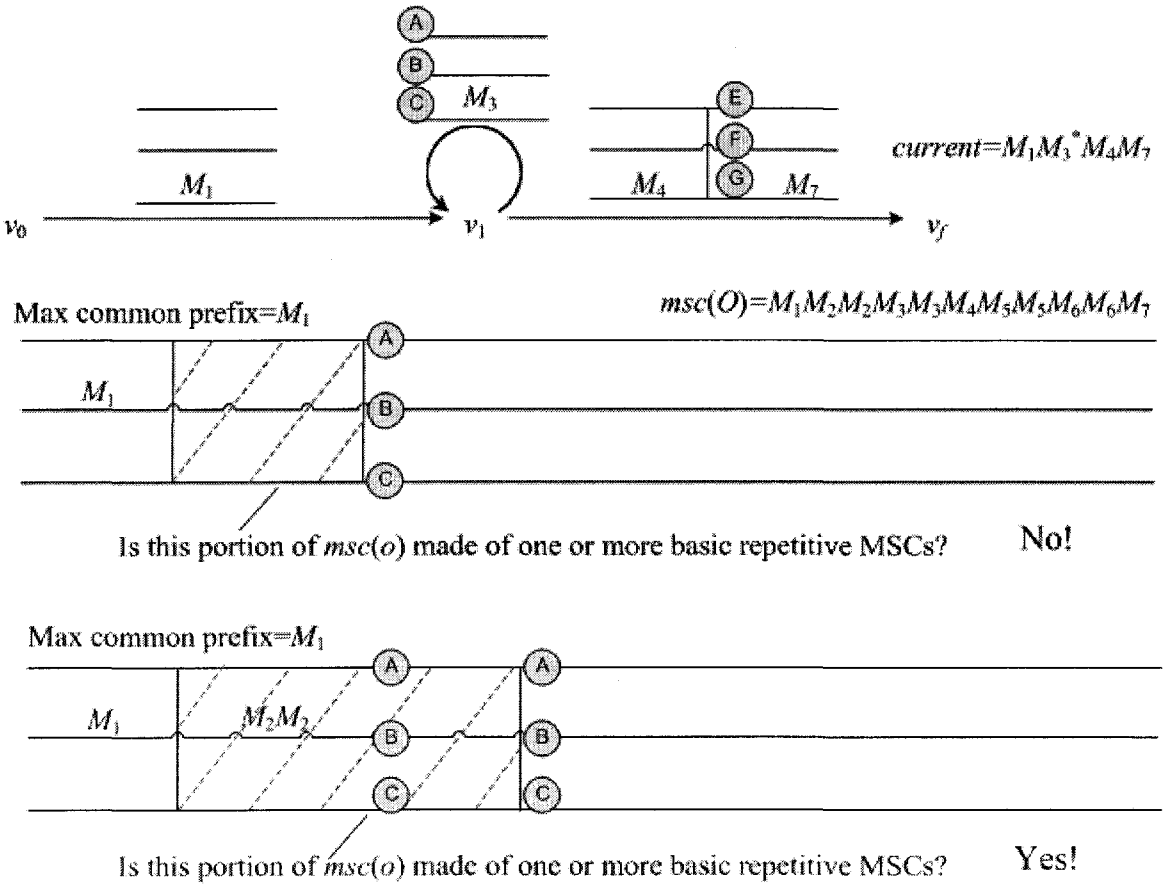

Figure 4.2. Tracing *msc*(*O*) in *current* from beginning to end (a)

Next, we have to see if we can complete the comparison of *current* and *msc*(*O*) starting from that possible connection cut ({A, B, C}) to the end (and possibly find a number of

additional repetitive sub-functions along the way). The simplest way to complete the comparison is to make a recursive call to the same algorithm, starting from that "connection cut" to the end. If the recursive call succeeds in finishing the comparison of *current* and *msc*(*O*), we are done. If not, it means the possible connection cut found on *msc*(*O*) is not true, and we have to look for the true "connection cut" that would be further down in *msc*(*O*). Consequently, in the next step of our example (illustrated in Figure 4.3), the max common prefix $M_3M_3M_4$ is calculated between *current* (begin from the previous connection cut {A, B, C} to the end) and *msc*(*O*) (begin from the previous connection cut {A, B, C} to the end). This time, the new connection cut is {E, F, G} (at the beginning of MSC M_7) on *current*, then we find a possible connection cut {E, F, G} (at the beginning of MSC M_7) on *msc*(*O*), which allow to recover a basic repetitive MSC M_5, followed by another basic repetitive MSC M_6. Further, the max common prefix M_7 is calculated between *current* (begin from the connection cut {E, F, G} to the end) and *msc*(*O*) (begin from the connection cut {E, F, G}to the end), and M_7 is actually at the end of both *current* and *msc*(*O*). Therefore, the comparison between *current* and *msc*(*O*) is successfully finished, and three basic repetitive MSCs M_2, M_5, and M_6 are identified. The resulting MSC graph is given in the Figure 4.3, which is $M_1.M_2^*.M_3^*.M_4.M_5^*.M_6^*.M_7$, and M_2, M_5, and M_6 are newly recovered basic repetitive MSCs. As expected, *msc*(*O*) can be obtained from that graph:

$v_0.(M_1).v_1.(M_2).v_1.(M_2).v_1.(\varepsilon).v_2.(M_3).v_2.(M_3).v_2.(M_4).v_3.(M_5).$

$v_3.(M_5).v_3.(\varepsilon).v_4.(M_6).v_4.(M_6).v_4.(M_7).v_f.$

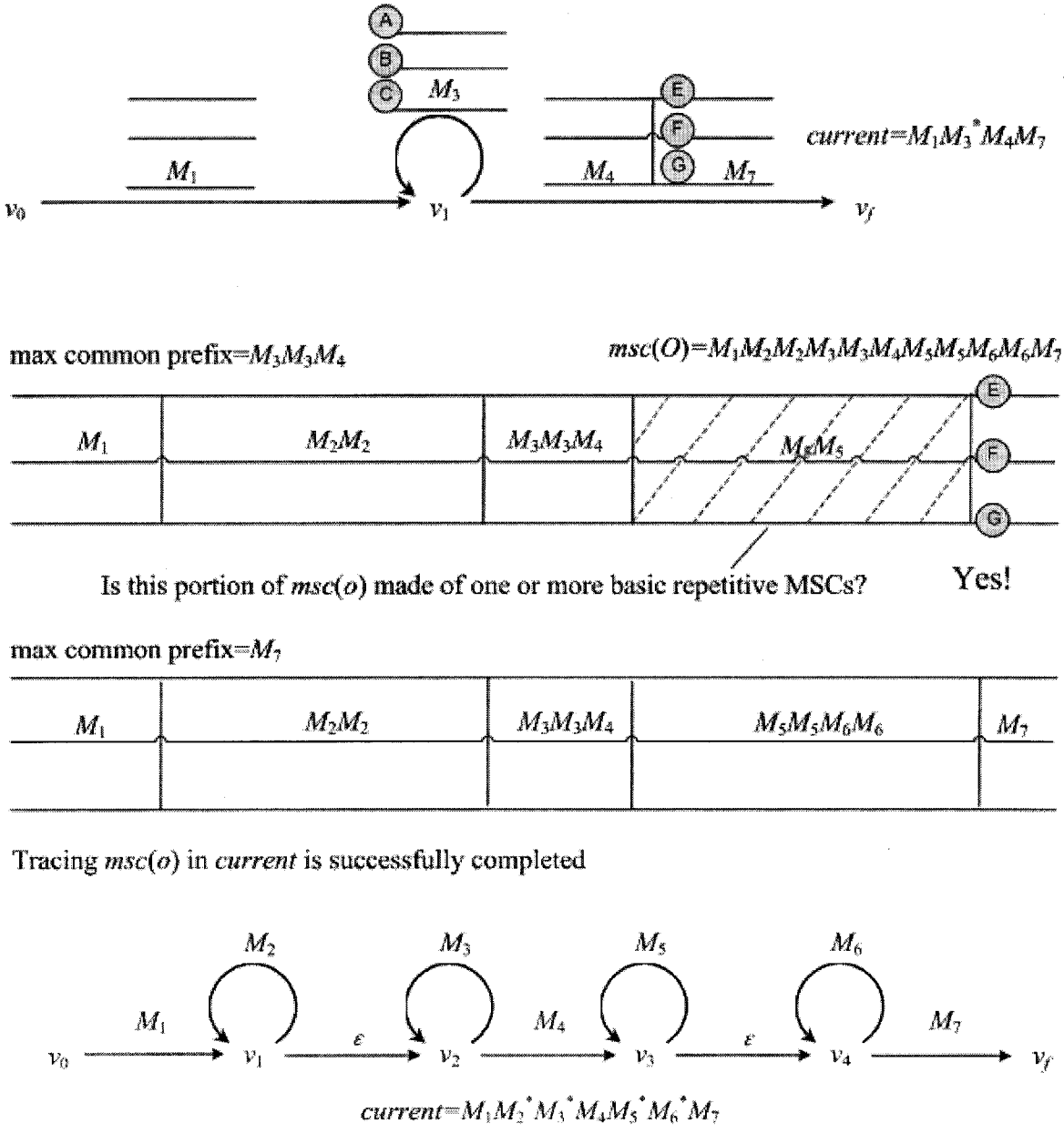

Figure 4.3. Tracing *msc(O)* in *current* from beginning to end (b)

The above sketch achieves the expected result, but can be very inefficient when trying to find the possible connection cut. Indeed, after identifying the maximum common prefix of *msc(O)* and *current*, we know that the next connection cut on *msc(O)* will have to match

the next event on each process of *current*. If these events are not causally related (that is, these are independent events) then any combination of matching events in O can potentially be a "connection cut". 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 combinations.

In order to avoid this combinatorial explosion, we can utilize Assumption 4 stating that repetitive sub-functions have a single initiator. That is, there is always a unique send event e at the beginning of a repetitive sub-function. The algorithm as described cannot benefit from such an assumption, since we locate the "connection cut" at the beginning of a coherent portion, which follows last basic repetitive MSC. If the location of a "connection cut" changed to the end of a coherent portion, and it is followed by next basic repetitive MSC, then this "connection cut" corresponds to the set *Previous*(e), and e is the first send event of the first iteration of the basic repetitive MSC. Therefore, the number of candidates of a "connection cut" is bounded by the number of send events in O. In order to search the connection cut in this way, we need 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.

Figure 4.4 illustrate the procedure of tracing the *msc*(O) in *current* from end to beginning, where *current*=$M_1M_3^*M_4M_7$, and *msc*(O)=$M_1M_2M_2M_3M_3M_4M_5M_5M_6M_6M_7$. We first find the longest common suffix M_7, and the cut on *current* is at the end of M_4, which is {A, B, C}. Next, we need to find the first possible "connection cut" {A, B, C} on *msc*(O),

which should correspond to the set *Previous*(e), and e is the first send event of the first iterate of a basic repetitive MSC. However, right now, we have not identified this repetitive sub-function yet, so we need to try a number of send events moving backward on a linearization of *msc*(O). If, we find a send event e, whose *Previous*(e) is equal to {A, B, C}, then the *Previous*(e) is a possible connection cut on *msc*(O). Further, we verify if this portion of *msc*(O) (the MSC between suffix and possible connection cut) is made of one or more basic repetitive MSCs. If the answer is yes, then recursively call the same algorithm on the remaining part of *current* and *msc*(O). If the recursive call succeeds in finishing the comparison of *current* and *msc*(O), then we add all newly recovered repetitive sub-functions to *current* to finalize the resulting MSC-graph. In the example given in Figure 4.4, after the max common suffix M_7, using the first possible connection cut {A, B, C} on *msc*(O) can not recover any basic repetitive MSC, so the next possible connection cut {A, B, C} further up on *msc*(O) is tried, and two concatenated basic repetitive MSCs M_5 and M_6 are inferred. Next, the algorithm is recursively called on what are left on *current* and *msc*(O). After the max common suffix $M_3M_3M_4$, the single basic repetitive MSC M_2 is recovered. The final resulting MSC graph is given in Figure 4.4, which is $M_1.M_2^*.M_3^*.M_4.M_5^*.M_6^*.M_7$.

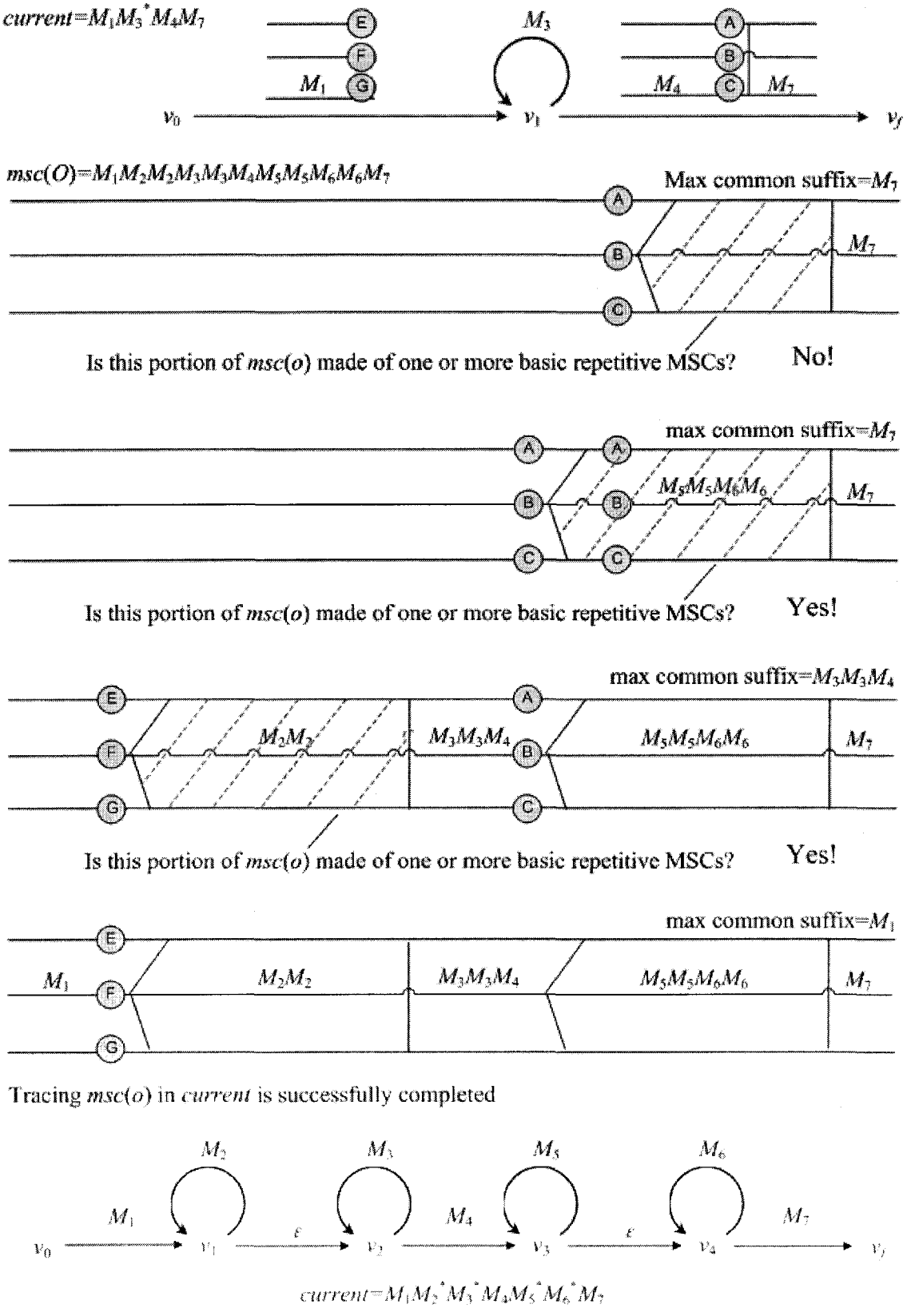

Figure 4.4. Tracing *msc*(*O*) in *current* from end to beginning.

