MODELING AND VERIFICATION OF MESSAGE SEQUENCE CHARTS USING PROCESS ALGEBRAS AND TEMPORAL LOGIC MODEL CHECKING

by

Wai Han Chiu
B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

M.Sc.

in the Department of Computer Science

We accept this thesis as conforming to the required standard

Dr. B.M. Kapron, Supervisor, Dept. of Computer Science

Dr. M.H.M. Cheng, Outside Member, Dept. of Computer Science

Dr. W. Wadge, Member, Dept. of Computer Science

Dr. I. Traoré, External Examiner, Dept. of Electrical and Computer Engineering

© Wai Han Chiu, 2003

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Supervisor: Dr. B.M. Kapron

ABSTRACT

This thesis presents a technique for verifying assumptions about the ordering of events in a simple message sequence chart (MSC) model. In this model, MSC's come with an interpretation which corresponds to a visual or enforced order. The basic problem of verification for these simple MSC's is checking whether or not a user-inferred ordering is in fact consistent with the underlying visual or enforced order. The technique of this thesis uses the Calculus of Communicating Systems (CCS) process algebra to model MSC's, and a temporal logic called GCTL* (Generalized Computation Tree Logic) to specify user-inferred orderings. We give a general method for translating visual order MSC's into CCS and we prove that this translation is correct. We also show how to translate MSC's with any enforced ordering into CCS and give a partial proof that this translation is correct. We show how to specify inferred orderings in temporal logic, and describe how we can now use a temporal logic model-checking tool such as the Concurrency Workbench to verify inferred orderings in MSC's. Finally we describe a tool that we have developed which translates visual and enforced-ordering MSC's into CCS in a format which is suitable for input to CWB. We give examples of how this tool can be used in combination with CWB to verify orderings in MSC's.

Examiners:

Dr. B.M. Kapron, Supervisor, Dept. of Computer Science

Dr. M.H.M. Cheng, Outside Member, Dept. of Computer Science

Dr. W. Wadge, Member, Dept. of Computer Science

Dr. I. Traoré, External Examiner, Dept. of Electrical and Computer Engineering

Table of Contents

Abstract ii

Table of Contents iv

List of Figures vii

List of Tables viii

Notation ix

Acknowledgement xi

Dedication xii

1 Introduction 1

1.1 Overview 4

1.2 Notes 5

2 Message sequence charts 7

2.1 Visual order message sequence charts 7

2.2 Interpreted message sequence charts 8

2.3 Examples 9

2.3.1 A simple visual order MSC 10

2.3.2 A MSC with multiple interpretations 10

2.4 Notes 13

3 Calculus of communicating systems 16

3.1 Agent expressions 16

3.2 Transition semantics of CCS 17

3.3 Traces and labelled transition systems 18

3.4 Examples 19
3.4.1 The clock example 19
3.4.2 The vending machine example 19
3.4.3 Using the restriction operator 20
3.5 Notes 21

4 Modeling message sequence charts using CCS **24**
4.1 Translating visual order MSC's 24
4.2 Translating interpreted MSC's 27
4.3 Examples 29
4.3.1 Two visual order translations 29
4.3.2 An enforced order example 30
4.4 Notes 32

5 Verifying assumed orderings using temporal logic **33**
5.1 Temporal logics 33
5.2 Syntax of GCTL* 34
5.3 Semantics of GCTL* formulas 35
5.4 Representing inferred orderings in GCTL* 36
5.5 Examples 37
5.6 Notes 37

6 Conclusion **39**

Bibliography **41**

Appendix A Modelling MSC into CCS Translation Tool User Manual **44**
A.1 Introduction 44
A.2 Installation 44
A.3 User Interface 45
A.3.1 User Input Menu 45
A.3.2 Translation Output 47
A.3.3 Pull-down Menu 49
A.4 Conclusion and Future Work 50

A.5 Notation 50
A.6 Error Message Description 50

Appendix B Program Code **53**

Appendix C Using the tool output and CWB-NC to verify inferred orderings **71**

Index **83**

List of Figures

Figure 1.1 Verifying assumed order in MSC using model checking 4

Figure 2.1 A simple MSC 7
Figure 2.2 A simple MSC with more than one interpretation 10
Figure 2.3 Inferred and enforced ordering examples 12

Figure 4.1 Enforced order translation example 31

Figure C.1 Example 1 71
Figure C.2 Example 2 76
Figure C.3 Example 3 79

List of Tables

Table 2.1 Buffering policies 13

Table A.1 Notation Table 50

Notation

List of Notations

M	Message Sequence Chart
P	A finite set of local processes
E	A finite set of events
L	Label function
c	Compatibility action
R	Received event
S	Send event
e,f,g	Event
$<$	Visual order
$<_p$	Total order
$<^*$	Transitive closure
$\ll$	Enforced order
$\sqsubset$	Inferred order
$\longrightarrow$	Transition
α,β	Action
$\overline{\alpha}$	Complementary action
$+$	Alternation
$\mid$	Composition
$\backslash$	Restriction
A	Agent
$[\,]$	Renaming
$\alpha.\mathcal{A}$	Prefix
$!$	Repeated parallel composition
τ	Internal or silent action
t	Complete observable trace of M

δ	Deadlock
Θ	An arbitrary atomic action proposition
$\mathcal{S}$	State formula
$\mathcal{P}$,φ	Path formula
X	If..., otherwise
U	Until
R	Released
F	Eventually operator
$\neg$	Negation
G	Always operator
tt,**ff**	Atomic state proposition

Acknowledgement

I would like to sincerely thank Dr. Bruce Kapron for his immeasurable help and support. Also, during my graduate studies, I must also thank all my friends and co-workers for unlimited encouragement and support. Finally, I must thank my husband for his understanding and support.

Dedication

"For the LORD gives wisdom, and from his mouth come knowledge and understanding."

Chapter 1

Introduction

As software systems grow more complex, the need to guarantee reliability using automated verification techniques is becoming more important. One area in which automated verification is particularly significant is in reactive and concurrent systems.

Reactive systems are characterized by being primarily involved in interaction with other systems or an external environment. An important subcategory of such systems are *concurrent systems* which are comprised of independent components which may interact with each other as well as with an external environment. These sort of systems usually have a very complex control structure, so that traditional validation techniques such as code inspection or testing are less applicable. On the other hand, many of these systems can be modeled in some way using finite-state systems, so that automated verification becomes a feasible alternative.

Examples of reactive systems include operating systems, network protocols and air traffic control systems. All of these systems display a high degree of complexity, and require a rigorous approach to design and implementation to ensure correct behaviour and adequate performance. A rigorous systematic approach to the design of reactive systems would include the phases:

- Specification: from an informal description of requirements for the systems behaviour and performance we create a formal specification, written in a suitable formal language.
- Modeling: using the specification, we get a formal model in a process language formalism
- Verification: check the formal model against the formal specification to ensure

all requirements are met. If requirements are not met, modeling phase is repeated.

- Implementation: the verified model is implemented in hardware/software.
- Testing: test suites are derived from the model and are used to test the actual behaviour and performance of the implemented system

Really, this is an idealized description of the design process, which we probably cannot always do in practice (an exception might be in the development of prototype systems.) However, each phase described here can have a valuable role to play in the development of real systems. In this thesis, our concern will be only with the first two phases described here.

In recent years, there have been many formal systems proposed for modeling and verifying systems, along with tools which do many things, including automated verification of models against a formal specification. It is often the case that these systems appear to be quite different in any number of ways, for example, modeling may be either textually or visually based, while specifications could be based on temporal logics, or on simpler descriptions of traces or orderings. These differences can make it difficult to see that the underlying semantics given to systems are really the same, or that techniques and algorithms can be applied in different cases. Because of this, the same problem may be solved over and over again, and new tools for a problem are developed when tools which are able to solve the problem already exist.

In this thesis, we give a simple example of this situation, and show how we can use existing tools from one formal system to solve a verification problem for another system. More specifically, we will show how we can use the modeling language CCS and a tool, called the Concurrency Workbench, for verifying temporal properties of CCS models, to solve the problem of verifying whether assumed orderings in a MSC are consistent with the ordering given by the underlying semantics of the system.

Message sequence charts (MSC's) are a common visual formalism for modeling concurrent systems. One drawback of MSC's is that a given MSC may be open to a number of (possibly conflicting) interpretations. A user may make assumptions about the ordering of events in the execution of an MSC, even though such an order might not be a consequence of the *visual* ordering of events in the MSC as it appears. Other conflicts can happen because certain orderings may be *enforced* by properties

of the underlying architecture of the system but might not appear in the visual representation given by an MSC. In such a situation it is even more likely that a user will *infer* an ordering on events which does not follow from the underlying properties of the system. So a simple but important verification issue for MSC's is the problem of verifying whether an ordering between events which is inferred by a user must necessary hold in every execution of a given MSC.

In this thesis we present a solution to this problem which is based on a well-established approach to modeling and verifying concurrent systems. We will model MSC's using process algebra (CCS) and verify them using model checking of a specification in a temporal logic (GCTL) against the CCS specification. With this approach, we can use an existing model checking tool, the Concurrency Workbench, to solve the verification problem described above.

The main contributions in this thesis are:

- We present methods for modeling MSC's with visual and enforced orderings into CCS
- We prove a theorem which shows the correctness of our modeling technique for the visual ordering
- We present a method for translating user-assumed orderings into temporal logic specifications in GCTL
- We build a tool which translates MSC's into CCS terms, in a format suitable for model checking by a standard tool (CWB)

Our approach is summarized in Figure 1.1

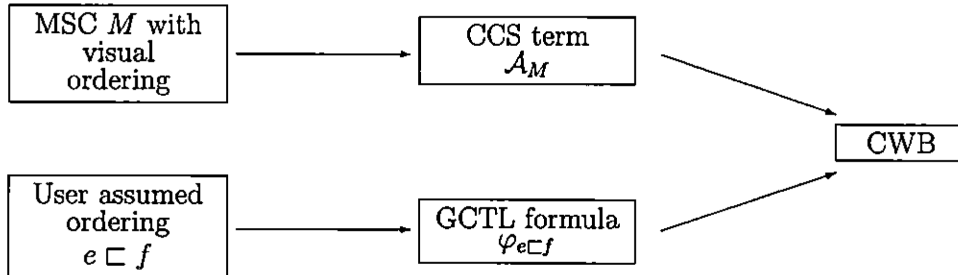


Figure 1.1. *Verifying assumed order in MSC using model checking*

1.1 Overview

This thesis is structured as follows:

Chapter 1 Introduction

Chapter 2 Review of basic message sequence charts

Chapter 3 Review of CCS

Chapter 4 A translation of basic message sequence charts with visual ordering into CCS

Chapter 5 Using temporal logics to verify properties of translated systems

Chapter 6 Conclusions and further work

Appendix A system for transforming MSC's into CCS format suitable for input to a model-checking tool

Most of the original work in the thesis appears in Chapters 4 and 5 and in the Appendix. Chapters 2 and 3 review the modeling systems that we will be dealing with, while the logics we will be using are discussed in Chapter 5. Most of the chapters in this thesis will be organized as: first of all, we explain the basic issues that we are dealing with, and give all the relevant definitions and results. After this, we will have a section of examples which clarify the material presented in the first section. Finally, we will have a section of notes where we provide references and give background and comparison to other work (where it is needed).