4.1.2 Main Algorithms

Our proposed method contains two main inference algorithms and four sub-functions. Algorithm 1, "initialization and main loop", performs pre-computation and calling algorithm 2 repeatedly to process all MSCs derived from the given observations. Algorithm 2, "infer basic repetitive MSCs", performs the main function, which could infer several new basic repetitive MSCs at once by comparing a given MSC with current recovered model represented by an MSC-graph *current*.

4.1.2.1 Algorithm 1 - Initialization and Main Loop

Algorithm 1 MainLoop
1: *current* := the MSC of the shortest observation
2: Q := a queue of all other observations that are in the form of MSC
3: *KeepGoing*:=*true*
{Precomputation on the set of observations}
4: for all MSCs *msc(O)* $\in$ Q do
5: for all send event e in *msc*(O) moving backward on O do
7: Compute *Previous*(e)
8: end for
9: end for
{Main loop through the observations}
10: while $Q \neq \phi$ AND *KeepGoing*= *true* do
11: *KeepGoing* := *false*;
12: for all MSCs *msc*(O) $\in$ Q do
13: if *InferRepetitive*(*current*, *msc*(O)) then
14 modify *current* to include the newly discovered repetitive sub-function(s)
15: remove *msc(O)* from Q
16: *KeepGoing*:=*true*
17: end if
18: end for
19: end while
{If $Q \neq \phi$, some observations were not handled}

20: if $Q \neq \phi$ then

21: FAILURE: some MSCs were not processed
22: else
23: SUCCESS: the MSC-graph is given as *current*
24: end if

Figure 4.5 Algorithm 1 – Initialization and Main Loop

The variable *current* holds the current recovered model of the system, initialized with the initial observation. Q is a queue of MSCs that are derived from the rest of given observations. The first loop (line 4-9) is a phase of pre-computation on the $msc(O) \in Q$: we pre-compute *Previous*(e) on all send events in $msc(O)$ in the order of their appearance in O that is a linearization of $msc(O)$. The results of pre-computation will be used later in Algorithm 2. Then, the MSCs derived from the given observations are compared with *current* one after the other, so that the basic repetitive MSCs hided in them can be inferred. If new basic repetitive MSC(s) are identified, then we need to modify *current* to include the newly discovered basic repetitive MSC(s). Furthermore, modifying *current* is not only recording the newly recovered MSC(s), and is also a process of recording and waiving the ambiguity of the respective order of concatenated basic repetitive MSCs in *current*. When we recovered a basic repetitive MSC, and do not know its respective order with already recovered basic repetitive MSCs, we can add the newly recovered MSC to *current* and record this ambiguity at the same time. Later, we could waive this ambiguity according to the actual order appeared in other observations.

It may be necessary to compare a given MSC $msc(O)$ with *current* more than once if the MSC includes a nested basic repetitive MSC since *current* might not have included the basic repetitive MSC containing the nested basic repetitive MSC the first time around. For example, suppose *current* has been inferred as $M_1M_5M_6^*M_7$, and a given MSC $M=M_1M_2M_4M_4M_3M_2M_3M_5M_6M_6M_7$, which includes a basic repetitive MSC M_2M_3 containing a nested basic repetitive MSC M_4. Because the basic repetitive MSC M_2M_3 has not been inferred, the algorithm cannot successfully process M the first time around. In order to recover the nested basic repetitive MSC M_4, the given MSC M need to be processed for another time after the basic repetitive MSC M_2M_3 are inferred and added to *current*. After the while loop (line 10-18), the MSC-graph *current* is inferred as the high level model of the system. If there are still some observations left in the queue Q at the end, the reason could be that those observations do not provide enough evidence to be processed. For example, if an observation O_1 contains a nested loop M_1 in a loop M_2, and M_2 is never inferred by other observations, so O_1 can not be successfully processed in the algorithm. Another reason could be that those observation are collected incorrectly because of some errors happened in the collection process.

4.1.2.2 Algorithm 2 - Infer basic repetitive MSCs

Algorithm 2 can infer several basic repetitive MSCs at once from an MSC *msc*(*O*) by comparing *msc*(*O*) with current model represented by the MSC-graph *current*, and the algorithm performs the main function of our proposed method. It is necessary to explain some variables below. *cutCurrent* and *cutO* are the output parameters of function *FindMaximumSuffix*(). *cutCurrent* actually represents the "connection cut" that we described in section 4.11 main idea. It is a cut of an MSC that is labeled on some edge of *current*, and it just proceeds the calculated max common suffix in *current*. *cutO* is a cut of *msc*(*O*), and it just proceeds the max common suffix in *msc*(*O*). *connectionPoint* is the first send event *e* that succeeds the "connection cut" in *msc*(*O*). *tracingResult* is the return parameter of function *FindMaximumSuffix*(), and three integer values could possibly be returned: 1, 2 and 3. In the first condition (return value is 1), max_common_suffix on *current* ends at the entry node v_0, and max_common_suffix on *msc*(*O*) are the entire *msc*(*O*), so *msc*(*O*) is a member of *current*. In the second condition (return value is 2), max_common_suffix on *current* ends at the entry node v_0, but max_common_suffix on *msc*(*O*) are not the entire *msc*(*O*); or max_common_suffix on *current* ends at a node (except the entry node v_0) or in the middle of an edge, but max_common_suffix on *msc*(*O*) are the entire *msc*(*O*). In the third condition (return value is 3), max_common_suffix on *current* ends at a node (except the entry node v_0) or in the middle of an edge, and max_common_suffix on *msc*(*O*) are not the entire *msc*(*O*).

Algorithm 2 Boolean InferRepetitive(IN-OUT MSC-graph *current*, IN MSC *msc*(*O*))

1: int *tracingResult* =*FindMaximumSuffix* (*current*, *msc*(*O*), *cutCurrent*, *cutO*)
2: if *tracingResult* = 1 then
3: return *true*
4: else if *tracingResult* = 2 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*, *msc*(*O*), *startingCut*, *connectionPoint*)
10: if *connectionPoint*=ϕ then
11: return false
12: end if
13: if *IsMadeOfBasicRepetitives* (*msc*(*O*), *connectionPoint* , *cutO*) then
14: if *InferRepetitive*(*current*[*cutCurrent*], *msc*(*O*)[*Previous*(*connectionPoint*)]) then
15: return true
16: end if
17: 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 tracing *O* inside current. We keep looping.}
18: *startingCut* := *Previous*(*connectionPoint*)
19: end while

Figure 4.6 Algorithm 2 – Infer basic repetitive MSCs

The algorithm first call *FindMaximumSuffix*(IN *current*, *msc*(*O*), OUT *cutCurrent*, *cutO*) on *current* and *msc*(*O*), and the location (starting) of this max common suffix is returned in *cutCurrent* and *cutO*. If return value is 1, then *msc*(*O*) is a member of *current*, and the algorithm terminates and returns *true*. If return value is 2, only one of *cutCurrent* or *cutO* is at the beginning, then *msc*(*O*) do not infer a loop in this time of comparison with *current*, so the algorithm terminates and returns false. If return value is 3, then one or

several new basic repetitive MSCs might end before *cutO*. Next, we need to look for a send event *e* that might be the initiator of the first new basic repetitive MSCs and then we can see if this portion of *msc*(*O*) is really made of basic repetitive MSCs. In order to find the initiator of the first new basic repetitive MSC, first we need to search for the possible "connection cut" where *current* and *msc*(*O*) might reconnect again because the initiator should just succeed the "connection cut" on *msc*(*O*). If we cannot find a "connection cut" backward along *O* which is a linear extension of *msc*(*O*), then the algorithm terminates. If we can find a possible "connection cut" which is some previous(*e*) and is equal to *cutCurrent* at the same time, then this *e* is returned as *connectionPoint* by sub-function *FindNextConnectionPoint*(IN *cutCurrent*, *msc*(*O*), *startingCut*, OUT *connectionPoint*). Further, we call *IsMadeOfBasicRepetitives* (*msc*(*O*), *connectionPoint* , *cutO*) to check if the MSC between *connectionPoint* and *cutO* of *msc*(*O*) contains one or more basic repetitive MSCs. Furthermore, we recursively call the function *InferRepetitive*() on the *current* and *msc*(*O*) beginning from that "connection cut". The algorithm finally will terminate itself, either successfully or unsuccessfully inferring new basic repetitive MSCs. In the unsuccessful condition, after execution of the function *FindMaximumSuffix*(), if only one of *cutCurrent* or *cutO* is at the beginning (return value is 2), then this time of tracing is failed. If all possibilities of connection point are tried out without identifying any new basic repetitive MSCs, then the whole algorithm terminates. Algorithm 2 is illustrated by two examples in section 4.3.

4.1.3 Infer Basic Repetitive Sub-functions

There are several sub-functions called by algorithm 1 and 2, including *FindMaximumSuffix*(), *FindNextConnectionPoint*(), *IsMadeOfBasicRepetitives*(), and *Previous*(*e*). The algorithms of these sub-functions are given in the following sub-sections.

4.1.3.1 Algorithm 3 - Find Maximum Suffix

Finding the maximum common suffix of *current* and *msc*(*O*) is based on tracing *msc*(*O*) in *current*. *current* is actually an MSC-graph, and contains two kinds of edges, transition edges and loop edges. Transition edges can be labeled by ε , or we can say several loops could be concatenated together. The tracing begins from the end of *current* and *msc*(*O*). Here, when we trace *msc*(*O*) in a specific edge of *current*, we use the algorithm in [Ura04], finding maximal common suffix to find the maximal common suffix between the MSC labeled on an edge of *current* and the specific part of *msc*(*O*). Below, we use max_common_suffix to refer max common suffix between *current* and *msc*(*O*).

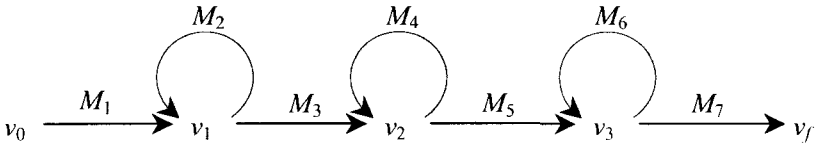

Figure 4.7 An example MSC-graph G_2 without concatenated loops

First, we consider the case without concatenated loops in *current*, and an example MSC-graph is given in Figure 4.7. Because no two loops start from the same node, there are at most two outgoing edges and two incoming edge for each node. In fact, only entry node

v_0 has one outgoing edge and exit node v_f has one incoming edge, and other nodes in *current* all have two outgoing edges: one of the edges is a loop edge, and the other is a "transition" edge. Furthermore, they all have two incoming edges, one is a loop edge, and the other is a "transition" edge. Because we trace backward in *current*, we actually trace incoming edges of nodes of MSC-graph *current*. Due to the assumption that a repetitive MSC (loop) has no common suffix with the part of the MSC (transition MSC) that proceeds it, there is only a unique edge to proceed for a specific node when we trace *msc*(O) in *current*. For example, in Figure 4.7, according to the assumption above, M_5 and M_6 do not have common suffix, so we immediately know which edge to follow when we reach node v_3 from node v_f.

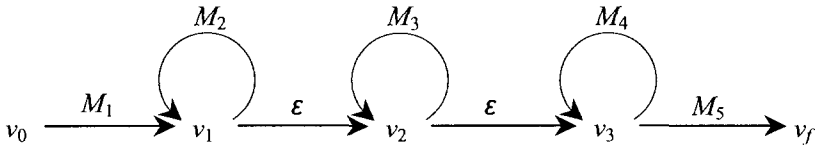

Figure 4.8 An example MSC-graph G_3 with concatenated loops

Next, we consider the case of concatenated loops in *current*, and an example MSC-graph is given in Figure 4.8. We can use the temporary assumption that we made at the beginning of section 4.1 to simplify the tracing, which is that concatenated basic repetitive MSCs do not have common suffix. For example, in Figure 4.8, according to the assumption, M_2, M_3 and M_4 do not have common suffix, so we immediately know which edge to follow when we reach node v_3 from node v_f.