1.2 Notes

The approach to system design we describe here has now become widespread in the design of both hardware and software systems, to the extent that there are now standard textbooks [Pel01], [CGP99] which give undergraduate-level presentations of this methodology. Conferences such as CAV (Computer Aided Verification) [BCF01] devoted to automated verification in system development attract hundreds of academic and industrial researchers every year.

Today, there are many systems and tools for doing this sort of modeling and verification. Many variations exist because of the many possible combinations of modeling and specification formalisms. Some of the more popular tools for verifying temporal properties of concurrent systems are SPIN [Hol97], SMV [CGP99], Murφ [DDHY92], and the Concurrency Workbench [CPS93]. A common characteristic of all these systems is that they provide formal languages for modeling systems, as well a formal language for specifying properties of systems. For example, SPIN provides a simple programming language called PROMELA for modeling finite-state systems, and uses Linear-time Temporal Logic (LTL) for specifications. As another example, the Concurrency Workbench uses the process language CCS for modeling and the modal μ-calculus for specification.

MSC's are a natural language for modeling communication protocols and other finite-state systems. So it makes sense to think about having a tool which would allow us to model systems using MSC's and then check whether these models satisfy a specification. But we need to make two decisions:

- What specification language should we use?
- Should we build a verification system from scratch, or base it on one of the existing systems?

We have made these choices about what to do. First of all, we decided that it made more sense from a practical point of view to use an existing tool. Once we chose the tool, our choice of specification language was limited to the ones supported by the tool. The tool we have chosen is a version of the Concurrency Workbench, and the specification language is the temporal logic language GCTL*.

While other researchers have considered the problem of verifying MSC specifications, it seems that no one has specifically addressed the issue of using an existing

verification system to do this, and no one has tried to do this with a very abstract modeling language like CCS.

One of the first papers to consider MSC verification is [AHP96], which considers the problem of automatically checking for race conditions in MSC specifications. It does not address the issue of checking general specifications, but does consider one of the most important verification problems for MSC's. A tool for doing this kind of verification is developed from scratch. This paper was followed by [AY99], which considered more general model checking problems for MSC's. Their approach involves specifications given by an automaton. The MSC is also translated into an automaton and verification is done purely with respect to these automata. They consider more complicated MSC models, such as MSC-graphs and hierarchical MSC-graphs (HMSC's). The method that is described in their work could be used to implement MSC verification using a tool such as SPIN. However, the authors are more concerned with the theoretical aspects of the verification problem. They show that the problem of model checking of MSC's can be done by building an automaton for the linearization of the partial order defined by the MSC. They also show that the problem is NP-complete, and that verification problems for MSC-graphs and HMSC's are even harder to solve. [AEY01] considers the related problem of *realizability* for MSC-graphs, which means deciding if a MSC-graph has a distributed implementation. *Safe* realizability means that the implementation is deadlock-free. Without this restriction we have *weak* realizability. Deciding realizability for MSC graphs also turns out to be computationally very hard.

The method for verifying properties of MSC's that we are using does not make the basic problem any easier and because of the NP-completeness we know that there will be MSC's that will be hard to verify using any method. In our method we do not really worry about the computational complexity of doing the verification because we leave the actual verification up to some other tool. This means that any improvement in the tool will make our verification method better. It might be possible to look for special cases of MSC's that are easier to verify, but we do not investigate this issue in this thesis.

Chapter 2

Message sequence charts

2.1 Visual order message sequence charts

Message sequence charts provide a visual formalism for representing communicating systems. A sample MSC is given in Figure 2.1. Each vertical line corresponds to a

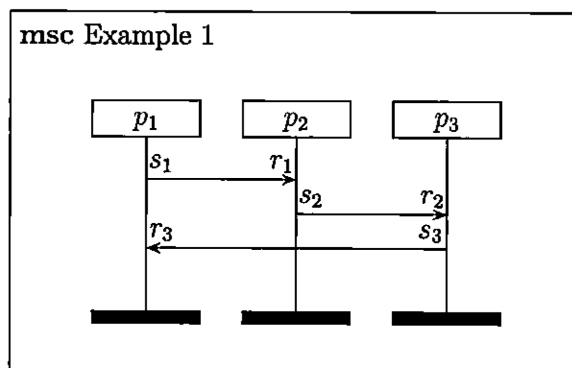


Figure 2.1. *A simple MSC*

process. Messages sent between processes are represented by arrows. The tail of the message is at the sending process and the head is at the receiving process. Message transmission is asynchronous, so associated with each message there are two events: a send event and a receive event. Arrows may be drawn horizontally or sloping downwards, but never sloping upwards. In the example there are three processes and three message transmissions. In the simplest interpretation of message sequence charts, also known as the *visual ordering*, events in each process occur in exactly the order that they are drawn, from top to bottom.

A formal definition of MSC is given as:

Definition 2.1 A *(visual order) message sequence chart (MSC) M* is given by a 5-tuple $\langle P, E, L, c, \{<_p \mid p \in P\}\rangle$ where

- P is a finite set of *(local) processes*
- E is a finite set of *events.* $E = S \cup R$ where S and R are disjoint. S is the set of *send events* and R is the set of *receive events.*
- $L : E \rightarrow P$ is a *labelling function* which maps each event e to a corresponding process $L(e) \in P$. For a given process p, E_p denotes the set $\{e \mid L(e) = p\}$. If $e \in E_p$, we say that e is *in local process p.*
- $c : S \rightarrow R$ is a *compatibility bijection* which uniquely associates every send event with a corresponding receive event. Visually, c corresponds to the *send-receive edges* of the MSC.
- For every process p there is a local total order $<_p$ on the events in E_p, which corresponds to the order in which events are visually displayed.

  We define the visual order on E as follows:

$$< \; = \; (\bigcup_{p \in P} <_p) \cup \{(s, c(s)) \mid s \in S\}$$

We will denote by $<^*$ the transitive closure of $<$[1]

2.2 Interpreted message sequence charts

The visual order, because it is visual, sometimes makes restrictions on the order of events in a system that are stronger than we need. This is shown by the fact that $<_p$ has to be a total ordering. Since the diagram representation of a MSC is two-dimensional, if e and f are in the same local process p then one of them has to come first on the line for this process, so either $e <_p f$ or $f <_p e$. In reality, it might be true that there really isn't a set order between e and f.

[1]This notation is really not standard because $<^*$ is *not* the reflexive transitive closure of $<$ (it is not true that $e <^* e$). But this is the standard notation and we will use it too.

We can deal with this drawback of visual-order MSC's by relaxing the condition that $<_p$ has to be a total order. This gives us the definition of an *interpreted message sequence chart.*

Definition 2.2 An *interpreted message sequence chart* is a 6-tuple $M = \langle P, E, L, c, \{<_p \mid p \in P\}, \{\ll_p \mid p \in P\}\rangle$ where P, E, L, c and $<$ are defined the same as in a visual-order MSC, and for each process p there is a local ordering $\ll_p$ which is consistent with $<_p$, and the *enforced ordering* $\ll$ is defined by

$$\ll \; = \; (\bigcup_{p \in P} \ll_p) \cup \{s, c(s) \mid s \in S\}$$

Again, we will write $\ll^*$ for the transitive closure of $\ll$ (not reflexive). The idea behind $\ll$ is that $e \ll_p f$ if we actually *know* that e always precedes f. For example, this might be enforced by the underlying architecture of the system we are modeling.

So in the most general case we have for each MSC M both a visual ordering $<$ and an enforced ordering $\ll$. However, the enforced ordering is always a subordering of the visual ordering. We will now introduce one more ordering, $\sqsubset$, which is the *inferred* or *assumed* ordering. Events are ordered by $\sqsubset$ according to how the user will expect them to occur. We could view an ordering $e \sqsubset f$ as part of the *specification* of the behaviour of a system, and so we are very interested in being able to tell whether this specified behaviour is actually reflected in the behaviour of the system. If assumed events can happen in the opposite order, we have a race condition which leads to a number of different execution errors. So a basic problem for message sequence charts is determining whether an assumed ordering $e \sqsubset f$ is consistent with $<^*$ or $\ll^*$. This verification problem and a way to solve the problem using an existing verification tool is the main topic of this thesis.

2.3 Examples

We now present examples of MSC's which demonstrate some of their basic properties and also problems that can result from the different possible ways to order events. The examples are based on examples from [AHP96].

2.3.1 A simple visual order MSC

A simple MSC is given in Figure 2.1. Here there are three processes labelled as P_1,P_2 and P_3, and some sending/receiving events labelled s_1, s_2, s_3 and r_1, r_2, r_3. Vertical lines represent the lifetime of a process and the horizontal arrows represent interactions between processes. For any process P, there is a natural ordering of the events belonging to P which is determined by the order in which they appear on the vertical line corresponding to P from top to bottom. This is the visual ordering $<$. For example, P_1 has $s_1 < r_3$, P_2 has $r_1 < s_2$ and P_3 has $r_2 < s_3$.

2.3.2 A MSC with multiple interpretations

It can be possible that the actual ordering of events in a system, based on possible hardware restrictions or buffering policies, does not correspond to the visual order. The visual ordering can be too restrictive because it requires that all events belonging to a process are linearly ordered by how they are drawn. This is demonstrated by the MSC in Figure2.2.

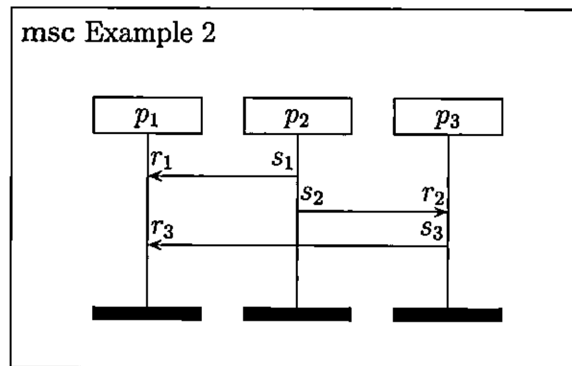


Figure 2.2. *A simple MSC with more than one interpretation*

We always make the reasonable assumption that a message must be sent before it is received. Also, although it is not necessary, we can easily verify whether the underlying hardware enforces the restriction that processes are blocked while waiting to receive a message. In the rest of this chapter, we will assume that processes are blocked while they are waiting for input.

For Figure 2.1, based on these assumptions we get $s_1 << r_1 << s_2 << r_2 << s_3 <$ $< r_3$.

For Figure 2.2, the situation is not so clear. If we make the same assumptions, we cannot necessarily conclude for example, that $s_1 << s_2$. A problem which is not as obvious occurs in P_1: according to the visual ordering, $r_1 < r_3$. However, unless we make some assumption about delivery times, we cannot necessarily conclude that $r_1 << r_3$. It might be possible that after s_1 occurs but before r_1 occurs, all of s_2, r_2, s_3 and r_3 occur. Because of the different orderings we could get even with the basic assumptions we made, we call the assumption that r_1 occurs before r_3 an *assumed* or *inferred ordering*, and write $r_1 \sqsubset r_3$. So with assumed orderings, a basic problem is verifying that the assumed ordering $\sqsubset$ for a MSC is consistent with its enforced ordering $<<$. As we said above, we always want that $<<$ consistent with $<$. And we always also want $\sqsubset$ to be consistent with $<$.

This example taken from [AHP96] shows how different buffering policies can change the interpretation of an MSC. Figure2.3 presents a number of different visual orderings on send and receive events.

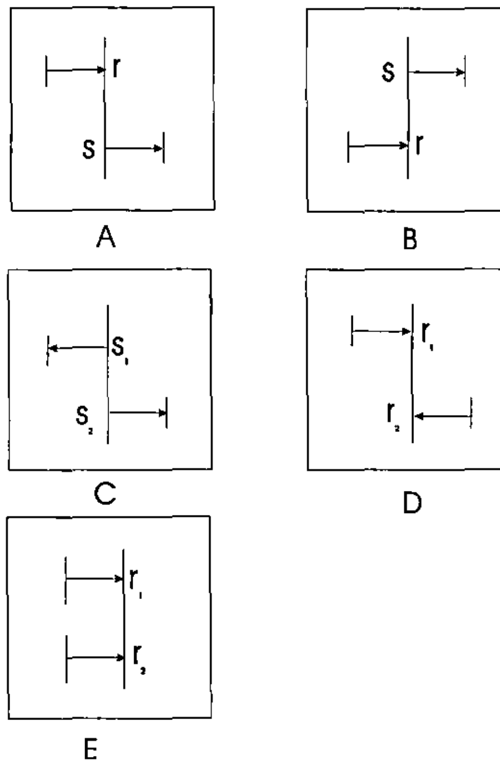


Figure 2.3. *Inferred and enforced ordering examples*

Consider the buffering policies: Single FIFO, in which each process has a single FIFO queue for all messages it receives, FIFO/Source in which each process has a separate FIFO queue for each process which sends it messages, Single non-FIFO in which each process has a single queue for receiving messages but without any guarantee on the ordering of messages and non-FIFO/Source in which each process has a separate queue for each sending process with no assumptions about ordering.

We make the same hardware assumptions described above. Table 2.1 summarizes the differences between these buffering policies with respect to the event sequences given in Figure 2.3. From the restriction about blocking on reads, A is the same

	A	B	C	D	E
Single FIFO	$r \ll s$	$s \sqsubset r$	$s_1 \ll s_2$	$r_1 \sqsubset r_2$	$r_1 \ll r_2$
FIFO/Source	$r \ll s$	$s \sqsubset r$	$s_1 \ll s_2$	none	$r_1 \ll r_2$
Single Non-FIFO	$r \ll s$	$s \sqsubset r$	$s_1 \ll s_2$	$r_1 \sqsubset r_2$	$r_1 \sqsubset r_2$
Non-FIFO/Source	$r \ll s$	$s \sqsubset r$	$s_1 \ll s_2$	none	$r_1 \sqsubset r_2$

Table 2.1. *Buffering policies*

in every case. C is the same in every case from the fact that locally a process has complete control over when it sends a message. C is the same in every case because the user will likely assume that since s may trigger r, it should occur before r. For D, the user may assume that message delivery is timely, and so with a single queue the user infers that r_1 occurs before r_2. For E, the FIFO queue enforces an order on messages received from the same source, so we have $r_1 \ll r_2$ for both. With a non-FIFO queue this is not enforced, but the user may assume timely delivery and the user concludes $r_1 \sqsubset r_2$.

2.4 Notes

Message sequence charts have been used for a long time as an informal notation for specifying or modeling the behaviour of communicating systems. However, since there was no standard or rigorous semantics for this notation, it was quite possible for specifications or models which used MSC's to be ambiguous.

A syntactic specification of message sequence charts is given in the ITU document Z.120 [ITU93]. A formal semantics using the process algebra ACP is given for the ITU specification in [MR94]. This semantics is only for what we are calling the *visual ordering*. There is no consideration of interpreted MSC's as defined in [AHP96]. Also, the primary focus of the paper is just to give a formal semantics for MSC's. They do not consider applications such as using this semantics to do verification with an existing tool. They consider MSC's with more features than basic MSC's such as timers, process creation and co-regions, and conditions and sub-charts for modularity.

MSC's were used only as an informal, illustrative language before the approval of the first MSC recommendation Z.120 in 1992, after which MSC has advanced to a formal and descriptive language. The publication of the first MSC recommendation in 1992 increased the use and popularity of the MSC language beyond the expectation of most people. The main reason for this success was that for the first time systematic specification and verification based on MSC's became possible due to its standardization. But the language constructs defined in MSC'92 were not enough to describe many features of information systems, so MSC's were considered useful only in combination with other languages, predominantly SDL and TTCN [PUW+99]. The development of a formal semantics based on process algebra ([MR94]) led to developments involving composition mechanisms. MSC'96 which is presented in [RGG], was a powerful synthesis of concepts taken from process algebra, Petri nets and beyond that, from object oriented modeling. The object-oriented community has shown increasing interest in the MSC standard as a means for the formalization for Use Cases, and are included as a part of UML [Qua01]. MSC'96 includes many features that go beyond the basic message sequence charts that we use in this thesis. But it should be possible to extend the techniques from this thesis to handle more complicated MSC models. This is left for future work.

The definition of basic message sequence charts that we give in this thesis comes from [AHP96]. This paper introduces *interpreted message sequence charts* and gives a rigorous mathematical definition of the semantics of basic message sequence charts under various orderings (visual, enforced and inferred or assumed). This paper shows that it is possible to give a simple mathematical semantics for MSC's based on partial orderings. Once this simple semantics is provided, it becomes easier to do the sort of

translation that we present in our thesis. The paper also defines race conditions in terms of an inconsistency between the inferred order and enforced order, and considers the problem of checking a given MSC for race conditions. The paper also considers timed MSC's and briefly describes a verification tool for checking race conditions. The problem of general model checking of MSC's is not considered in this paper.

Chapter 3

Calculus of communicating systems

CCS is a language for modeling concurrent systems based on a model of *synchronous hand-shake* communication between concurrently executing *agents*. Agents may perform *actions* which can be viewed as potential interactions with other agents. Agents in CCS are built up using a few simple syntactic operators. We will use the term *agent expression* to indicate this. The basic meaning of an agent expression is given by a *labelled transition relation* between agents. We write

$$\mathcal{A} \xrightarrow{\alpha} \mathcal{B}$$

to mean that the agent $\mathcal{A}$ may perform an action α and thereby become the agent $\mathcal{B}$.

3.1 Agent expressions

Definition 3.1 We have an infinite set $\{A, B, C, \dots\}$ of *agent identifiers*, and an infinite set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \dots\}$ of *labels*. For every label a, there are two corresponding bijection actions a and $\overline{\mathtt{a}}$. We will denote actions by $\alpha, \beta, \gamma, \dots$. There is also a special *silent action* τ. If α is an action, $\overline{\alpha}$ is the *complementary action.* Complementary actions are used to represent communication between processes. Note that if $\alpha = \mathtt{a}$ then $\overline{\alpha} = \overline{\mathtt{a}}$, and if $\alpha = \overline{\mathtt{a}}$, then $\overline{\alpha} = \mathtt{a}$.

The collection of *agent expressions* is defined as:

- Every agent identifier is an agent expression
- If $\mathcal{A}$ is an agent expression and α is an action, then $\alpha.\mathcal{A}$ is an agent expression (*prefix*)
- If $\mathcal{A}$ and $\mathcal{B}$ are agent expressions then so is $\mathcal{A} + \mathcal{B}$ (*alternation*)

- If $\mathcal{A}$ and $\mathcal{B}$ are agent expressions then so is $\mathcal{A}|\mathcal{B}$ (*composition*)
- If $\mathcal{A}$ is an agent expression and L is a set of labels, then $\mathcal{A}\backslash L$ is an agent expression (*restriction*)
- If $\mathcal{A}$ is an agent expression and f is a finite mapping from labels to labels, then $\mathcal{A}[f]$ is an agent expression (*renaming*)

3.2 Transition semantics of CCS

To understand the meaning of these agent expressions, we must see how they are evaluated. This is given by the *transition rules*. In general, we will write $\mathcal{A} \xrightarrow{\alpha} \mathcal{B}$ if agent $\mathcal{A}$ can perform action α to become agent $\mathcal{B}$.

The definition of $\xrightarrow{\alpha}$ depends on the structure of $\mathcal{A}$. We will present this informally.

An agent of the form $\alpha.\mathcal{A}$ can perform an α action and become agent $\mathcal{A}$. So $\alpha.\mathcal{A} \xrightarrow{\alpha} \mathcal{A}$.

The $+$ operator represents nondeterministic choice. So if either $\mathcal{A} \xrightarrow{\alpha} \mathcal{C}$ or $\mathcal{B} \xrightarrow{\alpha} \mathcal{C}$ then $\mathcal{A} + \mathcal{B} \xrightarrow{\alpha} \mathcal{C}$.

The $|$ operator represents parallel composition. In an agent $\mathcal{A}|\mathcal{B}$, both $\mathcal{A}$ and $\mathcal{B}$ are proceeding independently in parallel, but they may interact by a synchronous handshake communication. In terms of the transition relation, this means several things. First of all, if $\mathcal{A} \xrightarrow{\alpha} \mathcal{A}'$ then $\mathcal{A}|\mathcal{B} \xrightarrow{\alpha} \mathcal{A}'|\mathcal{B}$. Also, if $\mathcal{B} \xrightarrow{\alpha} \mathcal{B}'$ then $\mathcal{A}|\mathcal{B} \xrightarrow{\alpha} \mathcal{A}|\mathcal{B}'$. These two rules capture the fact that $\mathcal{A}$ and $\mathcal{B}$ are proceeding independently in parallel. Finally, if $\mathcal{A} \xrightarrow{\alpha} \mathcal{A}'$ and $\mathcal{B} \xrightarrow{\overline{\alpha}} \mathcal{B}'$, then $\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'$. This rule is the most important rule of CCS. The idea is that $\mathcal{A}$ and $\mathcal{B}$ proceed synchronously through a handshake on the complementary actions α and $\overline{\alpha}$. This can also be thought of as a communication from one agent to another. So the complementary actions of the individual agents are replaced by the silent action τ for the whole system.