Tracing *msc*(O) in *current* could end in three different conditions as we described in the last section. In the first condition, max_common_suffix on *current* ends at the entry node v_0, and max_common_suffix on *msc*(O) are the entire *msc*(O). In the second condition, max_common_suffix on *current* ends at the entry node v_0, but max_common_suffix on *msc*(O) are not the entire *msc*(O); or max_common_suffix on *current* ends at a node (except v_0) or in the middle of an edge, but max_common_suffix on *msc*(O) are the entire *msc*(O). In the third condition, max_common_suffix on *current* ends at a node (except v_0) or in the middle of an edge, and max_common_suffix on *msc*(O) are not the entire *msc*(O). The algorithm of *FindMaximumSuffix*() returns a corresponding integer number to represent the certain ending condition. The algorithm of *FindMaximumSuffix*() is given below:

Algorithm 3 Int FindMaximumSuffix(IN MSC-graph *current*, MSC *msc(O)*, OUT cut *cutCurrent*, cut *cutO*)

```
1:  currentEdge=the edge leading to exit node v_f in current
2:  M_o:= msc(O)
3:  while true do
    {Let currentEdge be (v_x, M, v_y)}
4:      If M≠ null then
5:          M_1:= MaxCommonSuffix(M, M_o);
6:          if M_1= null then
7:              if v_x= v_y then
                    {Basic repetitive MSC could be skipped}
8:                  currentEdge:= the next edge in current
9:              else
                    {Tracing ends. Condition 3: max_common_suffix on current ends at
                    node v_y, and max_common_suffix on msc(O) are the portion being
                    moved from M_o }
10:                 cutCurrent:= the cut just proceed the max common suffix on current
11:                 cutO:= the cut just proceed the max common suffix on msc(O)
12:                 return 3
13:             end if
14:         else if M_1= =M then
```

```
                {We have traced the entire M}
15:             Move M1 from Mo
16:             if Mo = null then
17:                 if currentEdge=the edge beginning from entry node v0 then
                        {Tracing ends. Condition 1: max_common_suffix on current
                        ends at node v0, and max_common_suffix on msc(O) are the
                        entire msc(O)}
18:                     cutCurrent= φ
19:                     cutO = φ
20:                     return 1
21:                 else
                        {Tracing ends. Condition 2: max_common_suffix on current
                        does not end at node v0, but max_common_suffix on msc(O) are
                        the entire msc(O)}
22:                     return 2
                    end if
23:             end if
24:             If vx ≠ vy then
25:              If currentEdge= the edge beginning from entry node v0 then
                    {Tracing ends. Condition 2: max_common_suffix on current ends at
                    node v0, but max_common_suffix on msc(O) are not the entire
                    msc(O)}
26:                 cutCurrent= φ
27:                 cutO = φ
28:                 return 2
29:              end if
                    currentEdge= the next edge of current
31:             end if
32:         else
                {Tracing ends. Condition 3: max_common_suffix on current ends in the
                middle of currentEdge, and max_common_suffix on msc(O) are M1 plus
                the portion being moved from Mo }
33:              cutCurrent:= the cut just proceed the max common suffix on current
34:              cutO:= the cut just proceed the max common suffix on msc(O)
35:              return 3
36:         end if
37:     else
                {skip ε edge}
38:             currentEdge= the next edge in current
39:     end if
40: end while
```

Figure 4.9 Algorithm 3 – Find Maximum Suffix

4.1.3.2 Algorithm 4 - Find Next Possible Connection Point

At the beginning of the section 4.1, we have talked about "connection cut" on *msc*(*O*), where *msc*(*O*) reconnect with *current* after a portion of one or several concatenated basic repetitive MSCs on *msc*(*O*). Therefore, "connection cut" is located just proceed those one or several concatenated basic repetitive MSCs. "connection point" is the first send event *e* of those basic repetitive MSCs (we have an assumption that sub-function only has a single initiator), and that send event *e* is just succeed the "connection cut". The algorithm 4 *FindNextConnectionPoint*() implements finding next possible "connection point", which is the possible starting point of one or several basic repetitive MSC. According to the description above, the possible "connection point" is a send event, it is just succeed the "connection cut" on *msc*(*O*), and the "connection cut" must match *cutCurrent* (the output parameter of function *FindMaximumSuffix*) exactly. In order to find such "connection point", we simply have to search backward on *msc*(*O*) in the order of the linear extension for a send event *e* so that *Previous*(*e*) matches *currentCut*. This search is begin from the *startingCut*, which is assigned by the value of the output parameter *cutO* of function *FindMaximumSuffix*(). If such a send event *e* is not found after searching all send events in *msc*(*O*) before *startingCut*, then no such "connection point" exist. The algorithm is given below:

Algorithm 4 Void FindNextConnectionPoint(IN cut *cutCurrent*, MSC *msc(O)*, cut *startingCut*, OUT event *connectionPoint*)

1: for all send event $e \in$ *msc(O)* before *startingCut*, moving backward on *O* do

2: if *Previous*(*e*) = the cut just preceeds *cutCurrent* then

3: *connectionPoint* := *e*

4: return

5: end if

6: end for

{Connection point not found}

7: *connectionPoint*:= ϕ

Figure 4.10 Algorithm 4 – Find Next Connection Point

4.1.3.3 Algorithm 5 - Infer Concatenated Basic Repetitive MSCs

In Algorithm 2, we extract a segment *S* of *msc(O)*, which is not present in *current*. We must now see if this segment is made of one or more repetitive sub-functions. In [Ura04], a linear time algorithm (function *BasicRepetitiveMSC*()) is provided to find the existence of a basic repetitive MSC *M* of a given MSC *M'*. This algorithm is used to decide whether or not a given MSC contains the repetition of one basic MSC. Under the present assumptions, the segment *S* of *msc(O)* can be the concatenation of more than one basic repetitive MSCs, that is, *S* could be of the form $M_1^{k_1} M_2^{k_2} \ldots M_p^{k_p}$, for $p \geq 1$ and $k_1 \geq 2$, $k_2 \geq 2$, ... $k_p \geq 2$. Therefore, our new algorithm try to find a portion that may only contains repetition of one basic MSC, and then call the original algorithm in [Ura04] to confirm our guess.

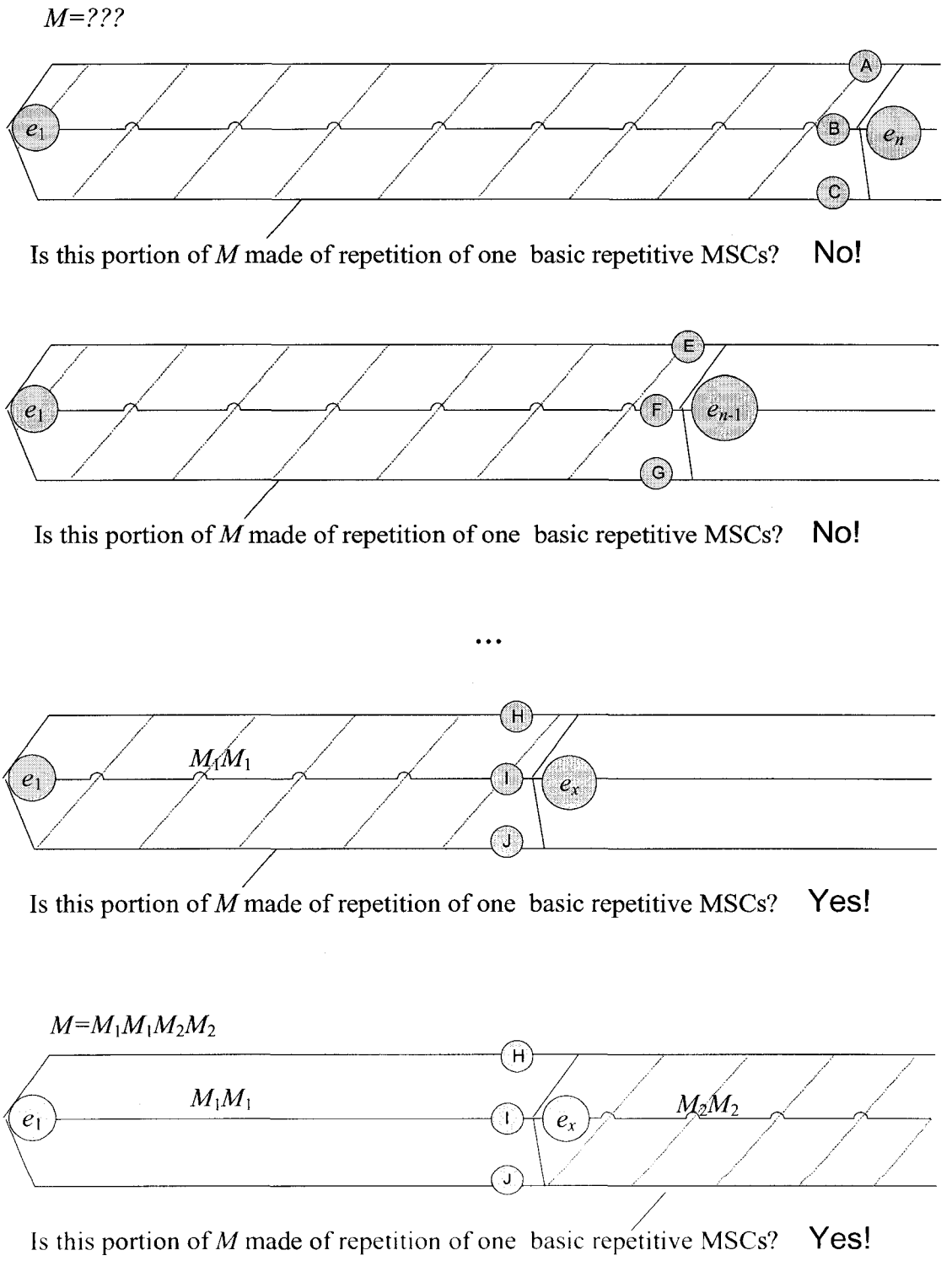

Figure 4.11. Try each send event *e* from end to beginning

Our approach to solve 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 4 stating that repetitive sub-functions have a single initiator, which allows speeding up the search quite dramatically. For example (Figure 4.11), suppose there exist two basic repetitive MSCs in segment S, the repetition of M_1 is followed by the repetition of M_2, and they are initiated from send event e_1 and e_x separately. The repetition of M_1 must begin from send event e_1 and end at previous(e_x), and what is left in S is the repetition of M_2. So the problem of separating the repetition of M_1 and M_2 is transformed to locating e_x (e_1 is at the beginning of the segment S). Suppose there are n send events in the segment S, so the location of e_x has at most n possibilities. According to the order given by the linear extension of S, we try each possible send event e from the end of S backward by calling *BasicRepetitiveMSC*() on the MSC between e_1 and previous(e). Once the function returns a basic repetitive MSC, we stop searching, and this send event e is probably the e_x that is the initiator of M_2. Next, we call *BasicRepetitiveMSC*() on the MSC begin from the send event e to the end of S to confirm it contains another basic repetitive MSC. If the function returns another basic repetitive MSC, then we have recovered two concatenated repetitive MSC successfully. If not, then there are two possibilities. One is that the MSC beginning from the send event e to the end of S contains two or more repetitive MSCs. If this is the case, we continue to separate those concatenated MSCs. Another possibility is that the send event e that we find in the last step

is not e_x, which means we stop searching e_x too early. Therefore, we continue to search e_x from where we stopped until we can recover M_1 and M_2 successfully. If we tried all possibilities, and we still fail to find that the segment S is completely made of one or more basic repetitive MSCs, then result is returned as *false*. The procedure of locating e_x is illustrated in Figure 4.11.

Algorithm 5 Boolean *IsMadeOfBasicRepetitives* (IN *msc*(*O*), *connectionPoint*, *cutO* Out *basicRepetitives*)

1: if *BasicRepetitiveMSC* (*msc*(*O*) [*connectionPoint*, *cutO*]) then
2: return *true*
3: end if
4: for all send event $e \in msc(O)$ between *connectionPoint* and *cutO*, moving backward on *O* do
5: *result*:=ϕ
result:=*BasicRepetitiveMSC*(*msc*(*O*) [*connectionPoint*, *Previous*(*e*)]) then
if *result* $\neq \phi$ then
add the newly recovered basic repetitive MSC *result* into *basicRepetitives*
6: if *IsMadeOfBasicRepetitives*(*msc*(*O*), *e*, *cutO*) then
7: return *true*
8: end if
9: end if
10: end for
11: return *false*

Figure 4.12 Algorithm 5 – Infer concatenated basic repetitive MSCs

4.1.3.4 Algorithm 6 - Compute the Set *Previous*(*e*)

We have given the definition of the set *Previous*(*e*) in the preliminaries. According to that definition, a given MSC M is an MSC with k processes $\{P_1, P_2, \ldots, P_K\}$, and e is a send event of M. The set *Previous*(*e*) contains k events, one per process of M, which do not happen after e and that are maximal on their process with that property. The following algorithm is used to compute the set *Previous*(*e*) from a given MSC M, and this sub-function return a cut on M, which is a set of events, one per process. We use $s.m_{x,i,j}$ to represent sending of message m_x by p_i to p_j.

The main idea of this algorithm is that all the events that happen after the send event $s.m_{x,i,j}$ are marked, the last unmarked events, one per process, are returned as *Previous*($s.m_{x,i,j}$). According to definition 4 in the preliminary, when we mark an event e on a process P_i, we must mark all the events that follow e along the process P_i. And if e is a send event $s.m_{y,i,j}$, we must mark the corresponding receive event $r.m_{y,i,j}$ as well, because a receive event always follows its send event.

Algorithm 6 *Cut Previous*(IN event $s.m_{x,i,j}$)

1: Mark $s.m_{x,i,j}$ and $r.m_{x,i,j}$
2: Mark the events that follow $s.m_{x,i,j}$ along the process i
3: Mark the events that follow $r.m_{x,i,j}$ along the process j
4: For all marked send events do
5: mark the corresponding unmarked receive event
6: end for

7: While there are unmarked events following marked events do

8: For all P_i do

9: If there are unmarked events following marked events on P_i then

10: mark the unmarked events and corresponding unmarked receive events

11: end if

12: End for

13: End While

14: Return the last unmarked event one per process as *Previous*($s.m_{x,i,j}$)

Figure 4.13 Algorithm 6 – Compute the set *Previous*(*e*)

4.1.4 Recap of proposed algorithm

In our proposed algorithm, the initial observation without any repetitive sub-function calls is transformed to an initial system model represented by an MSC-graph G. Next, we gradually build the MSC-graph G by recovering basic repetitive MSCs from a set of MSCs transformed from other given observations. Our algorithm is capable of identifying repetitive patterns and repetitive sub-patterns. The language of the MSC-graph G derived consists of all the MSCs corresponding to the given observations and the inferred observations. In the resulting MSC-graph G, the basic repetitive MSCs and their relative positions are all explored. Therefore, the resulting MSC-graph G is a high-level design model of the existing system.

Because this is a general algorithm without referencing any specific distributed system, we could only try out all possibilities to recover the repetitive pattern in an individual

functionality. Algorithm 2 terminates either after the new basic repetitive MSCs are successfully inferred, or all possibilities are try out, but fail to recover anything. Algorithm 1 terminates when all observations are processed (each observation could be processed more than once) and cannot recover any more basic repetitive MSCs.

4.2 Complexity of the Algorithm

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 an observation of an individual functionality contains up to n events. There are up to p observations, and the total number of the events in all observation is m. Clearly, $m \in O(p.n)$.

4.2.1 Complexity of the Algorithm 5 (Infer Concatenated Basic Repetitive MSCs)

Function BasicRepetitiveMSC() can be made to run in $O(n)$ [Ura04], and this algorithm is called up to n times in the for loop (line #4). In addition, one should note that it is not necessary to recursively call IsMadeOfBasicRepetitives() with the same *connectionPoint–cutO* arguments twice, since the algorithm would always return the same result. Actually only if IsMadeOfBasicRepetitives() with an *connectionPoint–cutO* arguments return false, then there is a possibility that the same *connectionPoint–cutO* is about to be called a second time, since the recursive call terminates immediately when a

connectionPoint–cutO call returns true. It is thus possible to record the fact that a particular *connectionPoint–cutO* was already computed, so we can avoid a recursive call when this is the case. Whether a *connectionPoint–cutO* has already been called can be checked in $O(n)$ and the number of recursive calls can be limited to a maximum of n because there is only n send events. Without recursive call, the complexity of the algorithm is $O(n^2)$, and complexity of the number of recursive calls is $O(n)$. Therefore, Algorithm IsMadeOfBasicRepetitives can be implemented to run in $O(n^3)$. We can now evaluate the complexity of Algorithm 2.

4.2.2 Complexity of the Algorithm 2 (Infer several basic repetitive MSCs at once)

The algorithm 4 *FindNextConnectionPoint* can be implemented to run in $O(n)$ because we at most try n send event e in the order of linear extension. We have known that the algorithm 4 *IsMadeOfBasicRepetitives* runs in $O(n^3)$, so the only missing information is the number of recursive calls to *InferRepetitive*. To calculate the number of recursive calls, 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 is 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 $O(n + k.m)$. The while true loop (line #8) can be run at most n iterations, so each complete run of one call of *InferRepetitive*(excluding recursive calls) can be completed in $(O(n^3) + O(n+k.m)) * O(n)$,

which is $O(n^4 + k.n.m)$. Algorithm *InferRepetitive* can be implemented to run in $O(n^4 + k.n.m) * O(n.\ m^k)$, which is $O(n^5.m^k + k.n^2.m^{k+1})$.

4.2.3 Complexity of the Algorithm 1 (Initialization and Main Loop)

In the algorithm 1, in the while loop (line #10), the algorithm 2 *InferRepetitive* can be repeatedly run $O(p^2)$ times in the worst case. We have shown that the time complexity of algorithm 2 *InferRepetitive* is $O(n^5.m^k + k.n^2.m^{k+1})$, so the complexity of algorithm 1 is $O(p^2.n^5.m^k + p^2.k.n^2.m^{k+1})$. That is to say our proposed method can be made to run in $\boldsymbol{O(p^2.n^5.m^k + p^2.k.n^2.m^{k+1})}$.

4.3 Waive the temporary assumption

At the beginning of this chapter, we assume that concatenated basic repetitive MSCs do not have common suffix. Under the assumption, there is only a unique edge to proceed for a specific node when we trace *msc*(*O*) in *current* backward. However, without the assumption, if we reach a node where several basic repetitive MSCs concatenating together, and having common suffix among themselves and corresponding part of *msc*(*O*), then we do not know which basic repetitive MSC should be followed. In this sub-section, we are going to handle this special case based on the original algorithm 3 *FindMaximumSuffix*, modified algorithm 3 is given in section 4.3.2. Same as the original algorithm, when we trace *msc*(*O*) in a specific edge of *current*, we use the algorithm in [Ura04], finding maximal common suffix to calculate the maximal common suffix between the MSC labeled on an edge of *current* and

the specific part of *msc*(*O*). Below, we use *max_common_suffix* to refer max common suffix between *current* and *msc*(*O*).

4.3.1 The main idea

In the process of finding max common suffix, when we reach a node in *current*, where several basic repetitive MSCs concatenated together, and having common suffix among them, several tracing paths could possibly be followed in *current* by skipping different basic repetitive MSCs. The main idea of our solution is that we trace *msc*(*O*) in *current* by all possible tracing paths at the same time, but only trace backward one step at a time. Here, one step means execute the algorithm finding maximal common suffix [Ura04] on the MSC labeled on an edge of *current* and the specific part of *msc*(*O*) once. Because every basic repetitive MSC is different from each other, at most one tracing path can go though to the next step without termination, and other tracing paths must all terminate after one step tracing. Consequently, the corresponding *max_common_suffix* stops too. Next, the results (pairs of *cutO* and *cutCurrent*) of all terminated paths are returned, and the path that has not terminated is recorded in case of returned results do not work. Finally, the pairs of *cutO* and *cutCurrent* (return values of algorithm 3) are passed in algorithm 2, and all results in the order of termination are tried to recover new basic repetitive MSCs from *msc*(*O*). If one of them success, then remaining ones are give up. If all passed pairs of *cutO* and *cutCurrent* fail, then we must continue to trace the path (the only one) that has not terminated in the previous step of finding maximum suffix to get the result... until we successful find the right one.

For example, suppose *current* is an MSC-graph given in Figure 4.14, $M_6=M_b\,M_a$, $M_5=M_c\,M_a$, $M_3=M_e\,M_d$, $M_2=M_f\,M_d$, that is, *current*= $M_1\,(M_f\,M_d)^*(M_e\,M_d)^*M_4\,(M_c\,M_a)^*(M_b\,M_a)^*M_7$. Suppose $msc(O)= M_1\,M_f\,M'\,M'\,M_d\,M_4\,M_b\,M_a\,M_7$.

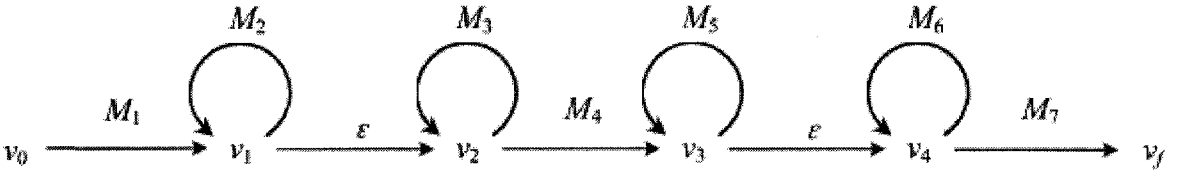

Figure 4.14. An example MSC-graph with concatenated loops

The tracing begins from the end of *current* and $msc(O)$. First we compare $msc(O) = M_1\,M_f\,M'\,M'\,M_d\,M_4\,M_b\,M_a\,M_7$ with M_7 that is labeled on the edge leading to v_f. Because M_7 is also the suffix of $msc(O)$, M_7 becomes a part of *max_common_suffix*. Next, we remove M_7 from $msc(O)$, and $msc(O)= M_1\,M_f\,M'\,M'\,M_d\,M_4\,M_b\,M_a$, and we reach the node v_4 in *current* at the same time. Then, in *current* we find the concatenated basic repetitive MSC $M_6=M_b\,M_a$ and $M_5=M_c\,M_a$ concatenated together, and they have a common suffix M_a at the same time. Because M_a is also at the end of $msc(O)$, we can not know immediately whether to follow M_6 or M_5. Therefore, we decide to follow both at the same time: the first tracing path begins from M_6, and the second one begins from M_5 by skipping M_6. The important thing is that we only trace both basic repetitive MSCs one step at a time. After one step tracing, we return all terminated tracing results. If one of these results is good, then we finish tracing. If none of the results is good, we go back to follow the one that has not terminated. Now, we trace M_6 and

M_5 only one step. By tracing $M_6 = M_b M_a$, the current *max_common_suffix*= $M_b\, M_a\, M_7$, tracing do not terminate and could be continued later. By tracing $M_5 = M_c M_a$, *max_common_suffix*= $M_a M_7$, tracing stops: in *current* it stops at the middle of M_5; in *msc*(*O*) it stops just succeed M_b. Therefore, we calculate the result of tracing M_5 first. The value of corresponding *cutO* and *cutCurrent* are the cut just succeed M_b in *msc*(*O*) and the cut on M_5 in *current*. Further we need to look for a next possible connection point on *msc*(*O*) to check if the portion between *cutO* and the connection point on *msc*(*O*) is made of one or more basic repetitive MSCs. However, we can not find any possible connection point on *msc*(*O*), so we immediately know the path we followed is not the right path. Therefore, we go back to trace another path instead. After the prior tracing, *msc*(*O*)= $M_1\, M_f M'\, M'\, M_d\, M_4$, and we are still at the node v_4 in *current*. Next, because M_4 (at the end of *msc*(*O*)) have no common suffix with M_5 and M_6, obviously we skip M_5 and M_6, then we find the common part M_4 between *msc*(*O*) and *current*. So we remove M_4 from *msc*(*O*), *msc*(*O*)= $M_1\, M_f M'\, M'\, M_d$, and we reach node v_2 in *current* at the same time. Now, we met another two concatenated basic repetitive MSCs in *current* again: $M_3 = M_e\, M_d$, $M_2 = M_f M_d$. Therefore, we use the same solution above, tracing one step only and using the results from the terminated path to calculate first. Finally, we find that M_2 is the right one to follow, and we can recover the basic repetitive MSC M' nested in M_2.

However, we need to consider another important case, that is, if the uncovered basic repetitive MSCs in *msc*(*O*) have common suffix with the already covered basic repetitive MSCs in *current* (uncovered one and covered one are in the same context), and then we may fail to infer those hiding basic repetitive MSCs by using the solution above. For example,

suppose *current*=$M_1\ (M_f\ M_d)^*(M_e\ M_d)^*M_4\ (M_c\ M_a)^*(M_b\ M_a)^*M_7$ is the MSC-graph given in Figure 4.14, and suppose *msc*(*O*)= $M_1M_4M_hM_aM_hM_aM_7$. We trace *msc*(*O*) in *current* backward using the same solution above. First, we find a common part M_7, then remove M_7 from *msc*(*O*). Therefore, *msc*(*O*)= $M_1M_4M_hM_aM_hM_a$, and we reach the node v_4 in *current* at the same time. Then, in *current* we meet the two basic repetitive MSCs M_6=$M_b\ M_a$ and M_5=$M_c\ M_a$ concatenated together, and they have a common suffix M_a. Because M_a is also a suffix of *msc*(*O*), so we finally would follow either M_5 or M_6. In that way, we would never recover the basic repetitive MSC M_hM_a contained in *msc*(*O*). The solution is that we first terminate tracing before we begin to trace the concatenated basic repetitive MSCs in *current*. Let *cutCurrent* just proceeds those concatenated repetitive MSCs in *current*, and *cutO* is the cut where the tracing terminate on *msc*(*O*). Next, check if the portion between *cutO* and the next possible connection point on *msc*(*O*) are made of one or several basic repetitive MSCs. If yes, we need to check if new basic repetitive MSCs are recovered. If no new ones are recovered, we continue to trace those concatenated repetitive MSCs as we explained at the beginning.

Now, we continue our example above. Before we trace into the concatenated basic repetitive MSCs M_6=$M_b\ M_a$ and M_5=$M_c\ M_a$, we terminate the tracing immediately. Let *cutCurrent* be the cut at the end of M_4, which just proceeds M_5 in *current*. Let *cutO* be the cut at the end of M_a, which just proceeds M_7. Further, we find that the next possible connection

cut on *msc*(*O*) is the cut at the end of M_4 in *msc*(*O*). Therefore, the portion between *cutO* and connection point is $M_hM_aM_hM_a$, which is made of basic repetitive MSC M_hM_a.

4.3.2 Alogorithm 3 - Find Maximum Suffix (modified)

In order to waiving the temporary assumption that concatenated basic repetitive MSCs do not have common suffix, the original algorithm 3 must be adapted to handle the special case. Note that, three kinds of tracing results (condition 1,2 and 3) could possibly be returned in the original algorithm. However, one new condition 4 under the special case that concatenated basic repetitive MSCs have common suffix, is added to the possible results set in the adapted algorithm. Since in the adapted algorithm, there are several pairs of *cutO* and *cutCurrent* that need be returned, we use two arrays *cutCurrent*[] and *cutO*[] to store the results. The algorithm is described as below:

If we reach a node without the special case, the tracing process is performed as before. If we reach a node where some of the concatenated basic repetitive MSCs have common suffix, then first we need to compare *msc*(*O*) with the "transition MSC" proceeding those concatenated loops in *current*, if there is a common suffix between *msc*(*O*) and the "transition MSC", then we skip all concatenated basic repetitive MSCs, and directly proceed to the "transition MSC". If not, then we first terminate tracing before we begin to trace into the concatenated basic repetitive MSCs in *current*. Let *cutCurrent*[0] be the cut that just proceeds those concatenated repetitive MSCs in *current*, and *cutO*[0] be the cut where the tracing terminates in *msc*(*O*). Next, we begin to trace into those concatenated basic repetitive

MSCs. If some of those concatenated basic repetitive MSCs have common suffix with *msc*(*O*), we trace *msc*(*O*) in *current* by all possible tracing paths at the same time, but only trace backward one step at a time. Because every basic repetitive MSC is different from each other, at most one tracing path can go though to the next step without termination, and other tracing paths must all terminate after one step tracing. Consequently, all the terminated tracing paths are returned. If there exists a single tracing path (at most one) that has not terminated in the previous step, then it should be recorded in case of all returned *max_common_suffix* do not work. The advantage of this solution is that we can easily process one step at a time, and possibly find the right tracing path without finishing all of the possible ones.