The $\backslash$ operator is used to make certain actions *internal* to a system. Note that because of the last rule we gave for $|$, whenever an agent $\mathcal{A}$ is able to make an α action, if it is running in parallel with an agent $\mathcal{B}$ which can make an $\overline{\alpha}$ action, then there is a potential communication or handshake between $\mathcal{A}$ and $\mathcal{B}$. By using $\backslash$, $\mathcal{A}$ would be able to hide α from $\mathcal{B}$ if does not want to allow such a communication. In

terms of $\xrightarrow{\alpha}$, we have that if $\mathcal{A} \xrightarrow{\alpha} \mathcal{B}$, then $\mathcal{A}\backslash L \xrightarrow{\alpha} \mathcal{B}\backslash L$, as long as α is not in $L \cup \overline{L}$, where $\overline{L} = \{\bar{\mathrm{a}} \ : \ \mathrm{a} \text{ is in } L\}$.

A key goal in the design of CCS is that it should allow us to build up complex systems from smaller components. Once we have designed a component, we might want to use it in a number of different settings. The *renaming* is very useful in allowing re-use of system components. A renaming function f is just a mapping from labels to labels. Since we are only ever concerned with renaming finitely many labels, we often write such an f as $[\mathrm{b}_1/\mathrm{a}_1, \mathrm{b}_2/\mathrm{a}_2, \ldots, \mathrm{b}_k/\mathrm{a}_k]$, meaning that a_1 gets renamed as b_1, etc. Such a mapping can be extended to a mapping from actions to actions as: $f(\bar{\mathrm{a}}) = \overline{f(\mathrm{a})}$ and $f(\tau) = \tau$. The idea is that $\mathcal{A}[f]$ is like $\mathcal{A}$ except that some of the actions occurring in $\mathcal{A}$ are renamed using f. So we have that if $\mathcal{A} \xrightarrow{\beta} \mathcal{B}$, and $\alpha = f(\beta)$, then $\mathcal{A}[f] \xrightarrow{\alpha} \mathcal{B}[f]$.

3.3 Traces and labelled transition systems

It is natural to extend the idea of a transition $\mathcal{A} \xrightarrow{\alpha} \mathcal{B}$ by a single action α to a transition $\mathcal{A} \overset{t}{\leadsto} \mathcal{B}$ by a *trace* of actions $t = \alpha_1\alpha_2\ldots\alpha_k$. In this thesis, we will mostly be concerned with properties of systems related to their traces, so this will be an important notion for us. If use the symbol ϵ to represent the empty trace, then we have

- For any $\mathcal{A}$, $\mathcal{A} \overset{\epsilon}{\leadsto} \mathcal{A}$
- If $\mathcal{A} \xrightarrow{\alpha} \mathcal{A}'$ and $\mathcal{A}' \xrightarrow{t} \mathcal{B}$, then $\mathcal{A} \overset{t}{\leadsto} \mathcal{B}$.

We will look at the behaviour of a system in terms of its traces. In fact, we will really be concerned with *observable* behaviour of systems, and so we do not want to use silent actions to distinguish different behaviours. This gives us the idea of *observable traces*. We will write $\mathcal{A} \overset{t}{\Longrightarrow} \mathcal{B}$ if $\mathcal{A}$ can become $\mathcal{B}$ after executing the observable trace t. If t is any trace, will write $\mathcal{O}(t)$ for t with all τ's removed. We then have that if $\mathcal{A} \overset{t}{\leadsto} \mathcal{B}$, then $\mathcal{A} \overset{\mathcal{O}(t)}{\Longrightarrow} \mathcal{B}$.

A more general way of looking at the behaviour of an agent is through the use of a *labelled transition system (LTS)*. An LTS is a graph whose nodes are labelled by agents and whose edges are labelled by actions. For an agent $\mathcal{A}$, the set of nodes would be $\{\mathcal{B} \ : \ \mathcal{A} \overset{t}{\leadsto} \mathcal{B} \text{ for some trace } t\}$. There will be an edge labelled α from

a node labelled $\mathcal{B}$ to a node labelled $\mathcal{B}'$ just when there is a transition $\mathcal{B} \xrightarrow{\alpha} \mathcal{B}'$. Generally speaking, for our purposes LTS's will have more generality than we need. However, we will need to use them when we introduce some of the temporal logics in Chapter 5. Note that a labelled transition system could also be based on just the *observable* transitions. In this case the state space would consist of agents reachable by some observable trace, and there will be an edge labelled α from $\mathcal{A}$ to $\mathcal{B}$ if $\mathcal{A} \stackrel{\alpha}{\Longrightarrow} \mathcal{B}$.

3.4 Examples

We will give examples to illustrate the basic constructions for agents.

3.4.1 The clock example

We begin with the prefix operator. The simplest system which performs some action is: `tick`.0. This system can perform one `tick` option, and then terminates.

Another very simple example (originally by Hoare, but given in CCS in [Sti01]) of how the prefix operator is used is in the definition of a clock which ticks forever:

$$Cl = \texttt{tick}.Cl$$

. By the transition rule for prefix we have $\texttt{tick}.Cl \xrightarrow{\texttt{tick}} Cl$. So then by the definition of Cl we have $Cl \xrightarrow{\texttt{tick}} Cl$.

Of course, it is possible to have more than a single prefix, so we could define a more interesting clock as:

$$Cl_2 = \texttt{tick.tock}.Cl_2.$$

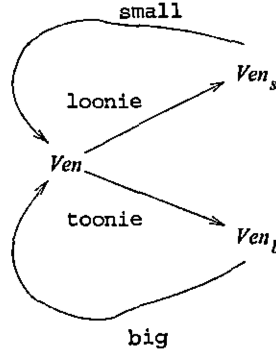
In this case we have $Cl_2 \stackrel{\texttt{tick}\ \texttt{tock}}{\Longrightarrow} Cl_2$.

3.4.2 The vending machine example

The next example illustrates the selection operator $+$:

$$\begin{aligned} Ven &= \texttt{loonie}.Ven_s + \texttt{toonie}.Ven_b \\ Ven_s &= \texttt{small}.Ven \\ Ven_b &= \texttt{big}.Ven \end{aligned}$$

This represents a vending machine which may either accept a loonie and produce a small item, or a toonie and produce a big item. It has the transition system:



The composition operator | allows us to compose processes in parallel, for example:

$$Cl_3 = \texttt{tick}.0 \mid \texttt{tock}.Cl_3$$

This system can perform any sequence of `tick`'s and `tock`'s.

Once we have parallel composition it is natural to consider communication between processes. As described above, synchronous communication is modeled via complementary actions. Consider the system:

$$Cl_4 = \overline{\texttt{tick}}.0 \mid \texttt{tick}.\overline{\texttt{tock}}.Cl_4$$

This system may perform a $\overline{\texttt{tick}}$ or a `tick` action, or the two composed processes may perform a synchronous communication, in which the complementary `tick` options are consumed and the entire system proceeds by a completed *internal* action. Then all of the following transitions are possible:

$$\begin{aligned}
Cl_4 &\xrightarrow{\overline{\texttt{tick}}} \texttt{tick}.\overline{\texttt{tock}}.Cl_4 \\
Cl_4 &\xrightarrow{\texttt{tick}} \overline{\texttt{tick}}.0 \mid \overline{\texttt{tock}}.Cl_4 \\
Cl_4 &\xrightarrow{\tau} \overline{\texttt{tock}}.Cl_4
\end{aligned}$$

Here τ denotes a completed internal (or *silent*) action. We are also identifying $0 \mid P$ with P.

3.4.3 Using the restriction operator

The restriction operator can be used to force certain actions to be used only for internal communications. For example, $Cl_4 \backslash \{\texttt{tick}\}$ can only perform the sequence

of actions:

$$Cl_4\backslash\{\mathtt{tick}\} \xrightarrow{\tau} \overline{\mathtt{tock}}.\mathtt{Cl}_4\backslash\{\mathtt{tick}\} \xrightarrow{\overline{tock}} Cl_4\backslash\{\mathtt{tick}\}$$

The relabelling operator is used to change the name of actions in a process. Usually this is used when putting together two or more systems, in order to get matching names for complementary input and output actions. For example, suppose we have three processes $Q_1 = r_1.\overline{s}_1.P_1$, $Q_2 = r_2.\overline{s}_2.P_2$ and $Q_3 = r_3.\overline{s}_3.P_3$ and we want to make a pipeline with a single input and a single output by putting them together in a sequence. We would get the following term:

$$((Q_1|Q_2|Q_3)[r_2/s_1, r_3/s_2])\backslash\{s_1, s_2\}$$

3.5 Notes

The term *process algebra* is used to refer to algebraic systems which are used to model concurrency. Concurrent processes are represented by terms in an algebra. The algebra has operators for building up complex terms from more simple ones. Usually, there will be some kind of equivalence between terms which respects the operators of the algebra. Today, there are very many process algebras which are used to model different interesting features of concurrent systems, such as timing, randomness, mobility and the use of computational and communication resources, but most of these systems come from one of the three original systems developed in this area. These systems are Hoare's CSP [Hoa85], Milner's CCS [Mil80],[Mil83],[Mil89] and Bergstra and Klop's ACP (see [BK84],[BK85]). Another very important algebraic system is LOTOS [fS86], which is a standardized formal description technique developed by the International Organization for Standards (ISO). LOTOS incorporates features of both CCS and CSP, but also provides features such as formal description of structured data which more theoretical systems like CCS do not have.

Before looking at the algebraic systems, we will note that there are other, non-algebraic approaches to modeling concurrent systems. The most popular of these is Petri nets [Rei85], which give a graphical representation for parallel processes. Other non-algebraic approaches include topological process theory [dBZ] which uses ideas from denotational semantics, and trace theory [AR88].

Algebraic systems share many similarities, but they are not identical. The first distinction that we can make is based on models. Both CCS and CSP start with the idea of a single mathematical model for concurrency, and then look for algebraic laws that hold in these models. ACP starts with a collection of laws and then tries to find models that satisfy these laws. Originally, CSP was based on trace semantics, where models are traces of system executions. A process is fully determined by the possible sequence of atomic actions that it can perform. This actually is quite close to how we think about models for message sequence charts. Later on CSP was modified to use a semantics based on *failure sets*. This allows more general executions where an external environment can prevent some actions so that a deadlock occurs. A process is determined by its failure set, that is the set of all pairs $\langle \sigma, X \rangle$ where σ is a (finite) trace up to failure and X is the allowable actions at the time of failure. As we have already seen, models for CCS are labelled transition systems. Processes are determined by their labelled transition systems. But there is an equivalence between processes called *bisimulation*. Bisimulation is stronger than trace equivalence, for example the CCS terms $a.(b.0 + c.0)$ and $a.b.0 + a.c.0$ are trace equivalent but not bisimilar. They are not bisimilar because in the second process, once an a action happens the choice of the next action is already forced (the nondeterministic choice of b or c happens before the a action). In the first term the choice of b or c can be made after the a action. Because CCS is based on LTS and bisimulation it is more general than CSP. ACP is inspired by CCS, but it is even more general. One important difference is that in ACP there is general sequential composition of processes (so there is $P.Q$ where P is not restricted to be an atomic action). In CCS only parallel composition of general processes is allowed.