Algorithm 3 Int FindMaximumSuffix(IN MSC-graph *current*, MSC *msc(O)*, OUT cut *cutCurrent*[], cut *cutO*[])

1: *currentEdge*=the edge leading to exit node v_f in *current*
2: M_o:= *msc*(*O*)
3: while *true* do
{Let *currentEdge* be (v_x, M, v_y)}
4: If *we reach a node where some of the concatenated basic repetitive MSCs have common suffix*, then
{compare *msc*(*O*) with the "transition MSC" M proceeding those concatenated loops in *current*,}
5: M_1:=MaxCommonSuffix(M, M_o);
6: if $M_1 \neq null$ then
{we skip all concatenated basic repetitive MSCs, and directly proceed to the "transition MSC"}
7: *currentEdge*:= the next transition edge in *current*
8: else
{we first terminate tracing before we begin to trace into the concatenated basic repetitive MSCs in *current*. }

9: *cutCurrent*[0] = the cut that just proceeds those concatenated repetitive MSCs in *current*
10: *cutO*[0]= the cut where the tracing terminates in *msc*(*O*)
{we continue to trace into those concatenated repetitive MSCs}
11: if *some of those concatenated basic repetitive MSCs have common suffix with msc*(*O*) then
12: we trace *msc*(*O*) in *current* by all possible tracing paths at the same time, but only trace backward one step at a time.
13: *cutO_All*[1..n] = all possible cuts on *msc*(*O*)
14: *cutCurrent_All*[1..n]= all possible cuts on *current*
15: end if
{Condition 5: suppose there are *n* terminated tracing results. All of them are recorded in *cutCurrent_All*[1..n] and *cutO_All*[1..n]. If there exists a path that has not terminated, then it should be recorded}
16: return 4
17: end if
{If we reach a node without the special case, and current edge is not ε edge.}
18: else if $M \neq$ null then
19: M_1:= MaxCommonSuffix(M, M_o);
20: if M_1= null then
21: if $v_x = v_y$ then
{Basic repetitive MSC could be skipped}
22: *currentEdge*:= the next edge in *current*
23: else
{Tracing ends. Condition 3: max_common_suffix on *current* ends at node v_y, and max_common_suffix on *msc*(*O*) are the portion being moved from M_o }
24: *cutCurrent*[0]:= the cut just proceed the max common suffix on *current*
25: *cutO*[0]:= the cut just proceed the max common suffix on *msc*(*O*)
26: return 3
27: end if
28: else if $M_1 = = M$ then
{We have traced the entire M}
29: Move M_1 from M_o
30: if M_o = *null* then
31: if *currentEdge*=the edge beginning from entry node v_0 then
{Tracing ends. Condition 1: max_common_suffix on *current* ends at node v_0, and max_common_suffix on *msc*(*O*) are the entire *msc*(*O*)}
32: *cutCurrent*= ϕ

```
33:                         cutO = φ
34:                         return 1
35:                     else
                            {Tracing ends. Condition 2: max_common_suffix on current
                            does not end at node v0, but max_common_suffix on msc(O) are
                            the entire msc(O)}
36:                         return 2
                        end if
37:                 end if
38:                 If vx ≠ vy then
39:                   If currentEdge= the edge beginning from entry node v0 then
                          {Tracing ends. Condition 2: max_common_suffix on current ends at
                          node v0, but max_common_suffix on msc(O) are not the entire
                          msc(O)}
40:                       cutCurrent= φ
41:                       cutO = φ
42:                       return 2
43:                   end if
44:                       currentEdge= the next edge of current
45:                 end if
46:             else
                    {Tracing ends. Condition 3: max_common_suffix on current ends in the
                    middle of currentEdge, and max_common_suffix on msc(O) are M1 plus
                    the portion being moved from Mo }
47:                  cutCurrent[0]:= the cut just proceed the max common suffix on current
48:                  cutO[0]:= the cut just proceed the max common suffix on msc(O)
49:                  return 3
50:             end if
51:         else
                {skip ε edge}
52:                  currentEdge= the next edge in current
53:         end if
54: end while
```

Figure 4.15. Algorithm 3 – Find Maximum Suffix (modified)

Since algorithm 3 is modified, algorithm 2 (Infer several basic repetitive MSCs at once) where algorithm 3 (*FindMaximumSuffix*()) is called needs to handle the new resulted condition: condition 4. In the original algorithm 2, if the return value of algorithm

FindMaximumSuffix() is 3, then one or several new basic repetitive MSCs might end before *cutO*, and the algorithm 2 continues recovering process. Actually, condition 4 should be handled in the same way as condition 3. However, in the modified algorithm 3, the *cutO*[] and *cutCurrent*[] (the return value of algorithm 3) are two arrays that contain different tracing results (only in condition 4). Consequently, the pairs of *cutO* and *cutCurrent* should be tried in order (the order in the array) until the algorithm successfully recovers one or several basic repetitive MSCs. If all pairs of *cutO* and *cutCurrent* are tried, but no basic repetitive MSCs is found, and at the same time there exists a path that does not terminate in the previous execution of algorithm 3 at the same time, then algorithm 2 should call algorithm 3 to continue tracing that path.

Note that, the complexity of algorithm 2 in the worst case is not changed after waiving the temporary assumption. The reason is that, in the worst case, the number of pairs of *cutO* and *cutCurrent* that need to be handled to recover the basic repetitive MSCs has been maximized when we compute the complexity of algorithm 2. Consequently, under the new condition 4, we just try more pairs of *cutO* and *cutCurrent*, but the number of pairs is still in the range of the maximum one.

4.4 Illustrating the Proposed Solution

Assume that we are observing a system with three process p_1, p_2 and p_3. Suppose we are provided with the following four observations (the total order of events of each process is assumed to be preserved in the provided lists):

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

and

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

and

$O_3 = s.m_{a,2,1}, s.m_{a,2,1}, s.m_{a,2,1}, r.m_{a,2,1}, r.m_{a,2,1}, s.m_{c,1,3}, r.m_{a,2,1}, r.m_{c,1,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{j,2,1}, s.m_{k,2,3}, s.m_{j,2,1}, s.m_{k,2,3}, s.m_{b,2,3}, r.m_{j,2,1}, r.m_{k,2,3}, r.m_{j,2,1}, r.m_{k,2,3}, r.m_{b,2,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{e,3,2}, s.m_{f,3,1}, s.m_{g,2,1}, s.m_{h,3,2}, s.m_{i,3,1}, s.m_{h,3,2}, s.m_{i,3,1}, s.m_{d,2,1}, r.m_{e,3,2}, r.m_{f,3,1}, r.m_{h,3,2}, r.m_{i,3,1}, r.m_{h,3,2}, r.m_{i,3,1}, r.m_{g,2,1}, r.m_{d,2,1}$

and

$O_4 = s.m_{a,2,1}, s.m_{a,2,1}, s.m_{a,2,1}, r.m_{a,2,1}, r.m_{a,2,1}, s.m_{c,1,3}, r.m_{a,2,1}, r.m_{c,1,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{j,2,1}, s.m_{k,2,3}, s.m_{j,2,1}, s.m_{k,2,3}, r.m_{j,2,1}, r.m_{k,2,3}, r.m_{j,2,1}, r.m_{k,2,3}, s.m_{d,2,1}, s.m_{b,2,3}, s.m_{d,2,1}, s.m_{b,2,3}, s.m_{b,2,3}, r.m_{d,2,1}, r.m_{b,2,3}, r.m_{d,2,1}, r.m_{b,2,3}, r.m_{b,2,3}, s.m_{c,1,3}, r.m_{c,1,3}, s.m_{d,2,1}, r.m_{d,2,1}$

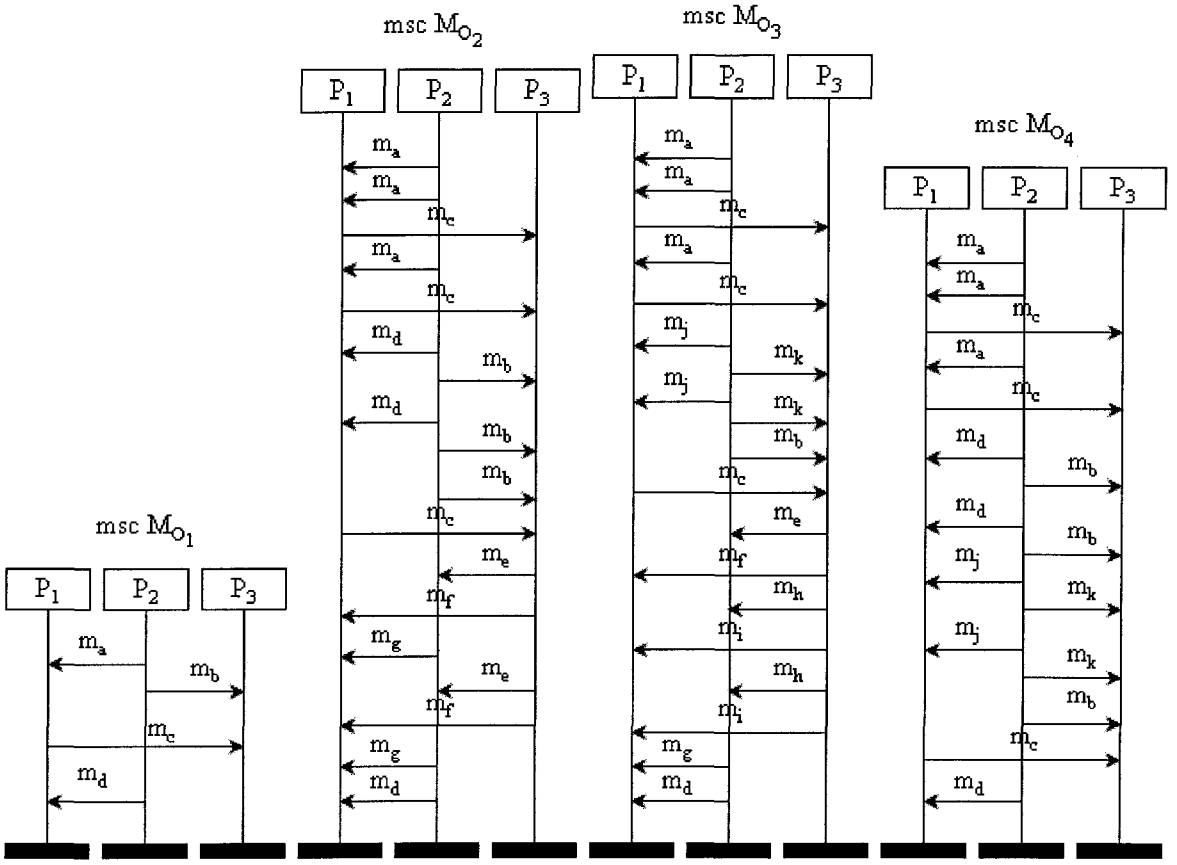

Figure 4.16. MSCs infered by O_1, O_2, O_3 and O_4

These four observations induce the MSCs M_{O1}, M_{O2}, M_{O3} and M_{O4} respectively, as depicted in Figure 4.16. The shortest observation, and thus the initial one, is O_1.

4.4.1 A simple example

First, we begin to process $msc(O_1)$ and $msc(O_2)$. The algorithm *InferRepetitive*($msc(O_1)$, $msc(O_2)$) is thus invoked.

The longest common suffix is the single-message MSC $M_1 = (m_{d,2,1})$. The possible reconnection cut on $msc(O_1)$ is 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 $msc(O_2)$ as Previous($s.m_{e,3,2}$). The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{e,3,2},m_{f,3,1},m_{g,3,1},m_{e,3,2},m_{f,3,1},m_{g,3,1}$, which infers the three-message MSC $M_2 = (m_{e,3,2}, m_{f,3,1}, m_{g,3,1})$ 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 $msc(O_1)$and $msc(O_2)$. This time, the maximum common suffix is the two-message MSC $M_3 = (m_{b,2,3},m_{c,1,3})$, and the possible reconnection cut is simply (s.$m_{a,2,1}$, r. $m_{a,2,1}$). The possible reconnection cut (s.$m_{a,2,1}$, r. $m_{a,2,1}$) is first found on $msc(O_2)$ 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 any basic repetitive MSC, and thus another reconnection cut on $msc(O_2)$ will be searched for. It is found as *Previous*($s.m_{a,2,1}$), and *IsMadeOfBasicRepetitives* 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 both $msc(O_1)$ and $msc(O_2)$, 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 terminates, with an MSC-graph as $M_6.M_5^k.M_4^k.M_3.M_2^k.M_1$.

Figure 4.17 shows the MSC-graph resulted from $msc(O_1)$ and $msc(O_2)$. As expected, both M_{O1} and M_{O2} can be obtained from that graph: M_{O1} comes from $v_0.(M_6).v_1.(\varepsilon).v_2.(M_3).v_3.(M_1).v_f$, and M_{O2} comes from $v_0.(M_6).v_1.(M_5).v_1.(M_5).v_1.(\varepsilon).v_2.(M_4).v_2.(M_4).v_2.(M_3).v_3.(M_2).v_3.(M_2).v_3.(M_1).v_f$.

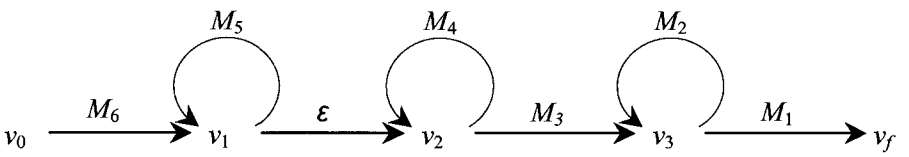

Figure 4.17. The MSC-graph obtained after processing *msc*(O_1) and *msc*(O_2)

4.4.1.1 Adding nested basic repetitive MSCs and ambiguities

Our proposed algorithm is able to recover nested basic repetitive MSCs as well. Now we continue our example to process *msc*(O_3) to illustrate this process. After processing *msc*(O_1) and *msc*(O_2), *current* is a regular expression $M_6.M_5^k.M_4^k.M_3.M_2^k.M_1$. $M_1 = (m_{d,2,1})$, $M_2 = (m_{e,3,2},m_{f,3,1},m_{g,3,1})$, $M_3 = (m_{b,2,3},m_{c,1,3})$, $M_4 = (m_{d,2,1},m_{b,2,3})$, $M_5 = (m_{a,2,1},m_{c,1,3})$, and $M_6 = (m_{a,2,1})$. Next, Algorithm *InferRepetitive*(*current*, *msc*(O_3)) is invoked.