An important development of CCS is the π-calculus of Milner [MPW92]. To understand π-calculus we can first think of CCS with value-passing. So now an atomic actions have the form $\overline{a}\langle T \rangle$ and $a(x)$. The first kind of action means send data item T on channel a, the second kind means receive a data item on channel a and bind it to the variable x. Once we have value-passing, we can go one step further and actually allow channel names as data items. So we can receive that name of a channel as input, and then go on to communicate on that channel. This is the important aspect of π-calculus.

The systems that we are looking at here are all *synchronous*, which means that all communications are blocking. But it is not hard to deal with asynchronous systems. If we know that only a bounded number of communications will take place, then we can use a bounded buffer like we have done later in this thesis. Or we could model the asynchronous writing of a data item T on channel a using parallel composition, e.g. $P|\overline{a}\langle T\rangle.0$ models a system which does not block on output of T on channel a.

Chapter 4

Modeling message sequence charts using CCS

Before we can model MSC using CCS, we have to deal with the different model of communication of these two approaches. In MSC's, send and receive events are basic atomic actions which occur asynchronously. The only restriction is that a send event s must occur before the corresponding receive event $c(s)$. In CCS, on the other hand, the basic unit of communication is a synchronous handshake between a send action $\overline{\alpha}$ and a corresponding receive action α. Fortunately, it is not hard to simulate the asynchronous communication of MSC's in CCS. This is partly because communication in the MSC model is very simple, where each event can be seen as the sending or receiving of a single atomic message. We can build a very simple buffering or blocking scheme in CCS to handle this kind of asynchronous communication.

4.1 Translating visual order MSC's

We are going to define a general mapping which will take any MSC M with a visual ordering into a CCS term $\mathcal{A}_M$. Later on we will show how to do this for any interpreted MSC M. The important fact about the translation is given in Theorem 4.5, which says that $e <^* f$ in the visual order of M is equivalent to e appearing before f in every trace of $\mathcal{A}_M$. This means that we can answer questions about assumed ordering relations $e \sqsubset f$ by asking the right kind of question about $\mathcal{A}_M$. We will see in Chapter 5 that there is a way of asking this kind of question in temporal logic.

Before defining the mapping of a MSC M into a CCS term $\mathcal{A}_M$, we will define the set of labels used by $\mathcal{A}_M$.

Definition 4.1 Suppose M is a MSC with event set $E = S \cup R$. Suppose that $S = \{s_1, \ldots, s_k\}$ and $R = \{r_1, \ldots, r_k\}$. Then labels(M) $= S \cup R \cup B \cup C$ where $B = \{b_1, \ldots, b_k\}$ and $C = \{c_1, \ldots, c_k\}$ is the *label set for M*. We will call $B \cup C$ the set of *blocking labels*. For any event $e \in E$, we define action(e), the *action corresponding to event e* as:

$$\text{action}(e) = \begin{cases} s_i.\overline{b_i} & \text{if } e = s_i \in S \\ c_i.r_i & \text{if } e = r_i \in R \end{cases}$$

We also define k *buffer processes* $\mathcal{B}_1, \ldots, \mathcal{B}_k$ where $\mathcal{B}_i = b_i.\overline{c_i}.0$.

The blocking labels b_i and c_i will be used to enforce the ordering between a send action s_i and its corresponding receive action r_i. Blocking labels are an artificial device needed to carry out the translation from M to $\mathcal{A}_M$, and we will end up using the restriction operator so that all of $B \cup C$ is hidden from external observation. The basic idea is that in $\mathcal{A}_M$, an event e from M gets mapped to an action action(e). Suppose e is s_i. Then after s_i happens, r_i cannot happen until the handshake of $\overline{b_i}$ and b_i, followed by the handshake of $\overline{c_i}$ and c_i. So r_i can't happen until s_i happens (c_i *blocks* r_i), but there is no need for synchronization between s_i and r_i. Furthermore, because $\overline{b_i}$ is consumed by the buffer process $\mathcal{B}_i$, events which come after r_i in the same local process are not blocked. This reflects the asynchronous communication model of MSC's.

We are now ready to give the general translation of any visual order MSC M into a CCS term $\mathcal{A}_M$.

Definition 4.2 Suppose that M is a MSC with a given visual ordering on events in $E = S \cup R$. Suppose that $P = \{p_1, \ldots, p_n\}$ is the set of local processes, and that the events in p_i are $e^i_1 < e^i_2 < \cdots < e^i_{k_i}$. We then define the CCS term $\mathcal{A}_i$ to be

$$\text{action}(e^i_1).\text{action}(e^i_2).\cdots.\text{action}(e^i_{k_i}).0$$

Then $\mathcal{A}_M$, the *CCS translation of M*, it the term

$$(!_{1 \leq i \leq n}\mathcal{A}_i \mid !_{1 \leq i \leq n}\mathcal{B}_i) \backslash (B \cup C)$$

where ! is used to denote repeated parallel composition.

Before we prove the main result about our translation, we need a few more definitions.

Definition 4.3 If $\mathcal{A}$ is a CCS term, we say that t is a *completed observable trace* of $\mathcal{A}$ if there is some term $\mathcal{B}$ for which $\mathcal{A} \stackrel{t}{\Rightarrow} \mathcal{B}$, and there is no term $\mathcal{C}$ and $t' \neq \epsilon$ for which $\mathcal{B} \stackrel{t'}{\Rightarrow} \mathcal{C}$. We then the define the set

$$\text{traces}(A) = \{t \mid t \text{ is a completed observable trace of } \mathcal{A}\}$$

If $t = e_1 e_2 \ldots e_k$ is a trace, and e and f are labels, we say the *e precedes f in t* if $e = e_i$, $f = f_j$ and $i < j$.

Definition 4.4 Let f be any event. rank(f) is defined by:

$$\text{rank}(f) = \begin{cases} 0 & \text{if there is no } e \text{ with } e < f \\ \max\{\text{rank}(e) \mid e < f\} + 1 & \text{otherwise} \end{cases}$$

We can now prove the main result about our translation.

Theorem 4.5 (Correctness of the Translation) *For any MSC M with a visual ordering on events, if $\mathcal{A}_M$ is the CCS term defined from M, then for any events e, f in E, then $e <^* f$ iff e precedes f in every $t \in$ traces$(\mathcal{A}_M)$.*

Proof: ($\Rightarrow$) It suffices to show that if $e < f$ then e precedes f in every completed trace of $\mathcal{A}_M$. This is because $e <^* f$ iff for some k, $e <^k f$, where $e <^k f$ is defined by:

$$\begin{aligned} e <^1 f \quad & \Leftrightarrow e < f \\ e <^{k+1} f \quad & \Leftrightarrow \text{For some } e',\ e <^k e' < f \end{aligned}$$

so we get the result by induction. Suppose that $e < f$. There will be two cases.
CASE I: e and f are in the same local process p_i. Then e precedes f in the prefix of actions of $\mathcal{A}_i$, and so e must precede f in every completed trace of $\mathcal{A}_M$.
CASE II: e and f are in different local processes. Then it must be the case that $f = c(e)$, say $e = s_i$ and $f = r_i$. Then the blocking action b_i in $\mathcal{A}_M$ ensures that e comes before f in every completed trace of $\mathcal{A}_M$.
($\Leftarrow$) Suppose that e precedes f in every completed trace of $\mathcal{A}_M$. Again we will consider two cases.

CASE I: e and f are in the same local process p_i. It must be the case that e precedes f in the prefix of actions of $\mathcal{A}_i$. By construction this means that $e < f$.
CASE II: e and f are in different local processes, say p_i and p_j. We will use induction on f. If rank(f) is 0, then there will a trace in which f is the first event, contradicting our assumption. Now suppose rank(f) > 0. We can assume without loss of generality that f is the first event in p_j that follows e in every completed trace of A_M (since if there is some f' in p_j that follows e in every completed trace, we could first show that $e <^* f'$, and then get $e < f$ from the fact that $f' < f$). Then f must be a receive event. Otherwise, it would not be blocked and would precede e in at least one completed trace of $\mathcal{A}_M$. Let $e' = c^{-1}(f)$. Then e must precede e' in every completed trace of $\mathcal{A}_M$: if there is a trace in which e' precedes e there is also a trace in which e' precedes e and e' *immediately precedes* f. But then we have a trace in which f precedes e. Since $e' = c^{-1}(f)$ it must be the case that rank(e') < rank(f). By induction we have $e <^* e'$, and so $e <^* f$. ∎

4.2 Translating interpreted MSC's

We will see in the next chapter how this theorem will help with our basic problem of verifying user-inferred orderings. Before doing that, we will consider how to extend the translation that we have given to arbitrary interpreted MSC's (with any enforced order <<). We are not going to prove a correctness theorem for this translation, but we believe that it would be possible to modify the proof we have given for this case as well. The main difference when we move to an arbitrary enforced order is how we will deal with the translation of local processes. In the visual order, events in a local process are ordered linearly, and this makes the translation easy. With an arbitrary enforced ordering, there could be a number of consistent linear orderings for the events in a local process. One way to deal with this would just be to use the nondeterministic selection operator + of CCS to select between one of these possible linear orderings. However, this approach could lead to an unacceptably large CCS term – in the case where there is no enforced ordering, a local process with k events would require a term with $k!$ different alternatives. Fortunately, the parallel composition operator of CCS allows a better solution. In the extreme case mentioned

above, there is a simple CCS term $!_{1\leq i\leq k} e_k.0$ which works. Unfortunately things get more complicated for arbitrary enforced orderings. We will consider a simple example which demonstrates where the difficulty arises. Suppose we have three events e, f, g with the enforced ordering: $e \ll f$ and $e \ll g$. There is a simple way to represent this in CCS, namely $e.(f.0|g.0)$, which forces e to come first, followed by f and g in either order. Suppose instead that we have $f \ll e$ and $g \ll e$. Now we are stuck, because CCS only allows a sequence of atomic actions as a prefix for any term (we can't have something like $(f|g).e.0)$. The solution is to let $e.0$ run in parallel with $(f.0|g.0)$ but with some kind of blocking mechanism to prevent e from occurring until both f and g have occurred. So we could have $(b.b.e.0|f.\overline{b}.0|g.\overline{b}.0)\backslash\{b\}$. It is not hard to see that the only observable traces that can arise from this term are fge and gfe, and that these are the only sequences of events that are consistent with the given enforced ordering. It isn't hard to see how to generalize this construction. Suppose that we have a local process p with events $e_1, \ldots, e_k$. We will have a corresponding set of *local blocking labels* $d_1, \ldots, d_k$. We use the idea of an event which *immediately precedes* another event: *e immediately precedes f* if $e < f$ and there is no event g such that $e < g$ and $g < f$. The translation of p will have the form

$$(!_{1\leq i\leq k}\mathcal{E}_i)\backslash\{d_1, \ldots, d_k\}$$

where each $\mathcal{E}_i$ has the form

$$d_i.d_i.\cdots.d_i.\text{action}(e_i).\overline{d_{j_1}}.\cdots.\overline{d_{j_l}}.0$$

The number of d_i's in the prefix is exactly the number of events in p which immediately precede e_i. The d_{j_i}'s in the prefix which come after action(e_i) for every event e_{j_i} in p for which e_i immediately precedes e_{j_i}.

Informally it is not hard to see that this translation will work: before e_i can happen, all the events in p which precede e_i must happen first, but they can happen in any order. This is because as each one of these events happen, they will offer a $\overline{d_i}$ action to $\mathcal{E}_i$. Once e_i happens, any action which e_i immediately precedes will no longer be blocked by e_i. We can give a rough upper bound on how large the CCS term would be in terms of the total number of events in the MSC. For a given event in a local process, the worst that could possibly happen would be a prefix which contains a d or $\overline{d}$ for every other event in the same local process. So if there are n

events in a local process, the size of the corresponding term would be $O(n^2)$. Now suppose that there are k local processes $p_1, \ldots, p_k$ and that p_i has n_i events, and that $n = n_1 + n_2 + \cdots + n_k$ is the total number of events. Then the size of the corresponding term would be $O(n_1^2 + \cdots + n_k^2) = O(n^2)$. It is also not hard to see that the construction of the CCS term from the MSC can be done in time $O(n^2)$ where n is the total number of events in the MSC.

We are not going to give a full proof that this translation works, but we give an idea of how the proof would go. First we can use the *expansion law* for CCS to show that the term for each local process is equivalent to a term which corresponds to the + of all terms corresponding to a possible linear ordering of the events in local process. For example, by the expansion law we would have that

$$(s_1.\overline{d}_3.0|s_2.\overline{d}_3.0|d_3.d_3.r_3.0)\backslash\{d_3\}$$

is equivalent to

$$s_1.s_2.r_3.0 + s_2.s_1.r_3.0$$

Once we have expanded each local process, we can see that any trace of the overall system will correspond to a trace of a system we get by first selecting one alternative for each local process and then composing all of them in parallel. On the other hand, to get a consistent linear ordering on events for the whole MSC, we would first choose a consistent linear ordering for each local process, and then consider the linear orderings for the entire system which are consistent with the choice. Once the choices are made, the proof is just the same as it was for the visual order.

4.3 Examples

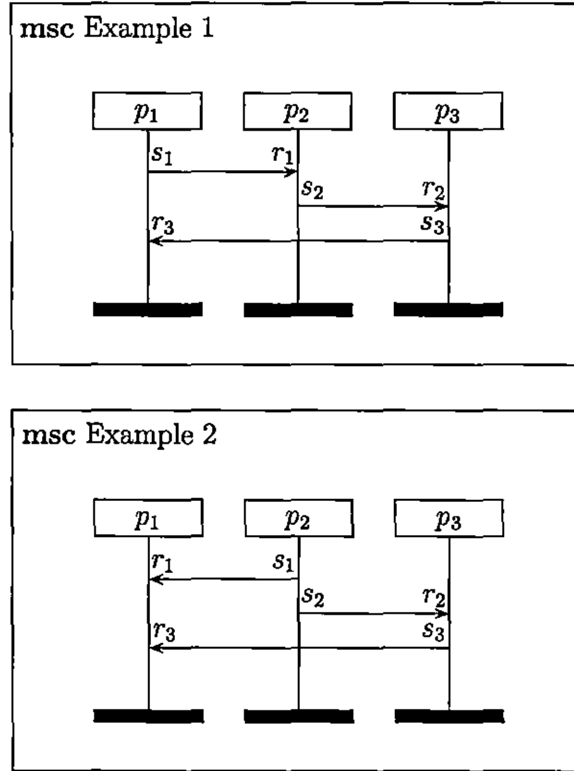
4.3.1 Two visual order translations

Recall the MSC from Figure 2.1:

Under the visual order, the corresponding CCS term is

$$(s_1.\overline{b}_1.c_3.r_3.0|c_1.r_1.s_2.\overline{b}_2.0|c_2.r_2.s_3.\overline{b}_3.0|b_1.\overline{c}_1|b_2.\overline{c}_2|b_3.\overline{c}_3)\backslash\{b_1, b_2, b_3, c_1, c_2, c_3\}$$

It is not hard to see that the only possible trace for this term is $s_1 r_1 s_2 r_2 s_3 r_3$, which corresponds exactly to the ordering of events under the visual order.



For the example from Figure 2.2 under the visual order we get

$$(c_1.r_1.c_3.r_3.0|s_1.\bar{b}_1.s_2\bar{b}_2.0|c_2.r_2.s_3.\bar{b}_3.0|b_1.\bar{c}_1.0|b_2.\bar{c}_2.0|b_3.\bar{c}_3.0)\backslash\{b_1, b_2, b_3, c_1, c_2, c_3\}$$

4.3.2 An enforced order example

Finally, consider the MSC in Figure 4.1. Suppose that in p_1 we have $r_1 << s_2 << r_4$. The term corresponding to p_1 will be

$$P_1 = (c_1.r_1.\bar{d}_2.\bar{d}_4.0|d_2.s_2.\bar{b}_2.\bar{d}_4.0|d_4.d_4.c_4.r_4.0)\backslash\{d_2, d_4\}$$

Suppose that in p_2 we have $s_1 << s_3$ and $s_1 << r_2$. The term corresponding to p_2 will be

$$P_2 = (s_1.\bar{b}_1.\bar{d}_2.\bar{d}_3.0|d_2.c_2.r_2.0|d_3.s_3.\bar{b}_3.0)\backslash\{d_2, d_3\}$$

Suppose that in p_3 we have $r_3 << s_4$. The term corresponding to p_3 will be

$$P_3 = (c_3.r_3.\bar{d}_4.0|d_4.s_4.\bar{b}_4.0)\backslash\{d_4\}$$

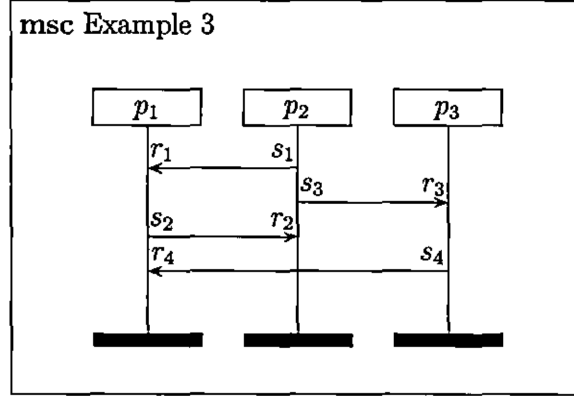


Figure 4.1. *Enforced order translation example*

Finally, the term representing the whole system will be

$$(P_1|P_2|P_3)\backslash\{b_1, b_2, b_3, b_4, c_1, c_2, c_3, c_4\}$$

It is not hard to check that these traces are all the traces for this CCS term, and that they are equal to all the total orderings consistent with the enforced ordering specified for the MSC:

$s_1r_1s_2r_2s_3r_3s_4s_4$ $s_1r_1s_3s_2r_3r_2s_4r_4$ $s_1s_3r_1s_2r_2r_3s_4r_4$ $s_1s_3r_1r_3s_4s_2r_2r_4$ $s_1s_3r_3r_1s_4s_2r_4r_2$
$s_1r_1s_2s_3r_2r_3s_4r_4$ $s_1r_1s_3s_2r_3s_4r_2r_4$ $s_1s_3r_1s_2r_3r_2s_4r_4$ $s_1s_3r_1r_3s_4s_2r_4r_2$ $s_1s_3r_3s_4r_1s_2r_2r_4$
$s_1r_1s_2s_3r_3r_2s_4r_4$ $s_1r_1s_3s_2r_3s_4r_4r_2$ $s_1s_3r_1s_2r_3s_4r_2r_4$ $s_1s_3r_3r_1s_2r_2s_4r_4$ $s_1s_3r_3s_4r_1s_2r_4r_2$
$s_1r_1s_2s_3r_3s_4r_2r_4$ $s_1r_1s_3r_3s_2r_2s_4r_4$ $s_1s_3r_1r_3s_2r_2s_4r_4$ $s_1s_3r_3r_1s_2s_4r_2r_4$
$s_1r_1s_2s_3r_3s_4r_4r_2$ $s_1r_1s_3r_3s_2s_4r_2r_4$ $s_1s_3r_1r_3s_2s_4r_2r_4$ $s_1s_3r_3r_1s_2s_4r_4r_2$
$s_1r_1s_3s_2r_2r_3s_4r_4$ $s_1r_1s_3r_3s_2s_4r_4r_2$ $s_1s_3r_1r_3s_2s_4r_4r_2$ $s_1s_3r_3r_1s_4s_2r_2r_4$

Now, we use the expansion law for CCS to show that the term of local process is equivalent to a term which corresponds to the + of all terms corresponding to a possible linear ordering of events in local process. Recall the MSC from Figure 4.1: Suppose that in p_1 we have $r_1 << s_2$ and $s_2 << r_4$. The term corresponding to p_1 will be

$$
\begin{aligned}
P_1 &= (c_1.r_1.\overline{d}_2.0|d_2.s_2.\overline{b}_2.\overline{d}_4.0|d_4.c_4.r_4.0)\backslash\{d_2, d_4\} \\
&= c_1.(r_1.\overline{d}_2|d_2.s_2.\overline{b}_2) + s_2.(\overline{b}_2.\overline{d}_4|d_4.c_4.r_4) \\
&= c_1.(r_1|\tau.s_2.\overline{b}_2) + s_2.(\overline{b}_2|\tau.c_4.r_4) \\
&= c_1.r_1.s_2.\overline{b}_2.0 + s_2.\overline{b}_2.c_4.r_4.0
\end{aligned}
$$

Suppose that in p_2 we have $s_1 << s_3$ and $s_1 << r_2$. The term corresponding to p_2 will be

$$
\begin{aligned}
P_2 &= (s_1.\overline{b}_1.\overline{d}_2.\overline{d}_3.0|d_2.c_2.r_2.0|d_3.s_3.\overline{b}_3.0)\backslash\{d_2, d_3\} \\
&= s_1.(\overline{b}_1.\overline{d}_2|d_2.c_2.r_2) + s_1.(\overline{b}_1.\overline{d}_3|d_3.s_3.\overline{b}_3) \\
&= s_1.(\overline{b}_1|\tau.c_2.r_2) + s_1.(\overline{b}_1|\tau.s_3.\overline{b}_3) \\
&= s_1.\overline{b}_1.c_2.r_2.0 + s_1.\overline{b}_1.s_3.\overline{b}_3.0
\end{aligned}
$$

4.4 Notes

The expansion law for CCS is given by Milner [Mil80], as one of the basic equational laws of CCS. The general expansion law can be given as:

$$
\begin{aligned}
(P|Q)\backslash L &= \sum\{a.(P'|Q) \ : \ P \xrightarrow{a} P' \text{ and } a, \overline{a} \notin L\} \\
&+ \sum\{\tau.(P'|Q') \ : \ P \xrightarrow{a} P' \text{ and } Q \xrightarrow{\overline{a}} Q'\} \\
&+ \sum\{(a.(P|Q') \ : \ Q \xrightarrow{a} Q' \text{ and } a, \overline{a} \notin L\}
\end{aligned}
$$

The $\sum$ just means the + of everything in the set. We could ignore the τ prefix because we are only concerned with actions which correspond to events in the MSC.