This time, the longest common suffix ends in the middle of basic repetitive MSC $M_2 = (m_{e,3,2},m_{f,3,1},m_{g,3,1})$, so the suffix we get is two MSCs, MSC $M_1 = (m_{d,2,1})$ and $M_7 = (m_{g,3,1})$. The possible reconnection cut on *current* is thus the sending of m_f on p_3, the reception of m_e on p_2 and the reception of m_f on p_1, which can be found in *msc*(O_3) as *Previous*($s.m_{h,3,2}$). The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{h,3,2},m_{i,3,1},m_{h,3,2},m_{i,3,1}$. Furthermore $m_{h,3,2},m_{i,3,1},m_{h,3,2},m_{i,3,1}$, infer the two-message MSC $M_8 = (m_{h,3,2},m_{i,3,1})$ to be basic repetitive which is nested in basic repetitive MSC $M_2 = (m_{e,3,2},m_{f,3,1},m_{g,3,1})$.

A recursive call to *InferRepetitive* is thus made on the MSCs leading up to the occurrence of $m_{f,3,1}$ on both *current* and *msc*(O_3). This time, the maximum common suffix is

two MSCs, $M_9 = (m_{e,3,2},m_{f,3,1})$ and $M_3 = (m_{b,2,3},m_{c,1,3})$. Because this time, the maximum suffix stops on *current* before a basic repetitive $M_4 = (m_{d,2,1},m_{b,2,3})$, there are several possibilities for the reconnection cut. One possibility is the reception of m_b on p_3, the sending of m_b on p_2 and the reception of m_d on p_1, which can not be found in $msc(O_3)$. So we skip the basic repetitive $M_4 = (m_{d,2,1},m_{b,2,3})$. Then the connection cut could be the reception of m_c on p_3, the sending of m_a on p_2 and the sending of m_c on p_1, which can be found in $msc(O_3)$ as $Previous(s.m_{j,2,1})$. The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{j,2,1},m_{k,2,3},m_{j,2,1},m_{k,2,3}$. Furthermore $m_{j,2,1}$ and $m_{k,2,3}$, infer the two-message MSC $M_{10} = (m_{j,2,1},m_{k,2,3})$ to be basic repetitive. Because this time we did not break a transition, ambiguity is generated. We don't know the respective order of the basic repetitive MSC $M_{10} = (m_{j,2,1},m_{k,2,3})$ and $M_4 = (m_{d,2,1},m_{b,2,3})$ until later.

The last recursive call to *InferRepetitive* is thus made on what is left of both *current* and $msc(O_3)$, namely $m_{a,2,1},m_{a,2,1},m_{c,1,3},m_{a,2,1},m_{c,1,3}$, which are immediately recognized as MSC $M_6 = (m_{a,2,1})$ and two occurrences of $M_5 = (m_{a,2,1},m_{c,1,3})$, and the algorithm terminates, with the MSC-graph as $M_6.M_5^k.(M_4^k|M_{10}^k).M_3.(M_9.M_8^k.M_7)^k.M_1$.

4.4.1.2 Waiving ambiguity

In our example, observation O_4 gives a respective order of the basic repetitive MSC $M_{10} = (m_{j,2,1},m_{k,2,3})$ and $M_4 = (m_{d,2,1},m_{b,2,3})$. By calling *InferRepetitive*(*current*, M_{O4}), we can waive the ambiguity and the MSC-graph is obtained as $M_6.M_5^k.M_{10}^k.M_4^k.M_3.(M_9.M_8^k.M_7)^k.M_1$. Figure 4.18 shows the MSC-graph we have got from $msc(O_1)$, $msc(O_2)$, $msc(O_3)$, and $msc(O_4)$.

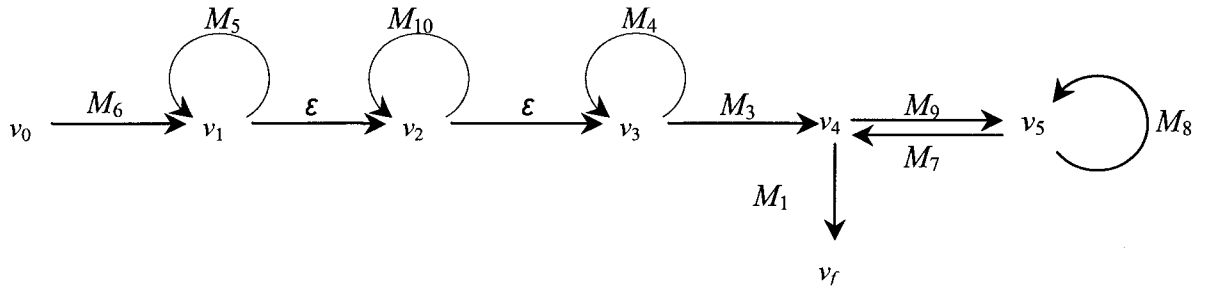

Figure 4.18 The MSC-graph obtained after processing *msc*(O_1), *msc*(O_2), *msc*(O_3), and *msc*(O_4)

4.4.2 The second example

The second example is as follows: assume that we are observing a system with six process p_1, p_2, p_3, p_4, p_5 and p_6. Suppose we are provided with three observations O_1, O_2, O_3. We have input to the software these three observations, which are given in Figure 4.19 and 4.20 in MSC representation. First, we begin to process *msc*(O_1) and *msc*(O_2). The algorithm *InferRepetitive*(*msc*(O_1), *msc*(O_2)) is thus invoked.

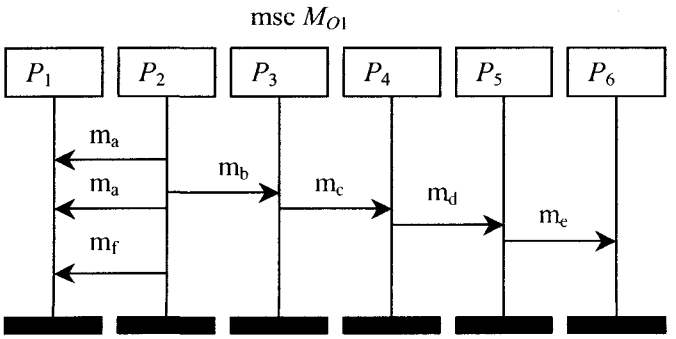

Figure 4.19 MSC infered by O_1

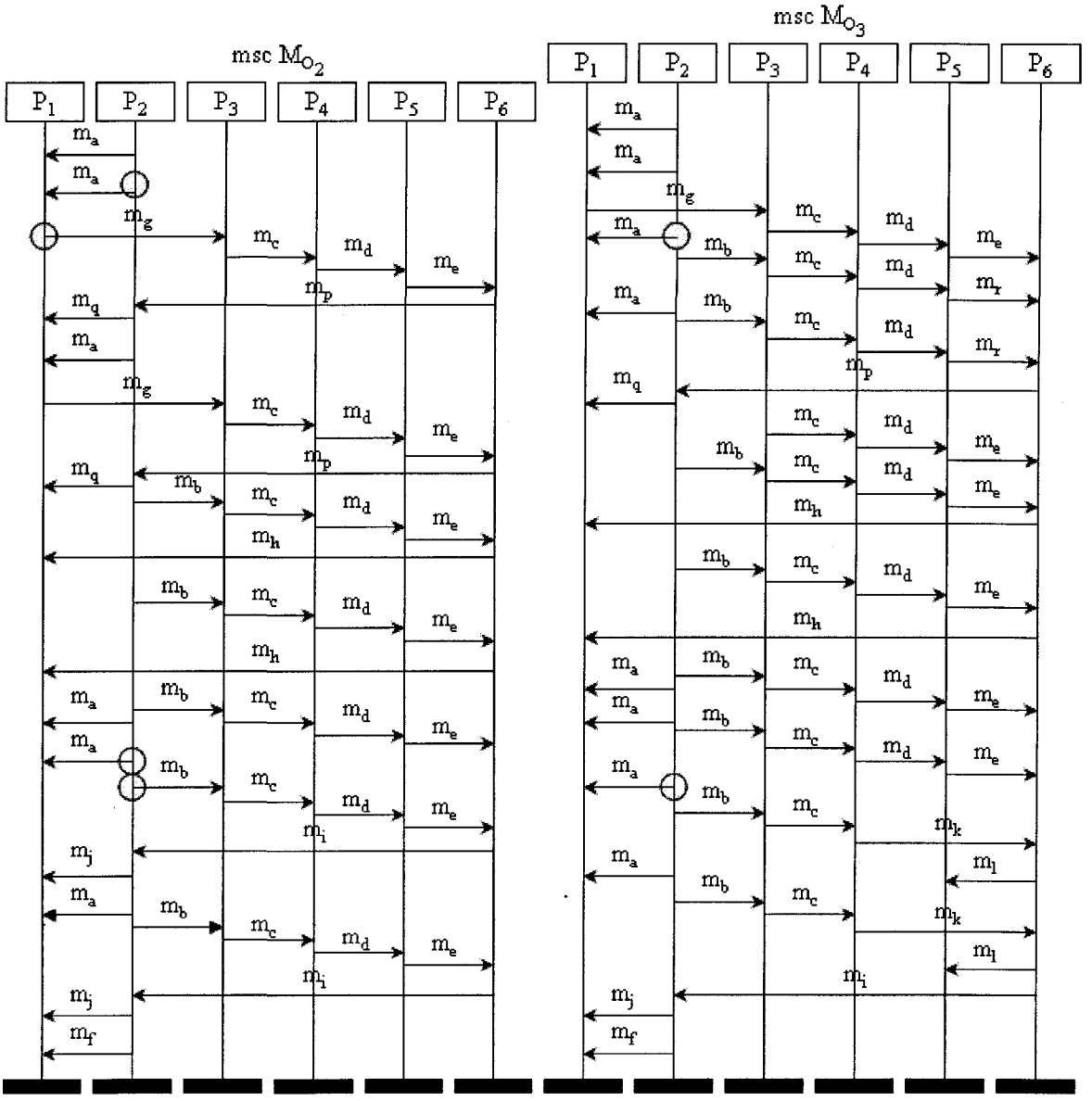

Figure 4.20. MSCs infered by O_2 and O_3

The longest common suffix is the single-message MSC $M_1 = (m_{f,2,1})$. The possible reconnection cut can be found in $msc(O_2)$ as Previous($s.m_{b,3,2}$). To make it clear, $s.m_{b,3,2}$ is circled in Figure 4.20. The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{b,2,3}, m_{c,3,4}, m_{d,4,5}, m_{e,5,6}, m_{i,6,2}, m_{j,2,1}, m_{a,2,1}, m_{b,2,3}, m_{c,3,4}, m_{d,4,5}, m_{e,5,6}, m_{i,6,2}, m_{j,2,1}$. This call will fail

identifying any basic repetitive MSC, and thus another possible connection cut on $MSC(O_2)$ will be searched for. Consequently, the possible connection cut is found as $Previous(s.m_{a,2,1})$, and *IsMadeOfBasicRepetitives* is called on the segment $m_{a,2,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{i,6,2},m_{j,2,1},m_{a,2,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{i,6,2},m_{j,2,1}$, which this time is recognized as the concatenation of two occurrences of the basic repetitive MSC $M_2 = (m_{a,2,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{i,6,2},m_{j,2,1})$.

A recursive call to *InferRepetitive* is thus made on the MSCs left on both $msc(O_1)$and $msc(O_2)$. This time, the maximum common suffix is the five-message MSC M_3 = $(m_{b,2,3},m_{a,2,1},m_{c,3,4},m_{d,4,5},m_{e,5,6})$, and the possible reconnection cut is simply the cut (s.$m_{a,2,1}$, r.$m_{a,2,1}$). The cut (s.$m_{a,2,1}$, r.$m_{a,2,1}$) is first found on $msc(O_2)$ as Previous($s.m_{g,1,3}$), and *IsMadeOfBasicRepetitives* is then called on the segment $m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{p,6,2},m_{q,2,1},m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{p,6,2},m_{q,2,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{h,6,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{h,6,1}$. This call will fail identifying any basic repetitive MSC, and thus another connection cut on $msc(O_2)$ will be searched for. It is found as $Previous(s.m_{a,2,1})$, and *IsMadeOfBasicRepetitives* is called on the segment $m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{p,6,2},m_{q,2,1},m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{p,6,2},m_{q,2,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{h,6,1},m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{h,6,1}$, which this time is recognized as the concatenation of two occurrences of the basic repetitive MSC $M_5 = (m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{p,6,2},m_{q,2,1})$ followed by two occurrences of the basic repetitive MSC $M_4 = (m_{b,2,3},m_{c,3,4},m_{d,4,5},m_{e,5,6},m_{h,6,1})$. A recursive call to *InferRepetitive* is thus made on what is left of both $msc(O_1)$ and $msc(O_2)$, 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 terminates, with an MSC-graph as $M_6.M_5^k.M_4^k.M_3.M_2^k.M_1$.

Now we continue our example to process $msc(O_3)$. After processing $msc(O_1)$ and $msc(O_2)$, *current* is a regular expression $M_6.M_5^k.M_4^k.M_3.M_2^k.M_1$. Next, Algorithm *InferRepetitive*(*current*, $msc(O_3)$) is invoked.

This time, the longest common suffix ends in the middle of basic repetitive MSC M_2 = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$,$m_{i,6,2}$,$m_{j,2,1}$), so the suffix we get is two MSCs, MSC M_1 = ($m_{f,2,1}$) and M_7 = ($m_{i,6,2}$,$m_{j,2,1}$). The possible reconnection cut can be found in $msc(O_3)$ as *Previous*($s.m_{a,2,1}$). The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{k,4,6}$,$m_{l,6,5}$,$m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{k,4,6}$,$m_{l,6,5}$, which infers the five-message MSC M_8 = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{k,4,6}$,$m_{l,6,5}$) to be basic repetitive which is nested in basic repetitive MSC M_2 = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$,$m_{i,6,2}$,$m_{j,2,1}$).

A recursive call to *InferRepetitive* is thus made on the MSCs left on both *current* and $msc(O_3)$. This time, the longest common suffix ends in the middle of basic repetitive MSC M_5 = ($m_{a,2,1}$,$m_{g,1,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$,$m_{p,6,2}$,$m_{q,2,1}$), so the suffix we get is four MSCs, M_{10} = ($m_{p,6,2}$,$m_{q,2,1}$), M_9 = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$), the basic repetitive MSC M_4 = ($m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$,$m_{h,6,1}$) and M_3 = ($m_{b,2,3}$,$m_{a,2,1}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$). The possible reconnection cut can be found in $msc(O_3)$ as *Previous*($s.m_{a,2,1}$). The call to *IsMadeOfBasicRepetitives* is then made on the segment $m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{r,5,6}$, $m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{r,5,6}$, which infers the five-message MSC M_{10} = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{r,5,6}$) to be basic repetitive which is nested in the basic repetitive MSC M_5 = ($m_{a,2,1}$,$m_{b,2,3}$,$m_{c,3,4}$,$m_{d,4,5}$,$m_{e,5,6}$,$m_{i,6,2}$,$m_{j,2,1}$).

The last recursive call to *InferRepetitive* is thus made on what is left of both *current* and *msc*(O_3), namely $m_{a,2,1},m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6}$, which is immediately recognized as MSCs $M_6 = (m_{a,2,1})$, $M_{12} = (m_{a,2,1},m_{g,1,3},m_{c,3,4},m_{d,4,5},m_{e,5,6})$, and the algorithm terminates, with an MSC-graph as $M_6.M_{12}.M_{11}^{k}.M_{10}.M_4^{k}.M_3.M_9.M_8^{k}.M_7.M_1$. We can see the resulting MSC-graph in Figure 4.21.

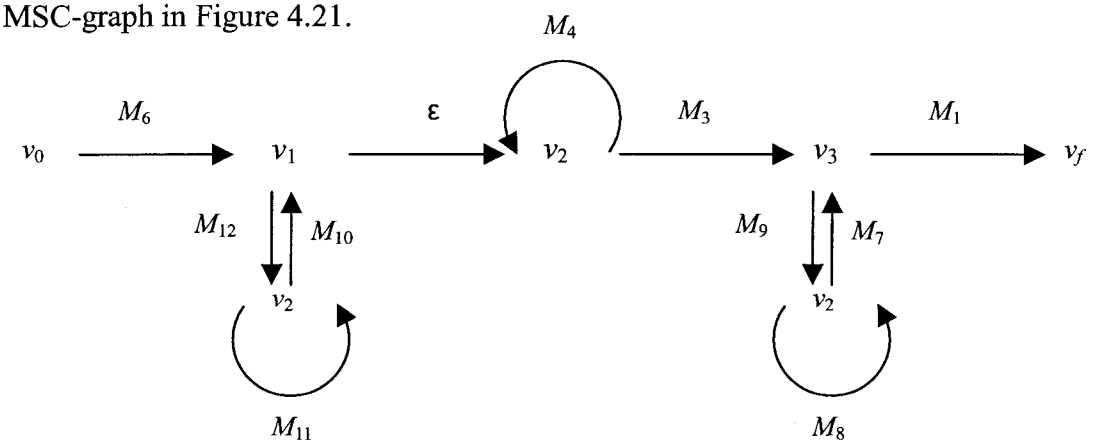

Figure 4.21 The MSC-graph obtained after processing *msc*(O_1), *msc*(O_2), and *msc*(O_3)

4.5 Implementation of the Algorithms

The implementation of the proposed algorithms is based on the implementation of the algorithms in [Ura04] and [Jou05], and the same data structures are used in our implementation. The software has been developed with Object-Oriented Programming using *C++*. The input of the software is a finite set of observations of a concurrent system. Each observation must be represented as a text file, and each file must contain the number of processes in the concurrent system and the sequence of the events. We can see the format of input files and an example of an input file in Appendix *B*.1. The software outputs a stream that describes the resulting MSC-graph, but the data structure can be easily modified to give the output in another structure. We can see an example of the output in Appendix *B*.2.

In the implementation, 15 classes are used to encapsulate the functionality of the implementation. An MSC is represented by an array of events as it shown in Figure 4.22. n is the number of processes, and m is the number of events in $M|_i$. The data structure is represented by the class graph shown in Figure 4.23. Has-A relationship is shown by arrows. Below, we are going to describe each class:

Class Name	Description
Message	Represent an event
Observation	Represent an observation as a vector of messages
ObservationSet	Represent a set of observations as a vector of observations
MSC	Represent an MSC as an array of vectors of messages
MscSet	Represent a set of MSCs as a vector of MSCs,
Node	Represent a node of a graph as a string
Transition	Represent an edge of a graph as two nodes and an MSC
Path	Represent a path as a vector of transitions
Graph	Represent a graph as a vector of paths and a couple of nodes representing the initial and final node of the graph
Couple	Trace two MSCs at the same time
GraphAndCut	The return type of function maxcommonprefix
MsgExtension	Store the results of pre-computation. It includes a message and corresponding set of *Previous*(*e*)
LinearExtension	Represent a set of MsgExtensions which belong to an MSC, as a vector of MsgExtension
LinearExtensionSet	Represent a set of LinearExtensions as a vector of LinearExtensions
myUtility	the implementation of Algorithms 1,2,3

Table 4.2 Classes used in the implementation

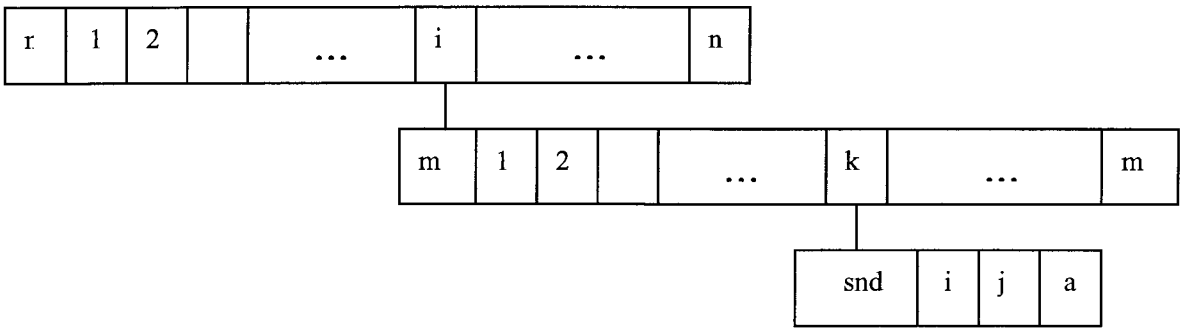

Figure 4.22. Internal structure of an MSC

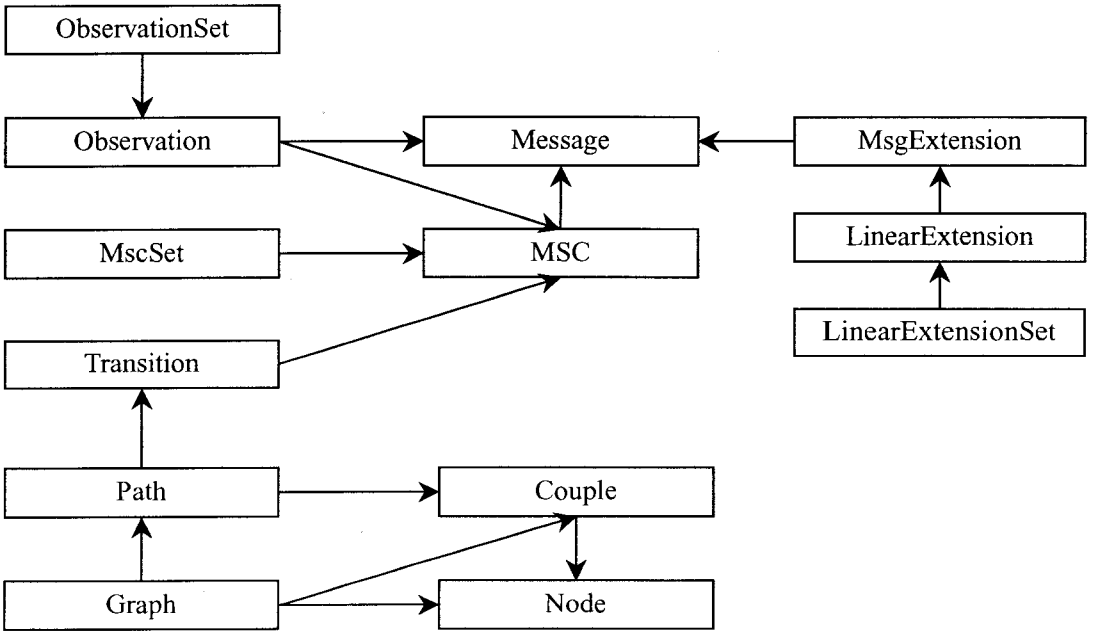

Figure 4.23 Class graph of the implementation

There are about 2000 lines of codes in the implementation. The software can be run under most common operating systems because we only use standard library (the list of the library can be find in Appendix B.4.), but it only was tested on Windows and Unix. The input and output formats are shown in Appendix B. There is an example of the input files listed in Appendix B.1. The output of the implementation based on the two examples given in the last section is also given in Appendix B. The output stream of the implementation for the first example is listed in Appendix B.2, and the output stream of the second one is listed in Appendix B.3.

Chapter Five

Conclusions

5.1 Final Remarks

We have presented an algorithm which, given a set of actual observations of the execution of a individual functionality of a distributed system, constructs under specific assumptions a high-level system design model represented by an MSC-graph where repetitive sub-functions of the functionality are identified. Our algorithm is capable of identifying repetitive patterns and repetitive sub-patterns of an individual functionality of a distributed system. The language of the MSC-graph derived consists of all the MSCs corresponding to the given observations and the inferred observations. In the resulting MSC-graph G, the basic repetitive MSCs and their relative positions are all explored. Therefore, the resulting MSC-graph G is a high-level design model of the existing system. It can be used in the system maintenance or evolutionary development.

5.2 Summary of Contributions

In this thesis, the proposed method waives a strong assumption from previous method [Jou05], so it is much less restrictive and much more practical than the previous one. In our proposed methods, processing one observation could recover a set of repetitive sub-functions instead of at most one repetitive sub-function in the previous method [Jou05]. We believe this to be a significant practical improvement 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 tight together in the design of the system.

A paper associated with this thesis has been published at an international conference, Formal Techniques for Networked and Distributed Systems (FORTE) 2007.

5.3 Directions for Future Research

It would be interesting to see this work improved and/or extended in the following directions:

Actual system states can be set or described by setting conditions in a MSC or MSC-graph [ITU99]; however, our definition of the MSC (Definition 2) and MSC-graph (Definition 9) has not included system states for the purpose of this thesis. However, the definition of MSC-graph (Definition 9) can be improved to interpret global system states associated with repetitive sub-functions. Actually, the finite set of nodes in the definition 9 can be used to represent a set of global system states. That is, the entry node v_0 and the exit node v_f can represent the initial and final system states respectively, and each node associated with a basic repetitive MSC represents the state where a loop starts and ends.

A basic limitation of MSC-graphs is that it can only model finitely generated behaviours, that is, each MSC in the language of an MSC-graph can only be defined as the sequential composition of elements chosen from a fixed finite set of MSCs. However, the behaviours of many protocols in asynchronous distributed systems are non-finitely

generated. This occurs for example with scenarios generated by the sliding window protocols. In the abstract level, sliding window protocols can be seen as a series of query messages from sender to receiver and answer messages from receiver to sender. Sliding window allows a sender to transmit a specified number of data units before an acknowledgement is received or before a specified event occurs. Therefore, in sliding window protocols, the events belonging to a process are partially ordered, and can induce a collection of braids that cannot be finitely generated. Consequently, our proposed method cannot be used on the execution traces of such protocols. However, an extension of MSCs called Causal Message Sequence Charts, which allow the events belonging to a lifeline to be partially ordered, is introduced in [Gaz07]. Causal Message Sequence Charts can be utilized to model scenarios from sliding window protocols [Gaz07]. Therefore, it will be a big improvement to utilize Causal Message Sequence Charts in our resulted high-level design model.

Furthermore, the method proposed in this thesis cannot recover repetitive sub-functions with choice (conditional statements) included. If there are choices inside a repetitive sub-function, then the execution traces of the repetitive sub-function could change in different executions of the implementation. Hence the question is, can an MSC-graph, which is a high-level design model, be generated to interpret choices under such evidences, and how?

Appendix

Appendix A: Algorithms from [Ura04] and [Jou05]

A.1 Checking if M_1 and M_2 infer an MSC M to be repetitive

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 $M_1'' \neq M_{mp}$ 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 $M \neq M_{mp}$ 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

A.2 Building the MSC graph based on a set of MSCs M

1: /* phase 1: infer repetitive subfunctions and form G*/
2: initially all the MSCs in M are unmarked
3: generate the initial and the final nodes v_0 and v_f in G
4: for each pair M_1, $M_2 \in$ 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_0 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 $1 \leq i < k$, $\text{src}(p_i) \cap \text{src}(p_{i+1}) \neq \phi$;

22: for all $P \in \Pi$ do

23: insert $merge(P)$ into G'

24: end for

A.3 Merging paths in a partition *P*

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 $\text{src}(p_m) = \text{src}(p)$

4: let $P = P - \{p\}$

5: while P is not empty do

6: let p be a path in P such that $\text{src}(p) \cap \text{src}(p_m) \neq \phi$ and the prefix label of p is the shortest among other such paths in P.