Chapter 5

Verifying assumed orderings using temporal logic

5.1 Temporal logics

We have seen how MSC's and CCS can be used to build models of concurrent systems. Building these kinds of models is an important part of formal methods for verifying the design of systems. One way of looking at these models is that they are actually abstract versions of the systems that we are interested in modeling. Another part of the verification process is the *specification* of properties that we expect the system in question to have. In a formal approach to verification, we need to have a formal language in which we can express these properties. For concurrent systems, or more generally *reactive* systems, researchers have found that *temporal logic* is a very useful specification language. Temporal logical takes ordinary propositional logic and adds *temporal operators.* For example, $\mathbf{F}p$ means that proposition p holds at some point in the future.

In order to give a formal meaning to temporal logic formulas we need to define a class of *models.* The models which are used in temporal logics for concurrent system are usually based on some kind of *transition system,* like we looked at in Chapter 3. The *states* of a transition system correspond to instants in time.

Temporal logics for these kinds of systems are either *linear time* or *branching time.* In linear time logics, the value given to a temporal proposition depends on whether or not it holds in all possible *traces* of the system. In branching time logics, the value given to a temporal proposition at a given instant of time (state) basically depends in some way on properties of the states that are reachable from it through some sequence

of transitions.

Another distinction in temporal logics depends on what kind of atomic propositions we allow. Most temporal logics only have atomic *state* propositions. Each state then has an associate valuation which indicates the atomic propositions which hold in the state. In this thesis, we are concerned with transitions between states (which represent events in an MSC) rather than internal properties of states. One logical approach to expressing properties involving transitions would be to use a logic like Hennessy-Milner logic or the modal μ-calculus which have *modalities* for expressing properties of transitions. We will take a slightly different approach and use the temporal logic *generalized computation tree logic* (GCTL*)*. This logic is a generalization of the more familiar computation tree logic* (CTL*). We choose this logic for a number of reasons, even though it is a branching time logic. First of all, there is tool support for verification of GCTL* formulas in CCS models. Secondly, GCTL* allows us to talk about transitions, and there doesn't seem to be any existing linear time temporal logic for doing this. Thirdly, the temporal operators of GCTL* are easier to understand that the modalities and fixed points of the μ-calculus.

In [CLS01] where GCTL* is introduced, it is given in an abstract form and in a concrete form which is used as input for the Concurrency Workbench tool. Here we will present a part of the concrete version which is good enough for the properties we will check.

5.2 Syntax of GCTL*

In GCTL*, as in CTL*, there are two kinds of formulas, *state formulas* and *path formulas*. Path formulas are interpreted over *paths*, which are maximal execution sequences of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$$

Note that in general such a sequence might be infinite, although this is not true for the systems that we will consider. For technical reasons, when doing GCTL* verification, finite paths are made infinite by adding a transition (with a dummy label δ that is not used anywhere else) from the last state in the sequence back to itself.

There are only two atomic state propositions, **tt** and **ff**. Atomic action proposi-

tions have the form $\{\alpha_1, \ldots, \alpha_k\}$. Θ stands for an arbitrary atomic action proposition. We cannot allow δ to be a member of Θ, since δ really means that no transition is possible (δ=deadlock).

Definition 5.1 The collections $\mathcal{S}$ of *state formulas* and $\mathcal{P}$ of path formulas are defined by the BNF grammar:

$$\mathcal{S} ::= \mathbf{tt} \mid \mathbf{ff} \mid \mathcal{S} \vee \mathcal{S} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathbf{A}\mathcal{P} \mid \mathbf{E}\mathcal{P}$$
$$\mathcal{P} ::= \Theta \mid \neg\Theta \mid \mathcal{S} \mid \mathcal{P} \vee \mathcal{P} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathbf{X}\mathcal{P} \mid \mathcal{P}\mathbf{U}\mathcal{P} \mid \mathcal{P}\mathbf{R}\mathcal{P}$$

5.3 Semantics of GCTL* formulas

We define the meaning of path formulas first. Suppose that φ is a path formula and $p = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$ is a computation path. Let p_i denote $s_i \xrightarrow{\alpha_i} s_{i+1} \xrightarrow{\alpha_{i+1}} \ldots$ (in particular $p = p_0$). We can define p *satisfies* φ (notation: $p \models \varphi$) by induction on the structure of φ.

- $p \models \Theta$ if $\alpha_1 \in \Theta$
- $p \models \neg\Theta$ if $\alpha_1 \notin \Theta$
- $p \models \varphi \vee \psi$ if $p \models \varphi$ or $p \models \psi$
- $p \models \psi \wedge \psi$ if $p \models \varphi$ and $p \models \psi$
- $p \models \mathbf{X}\varphi$ if $p_1 \models \varphi$ when $\alpha_1 \neq \delta$ and if $p \models \varphi$ otherwise.
- $p \models \varphi\mathbf{U}\psi$ if there is some i such that $p_i \models \psi$ and for every $j < i$ $p_j \models \varphi$ (φ holds *until* ψ holds).
- $p \models \varphi\mathbf{R}\psi$ if for every i and every $j < i$, if $p_j \not\models \phi$ then $p_i \models \psi$ (ψ is *released* by φ).

The set of temporal operators from GCTL* is enough to define all the other temporal operators (the *always* operator $\mathbf{G}$ is defined by $\mathbf{G}\varphi \equiv \mathbf{ff}\mathbf{R}\varphi$, and the *eventually* operator $\mathbf{F}$ is defined by $\mathbf{tt}\mathbf{U}\phi$).

Once we have defined the semantics for path formulas it is easy to do the same for state formulas. For any s, $s \models \mathbf{tt}$ and $s \not\models \mathbf{ff}$. The definition of $s \models \phi \wedge \psi$ and $s \models \phi \vee \psi$ is just the definition from propositional logic. $s \models \mathbf{A}\phi$ if for *every* path p starting at state s, $p \models \phi$. $s \models \mathbf{E}\phi$ if for *some* path p starting at s, $p \models \phi$.

For a LTS to satisfy a formula it means that the *initial state* in the LTS satisfies the formula. We have to pick a state which is the initial state. For the LTS which is defined by the term $\mathcal{A}_M$ which comes from a MSC M, the initial state is just $\mathcal{A}_M$ itself.

5.4 Representing inferred orderings in GCTL*

We are now ready to show how we can describe user inferred orderings $e \sqsubset f$ over an MSC M using a GCTL* formula $\phi_{e \sqsubset f}$ over the LTS of observable actions of $\mathcal{A}_M$. Let $\phi_{e \sqsubset f}$ be the formula:

$$\mathbf{A}(\neg\{f\}\mathbf{U}\{e\})$$

Then $\mathcal{A}_M \models \phi_{e \sqsubset f}$ iff on every path starting from $\mathcal{A}_M$, there is some i such that $p_i \models \{e\}$ and for every $j < i$, $p_j \models \neg\{f\}$. But this is true iff on every path starting from $\mathcal{A}_M$, there is some i such that $s_i \xrightarrow{e} \ldots$, but for every $j < i$, $s_j \xrightarrow{g} \ldots$ where $g \neq f$. But this is the case iff in every trace of $\mathcal{A}_M$, e precedes f in every $t \in \text{traces}(\mathcal{A}_M)$. By 4.5, this is true iff $e <^* f$.

So we have shown how to solve the basic problem of checking whether a user inferred ordering is consistent with the visual ordering or with the enforced ordering of an MSC using temporal logic model checking of CCS-based models. This means that we can use an off-the-shelf tool like the Concurrency Workbench to this checking. In Appendix A we describe a tool that we have developed to partially automate this process.

Once we have introduced GCTL* as a language for expressing properties of MSC's using the CCS translation, we can use it to specify other properties besides the consistency of an inferred ordering. Actually, full GCTL* is more powerful than we really need, because it can be used to give *branching time* properties of systems. The semantics we gave for MSC's just involves the orderings on events, and so all the meaning of an MSC is given by the possible linear orderings of events. We showed in Chapter 4 that this is the same as the traces of the CCS term defined from the MSC. So we can really only use GCTL* to model *linear time* properties of MSC's. For GCTL*, we can do this by just using formulas which have the form $\mathbf{A}\varphi$ where φ is a path formula.

of Emerson and Clarke [CE81],[EC82]. Model-checking is now well established and is a widespread technique in verification (see [CGP99]).

Usual verification of systems modeled in CCS was done with specifications given in temporal logic known as the μ-calculus, which was originally formulated in [Koz83]. Original versions of the Concurrency Workbench (see [CPS93]) used μ-calculus, but recent versions [CLS01] have introduced more user-friendly logics such as GCTL*. GCTL* is closely related to the logic CTL* which is used in the SMV model checker [CGP99]. The main difference between the two is the CTL* is appropriate for specifying properties of the *states* of a system, while GCTL* is better for specifying properties of the *transitions* of a system. This is why GCTL* is better for verifying CCS models, because of the labelled transitions in CCS.

A lot of work in model checking uses *linear time* temporal logic (LTL) which was introduced by Pnueli [Pnu77], and which does not have **A** and **E** operators and can specify properties of traces. The reason that we did not use LTL is that the LTL logics in verification tools express properties of states of a system and not transitions. Also there aren't any LTL tools that work with CCS.

There is work on μ-calculus model checking such as [Sti01], [ASW94] has looked at *compositional* proof systems, in which the model-checking algorithm uses the structure of CCS terms instead of working with an unstructured labelled transition system. This can be helpful when dealing with problems like *state explosion* [BCM+90]. Compositional proof systems are also good for working with infinite-state systems [GBK96].

Chapter 6

Conclusion

In this thesis, we have demonstrated the possibility of using off-the-shelf technology in order to develop techniques for verification in a given domain using existing techniques from another domain. By choosing a powerful existing technology (CCS and temporal logic) we were able to solve a verification problem for Message Sequence Charts (MSC's), which at first do not look very similar to process algebra models. Our approach consisted of two parts: a general technique for translating MSC's into CCS terms, and a technique for specifying user-inferred orderings for an MSC using the temporal logic GCTL*. An important feature of our approach is that we provided a general proof of correctness for the translation, which ensures the correctness of the technique for checking that a user-inferred ordering is consistent with the visual or enforced order of the MSC. We have created a tool which automates the translation of MSC's into CCS. This provides a way to automatically check consistency as described above.

Because of the generality of our translation, it is possible to extend it in a number of ways. First of all, once we have a CCS term corresponding to a MSC, we express other properties of the MSC in GCTL* and check them with a model-checking tool like CWB. The problem of general model checking of MSC's has been investigated by other researchers [AY99] who were interested in developing direct techniques for checking temporal properties of MSC's. Our approach reduces this problem to the well-studied existing problem of checking temporal properties for CCS which should be quite effective from a practical standpoint, although it may not match the efficiency which is at least theoretically possible using a direct approach. This would be a very interesting area for further research. A second natural way to extend the results presented here would be to consider more general models for MSC's. We believe

that our approach should "scale-up" to these more general models in a natural way, especially since the parallel composition and hiding operators of CCS are designed to make this sort of modular construction of systems easy.