7: $\text{src}(p_m) = \text{src}(p_m) \cup \text{src}(p)$

8: $P = P - \{p\}$

9: trace the concatenation of 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)

A.4 Finding maximal common prefix of *M'* and *M"*

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 $n_e \rightarrow n_{e'}$ 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 $n_e \rightarrow n_{e'}$ and $n_{e'} \rightarrow n_e$ 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

A.5 Finding basic repetitive MSC *M'* of a given MSC *M*

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 = gcd($\{r_i\}$)
6: if $r \geq 2$ then
7: for all P_i do
8: $M'|_i$ = first | E_i | r events of $M|_i$
9: end for
10: else
11: return empty MSC
12: end if

A.6 construct a lattice

1: *topLabel* = the MSC of the shortest observation
2: S = set of all observations minus *topLabel*

3: while $S \neq \phi$ do

4: /* First phase: discovering the next set of repetitive sub-functions */
5: for all observations $o \in S$ do
6: Let *prefix* = *MaxCommonPrefix*(*topLabel*, *o*)
7: Let *suffix* = *MaxCommonSuffix*(*topLabel*, *o*)
8: if *prefix* ≠ null and *suffix* ≠ null and *prefix.suffix* ∈ *topLabel* then
9: Let M be the portion of the MSC of o between *prefix* and *suffix*
10: /* MSC representation of *o=prefix.M.suffix* */
11: if M_1 = basic_repetitive_MSC(M) then
12: /* $M = M_1^k$ for some k. We have found *prefix.M.suffix* */
13: Add *prefix.M.suffix* as a successor of *topLabel* in the lattice
14: Remove o from S
15: end if
16: end if
17: end for
18: /* second phase: reconstruction phase */
19: Close the lattice by combining all the found successors of *topLabel*.
20: Assign *topLabel* to the top of the lattice.
21: end while

Appendix B

B.1 Example of Input

All of the input files must be exactly as the following:

--The first line must be a number that is the number of the processes in the concurrent system.

--The other lines, describing events, must be in this order:

* 's' or 'r'
    * '\t' (a tabulation character)
    * the sender of the message
    * '\t' (a tabulation character)
    * the receiver of the message
    * '\t' (a tabulation character)
    * the label of the message
    * '\n' (a new line character)

An example of an input file of the software is as the following:

```
3
s   2   1   ma
r   2   1   ma
s   2   3   mb
r   2   3   mb
s   1   3   mc
r   1   3   mc
s   2   1   md
r   2   1   md
```

B.2 The Output of the First Example

The output of the software is a kind of stream that describes the resulting MSC-graph, including the description of paths, nodes and corresponding MSC. And it can be easily adapted to output a graph in a different way. The software prints this kind of stream as output in the above typical example.

```
SUCCESS: the MSC-graph is given as the following:
[=] Graph [=]
        [-] Path 0 [-]
                [.] Transition 0 [.]
                V0 -- MSC 9 --> Node17
                        MSC 9 : 2 events
                        Process 1 :
                                (r,1,2,ma,false)
```

```
            Process 2 :
                        (s,2,1,ma,false)
            Process 3 :
                              no message
[.] Transition 1 [.]
Node17 -- MSC 8 --> Node17
            MSC 8 : 4 events
            Process 1 :
                        (r,1,2,ma,false)
                        (s,1,3,mc,false)
            Process 2 :
                        (s,2,1,ma,false)
            Process 3 :
                        (r,3,1,mc,false)
[.] Transition 2 [.]
Node17 -- Epsilon --> Node48
[.] Transition 3 [.]
Node48 -- MSC 7 --> Node48
            MSC 7 : 4 events
            Process 1 :
                        (r,1,3,mk,false)
            Process 2 :
                        (r,2,3,mj,false)
            Process 3 :
                        (s,3,2,mj,false)
                        (s,3,1,mk,false)
[.] Transition 4 [.]
Node48 -- Epsilon --> Node47
[.] Transition 5 [.]
Node47 -- MSC 6 --> Node47
            MSC 6 : 4 events
            Process 1 :
                        (r,1,2,md,false)
            Process 2 :
                        (s,2,1,md,false)
                        (s,2,3,mb,false)
            Process 3 :
                        (r,3,2,mb,false)
[.] Transition 6 [.]
Node46 -- MSC 5 --> Node27
            MSC 5 : 4 events
```

Process 1 :
(s,1,3,mc,false)
Process 2 :
(s,2,3,mb,false)
Process 3 :
(r,3,2,mb,false)
(r,3,1,mc,false)
[.] Transition 7 [.]
Node27 -- MSC 4 --> Node28
MSC 4 : 4 events
Process 1 :
(r,1,3,mf,false)
Process 2 :
(r,2,3,me,false)
Process 3 :
(s,3,2,me,false)
(s,3,1,mf,false)
[.] Transition 8 [.]
Node28 -- MSC 3 --> Node28
MSC 3 : 4 events
Process 1 :
(r,1,3,mi,false)
Process 2 :
(r,2,3,mh,false)
Process 3 :
(s,3,2,mh,false)
(s,3,1,mi,false)
[.] Transition 9 [.]
Node28 -- MSC 2 --> Node27
MSC 2 : 2 events
Process 1 :
(r,1,2,mg,false)
Process 2 :
(s,2,1,mg,false)
Process 3 :
no message
[.] Transition 10 [.]
Node27 -- MSC 1 --> Vf
MSC 1 : 2 events
Process 1 :
(r,1,2,md,false)

```
                    Process 2 :
                      (s,2,1,md,false)
                Process 3 :
                        no message
[=] End of Graph [=]
```

B.3 The Output of the Second Example

```
SUCCESS:the MSC-graph is given as the following:
[=] Graph [=]
        [-] Path 0 [-]
                [.] Transition 0 [.]
                V0 -- MSC 10 --> Node36
                        MSC 10 : 2 events
                        Process 1 :
                                (r,1,2,ma,false)
                        Process 2 :
                                (s,2,1,ma,false)
                        Process 3 :
                                no message
                        Process 4 :
                                no message
                        Process 5 :
                                no message
                        Process 6 :
                                no message
                [.] Transition 1 [.]
                Node36 -- MSC 9 --> Node37
                        MSC 9 : 10 events
                        Process 1 :
                                (r,1,2,ma,false)
                                (s,1,3,mg,false)
                        Process 2 :
                                (s,2,1,ma,false)
                        Process 3 :
                                (r,3,1,mg,false)
                                (s,3,4,mc,false)
                        Process 4 :
                                (r,4,3,mc,false)
                                (s,4,5,md,false)
                        Process 5 :
                                (r,5,4,md,false)
```

(s,5,6,me,false)
Process 6 :
(r,6,5,me,false)
[.] Transition 2 [.]
Node37 -- MSC 8 --> Node37
MSC 8 : 10 events
Process 1 :
(r,1,2,ma,false)
Process 2 :
(s,2,1,ma,false)
(s,2,3,mb,false)
Process 3 :
(r,3,2,mb,false)
(s,3,4,mc,false)
Process 4 :
(r,4,3,mc,false)
(s,4,5,md,false)
Process 5 :
(r,5,4,md,false)
(s,5,6,mr,false)
Process 6 :
(r,6,5,mr,false)
[.] Transition 3 [.]
Node37 -- MSC 7 --> Node36
MSC 7 : 4 events
Process 1 :
(r,1,2,mq,false)
Process 2 :
(r,2,6,mp,false)
(s,2,1,mq,false)
Process 3 :
no message
Process 4 :
no message
Process 5 :
no message
Process 6 :
(s,6,2,mp,false)
[.] Transition 4 [.]
Node36 -- Epsilon --> Node35
[.] Transition 5 [.]

```
Node35 -- MSC 6 --> Node35
        MSC 6 : 10 events
        Process 1 :
                (r,1,6,mh,false)
        Process 2 :
                (s,2,3,mb,false)
        Process 3 :
                (r,3,2,mb,false)
                (s,3,4,mc,false)
        Process 4 :
                (r,4,3,mc,false)
                (s,4,5,md,false)
        Process 5 :
                (r,5,4,md,false)
                (s,5,6,me,false)
        Process 6 :
                (r,6,5,me,false)
                (s,6,1,mh,false)
[.] Transition 6 [.]
Node35 -- MSC 5 --> Node27
        MSC 5 : 10 events
        Process 1 :
                (r,1,2,ma,false)
        Process 2 :
                (s,2,3,mb,false)
                (s,2,1,ma,false)
        Process 3 :
                (r,3,2,mb,false)
                (s,3,4,mc,false)
        Process 4 :
                (r,4,3,mc,false)
                (s,4,5,md,false)
        Process 5 :
                (r,5,4,md,false)
                (s,5,6,me,false)
        Process 6 :
                (r,6,5,me,false)
[.] Transition 7 [.]
Node27 -- MSC 4 --> Node28
        MSC 4 : 10 events
        Process 1 :
```

(r,1,2,ma,false)
Process 2 :
(s,2,1,ma,false)
(s,2,3,mb,false)
Process 3 :
(r,3,2,mb,false)
(s,3,4,mc,false)
Process 4 :
(r,4,3,mc,false)
(s,4,5,md,false)
Process 5 :
(r,5,4,md,false)
(s,5,6,me,false)
Process 6 :
(r,6,5,me,false)
[.] Transition 8 [.]
Node28 -- MSC 3 --> Node28
MSC 3 : 10 events
Process 1 :
(r,1,2,ma,false)
Process 2 :
(s,2,1,ma,false)
(s,2,3,mb,false)
Process 3 :
(r,3,2,mb,false)
(s,3,4,mc,false)
Process 4 :
(r,4,3,mc,false)
(s,4,6,mk,false)
Process 5 :
(r,5,6,ml,false)
Process 6 :
(r,6,4,mk,false)
(s,6,5,ml,false)
[.] Transition 9 [.]
Node28 -- MSC 2 --> Node27
MSC 2 : 4 events
Process 1 :
(r,1,2,mj,false)
Process 2 :
(r,2,6,mi,false)

(s,2,1,mj,false)
Process 3 :
no message
Process 4 :
no message
Process 5 :
no message
Process 6 :
(s,6,2,mi,false)
[.] Transition 10 [.]
Node27 -- MSC 1 --> Vf
MSC 1 : 2 events
Process 1 :
(r,1,2,mf,false)
Process 2 :
(s,2,1,mf,false)
Process 3 :
no message
Process 4 :
no message
Process 5 :
no message
Process 6 :
no message
[=] End of Graph [=]

B.4 Library References

The list of the library used in the c++ program is given in alphabetic order:

- assert,
- fstream,
- iostream,
- iterator,
- list,
- stdlib,
- stdio
- string,
- time,
- vector.

References

[Alu99] Alur, R., Yannakakis, M.: *Model checking of message sequence charts.* In: 10th International Conference on Concurrency Theory, Springer Verlag (1999) 114–129

[Alu00] Alur, R., Etessami, K., Yannakakis, M.: *Inference of message sequence charts.* In:22nd International Conference on Software Engineering. (2000) 304–313

[Alu03] Alur, R., Etessami, K., Yannakakis, M.: *Inference of message sequence charts.* IEEE Transactions on Software Engineering **29** (2003) 623–633

[Ben97] Ben-Abdallah, H., Leue, S.: *Syntactic detection of progress divergence and non–local choice in message sequence charts.* In: 2nd International Conference Tools and Algorithms for the Construction and Analysis of Systems. (1997) 259–274

[Che02] Chen, X. J., Ural, H.: *Construction of deadlock–free designs of communication protocols from observations.* The Computer Journal 45 (2002) 162–173

[Chi90] Chikofsky, E., Cross, J.: *Reverse engineering and design recovery.* IEEE Software 7 (1990) 13–17

[Gaz07] Thomas Gazagnaire, Blaise Genest, Loic Helouet, P.S. Thiagarajan, and Shaofa Yang: *Causal Message Sequence Charts*. In: Proc. of 18th Int. Conf. on Concurrency Theory (CONCUR), a volume of Lecture Notes in Computer Science. Springer. 2007.

[ISO IS 9074] ISO-Information Processing Systems-Open Systems Interconnection: *Estelle-a formal description technique based on an extended state transition model*, IS 9074

[ITU99] *Z.120. message sequence charts*. In ITU-T Recommandation, 1999.

[Jou05] Jourdan, G.V., Ural, H., Yenigun, H.: *Recovering the lattice of repetitive sub-functions*. In: International Symposium on Computer and Information Sciences 2005, LNCS 3733. (2005) 956–965

[Kos94] Koskimies, K., Makinen, E.: *Automatic synthesis of state machines from trace diagrams*. Software–Practice & Experience 24 (1994) 643–658

[Lee93] Lee, D., Sabnani, K.: *Reverse engineering of communication protocols*. In: IEEE International Conference on Network Protocols (1993) 208–216

[Mus98] A. Muscholl, D. Peled, and Z. Su: *Deciding properties of message sequence charts*. Foundations of Software Science and Computation Structures, 1998.

[Pro91] R.L. Probert, K. Saleh: *Synthesis of communications protocols. Survey and assessment*, IEEE Transactions on Computers 40 (1991) 468–476.

[Raj91] Rajagopal, M., Miller, R. E.: *Synthesizing a protocol converter from executable protocol traces.* IEEE Transactions on Computers 40 (1991) 487–499

[Rud96] Rudolph, E., Graubmann, P., Gabowski, J.: *Tutorial on message sequence charts.* Computer Networks and ISDN Systems–SDL and MSC **28** (1996)

[Rum99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[Sal96] Saleh, K., Boujarwah, A.: *Communications software reverse engineering: A semi–automatic approach.* Information & Software Technology 38 (1996) 379–390

[Sal96b] Saleh, K., Probert, R. L., Manonmani, I.: *Recovery of communication protocol design from protocol execution traces.* In: IEEE International Conference on Engineering of Complex Computer Systems (1996), 265–272

[Sal99] Saleh, K., Probert, R. L., Al-Saqabi, K.: *Recovery of communication protocol design from protocol execution traces.* In: Information and Software Technology, Volume 41, Number 11 (1999), 839-852

[Ura04] Hasan Ural and Hüsün Yenigün. *Towards design recovery from observations.* In: Formal Techniques for Networked and Distributed Systems 2004, LNCS 3235. (2004) 133–149.