A summary of the benefits of the approach presented in this thesis are:

- By reducing the problem of model checking MSC's to model checking of CCS we can take advantage of any advances in the technology for CCS model checking. Our system can plug in to any CCS model checker that recognizes GCTL*.
- The way to define CCS terms from MSC's is very modular. The structure of the terms only depends on the visual or enforce orderings for local processes. This means that it should be easy to model extra features of MSC's without really changing the definitions and correctness proof that we gave.
- Our translation results in CCS terms which are not too big in terms of the number of events in the original MSC (at most squared). The translation can be done efficiently. The terms we get have an intuitive interpretation which is related to the interpretation of MSC's.

A summary of the future work we could do will include

- Give a full proof of the translation for the enforced ordering.
- Extend the translation to more complicated models such as hierarchical MSC's or MSC graphs. When we have done this we could look at verifying a real-world example using our methods.
- Look at the structure of the CCS terms that are generated by the translation. Are there classes of terms for which compositional model checking could help to get a more efficient verification?
- Look at more general properties of MSC's. Are there interesting *branching time* properties? These are properties which do not depend just on traces but on possible branches in the execution. Branching time properties can be expressed in logics like GCTL*. We would have to develop branching-time semantics for MSC.

Bibliography

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. In 28th *International Colloquium on Automata, Languages and Programming*, number 2076 in Lecture Notes in Computer Science, pages 797–808. Springer-Verlag, 2001.

[AHP96] R. Alur, G.J. Holzmann, and D.A. Peled. An analyzer for message sequence charts. *Software – Concepts and Tools*, 17(2):70–77, 1996.

[AR88] I.J. Aalbersberg and G. Rozenberg. Theory of traces. *Theoretical Computer Science*, 60:1–82, 1988.

[ASW94] A. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal mu-calculus. In *LICS: IEEE Symposium on Logic in Computer Science*, 1994.

[AY99] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *10th International Conference on Concurrency Theory*, number 1664 in Lecture Notes in Computer Science, pages 114–129. Springer-Verlag, 1999.

[BCF01] G. Berry, H. Comon, and A. Finkel, editors. *Computer-aided Verification, 13th International Conference, CAV 2001,Proceedings.* Number 2102 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[BCM+90] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[CE81] E.M. Clarke and E.A. Emerson. Design and verification of syncrhonization skeletons using branching time temporal logic. In *Logics of Programs Workshop*, number 131 in Lecture Notes in Computer Science, pages 52–71. May 1981.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press,

1999.

[CLS01] R. Cleaveland, T. Li, and S. Sims. *The Concurrency Workbench of the New Century User's Manual, Versoin 1.2*, July 2001.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[dBZ] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120.

[DDHY92] D. Dill, A. Drexler, A. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize syncrhonization skeletons. *Science of Computer Programming*, 2, 1982.

[fS86] International Organization for Standards. Information processing systems – open systems interconnection – lotos– a formal description technique based on the temporal ordering of observational behaviour, 1986.

[GBK96] D. Gurov, S. Berezin, and B.M. Kapron. A modal mu-calculus and a proof system for value passing processes. *Electronic Notes in Theoretical Computer Science*, (6), 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequestial Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Hol97] G. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[ITU93] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[Mil80] M. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[MR94] A. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[Pel01] D.A. Peled. *Software Reliability Methods*. Springer-Verlag, New York, 2001.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, 1977.

[PUW+99] R. Probert, H. Ural, A. Williams, J. Li, and R . Plackowski. Experience with rapid generation of functional tests using mscs, sdl, a nd ttcn. In *Special Issue of Formal Descriptions Techniques in Practice of Comp uter Communications*, 1999.

[Qua01] Terry Quatrani, editor. *Introduction to the Unified Modeling Language*. Rational Developer Network, 2001.

[Rei85] Wolfgang Reisig. *Petri Nets (an Introduction)*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, Berlin-Heidelberg-New York, 1985.

[RGG] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, (28):1629–1641.

[Sti01] C. Stirling. *Modal and Temporal Properties of Processes*. Springer-Verlag, 2001.

Appendix A

Modelling MSC into CCS Translation Tool User Manual

We now describe a tool which can translate any visual order MSC M into a CCS term $\mathcal{A}_M$

A.1 Introduction

Modeling MSC into CCS translation Tool provides an interface for users to enter all the components of a MSC such as the events and processes. And, it based on the translation definition of visual order and enforced order established in this thesis paper, translates the semantics of MSC into CCS terms. The tool can also produce an output text file for the Concurrency Working Bench which is a verification tool for validating all the syntax rules of Calculus of Communicating Systems (CCS). This manual specifies the installation criteria and procedure first, and then it will briefly introduce the interface for the users. Notation outlines the symbols representation and Error Message Description describes possible error messages.

A.2 Installation

This tool is written in Visual Basic 6.0. In order to run the tool properly on your computer, your computer should meet the following criteria:

- Windows Operation System: Windows 95 or later version.
- Memory 128MB and hard drive size 1.2GB

- Mouse installed

Procedure of internet downloading:

- Download the tool at http://www.csc.uvic.ca/~wchiu/Msc.msi
- Follow the instruction from the pop-up windows
- Reboot your computer when its done
- Double Click msc.exe to run the program

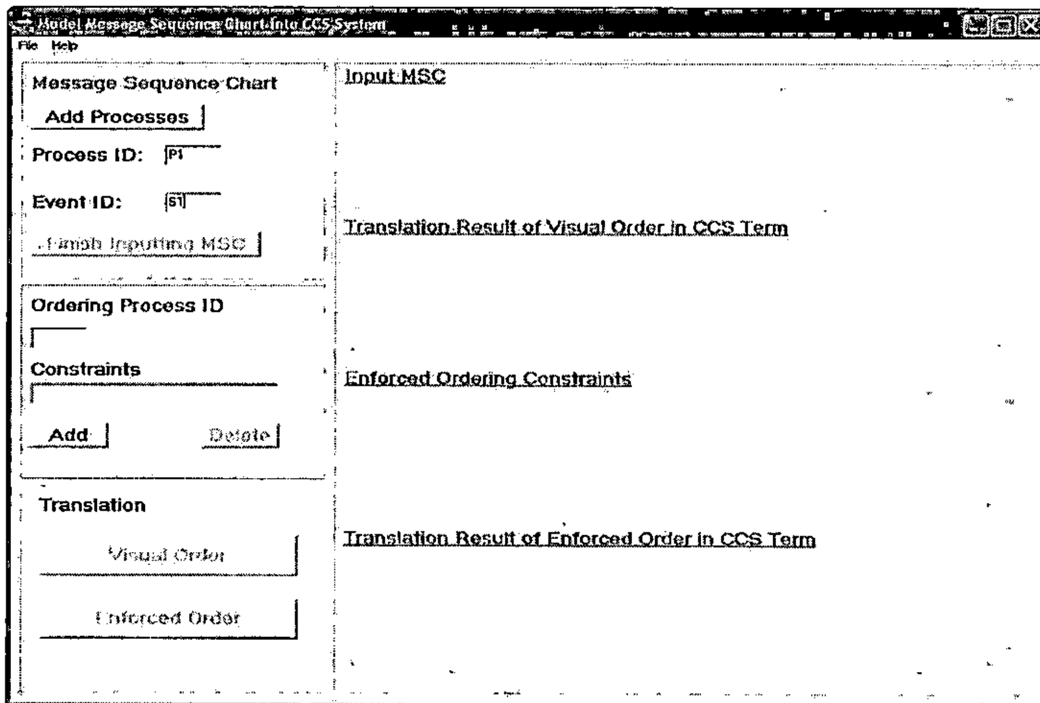
If any errors occur during the installation procedure, you can re-install the tool. The installation program will restore all system files used by the tool.

A.3 User Interface

The interface of the tool is made up of three parts: User Input Menu, Translation Output and Pull-down Menu.

A.3.1 User Input Menu

The left side of the window is for user input. The upper part of the main menu allows users to input the components of a MSC such as processes and events. The middle part allows users to input the constraints (enforced order) of each of the processes of the MSC, and the bottom part allows users to translate the visual order and the enforced order into CCS.



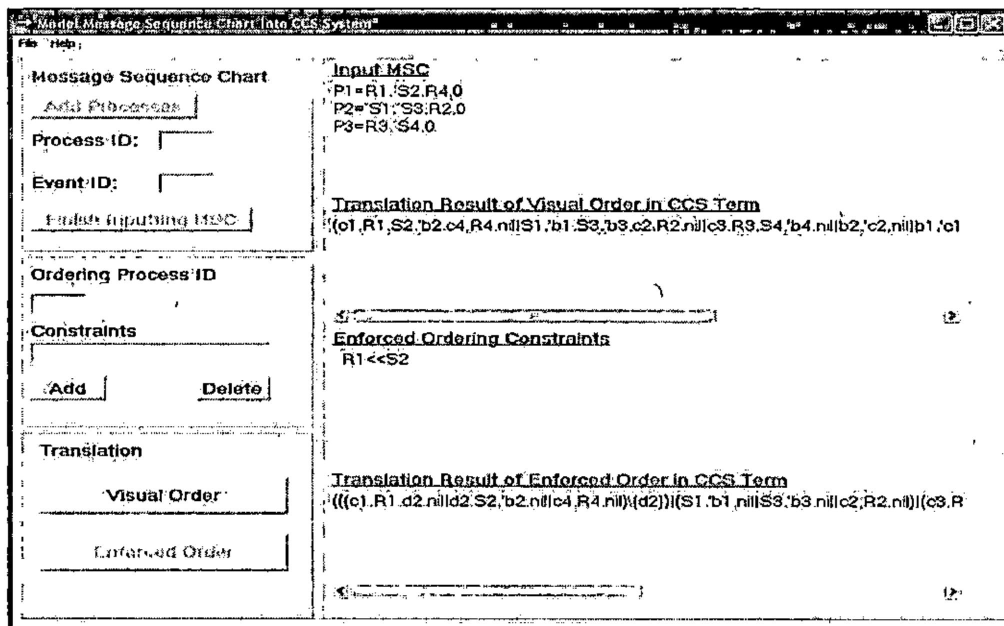
- Add Processes command button: Create a new process of a message sequence chart.
- Process ID Text box: Input process id. Users have to type P for process followed by a number such as P1, P2 or P3. There is no limit on the number of entries. An error message box will appear if users violate the criteria.
- Event ID Text box: Input the event id of each process. Users have to type S for send event or R for received event followed by a number such S1, R1 or S2. There is no limit on the number of entries. Users have to press Enter in order to accept the input event before entering another event for the process. An error message box will appear if users violate the criteria.
- Finish Inputting MSC command button: Translate MSC into CCS. Users have to click this button in order to translate the input MSC into CCS. In addition, the visual order button will be enabled.
- Process ID text box: Input the process id that belongs to the message sequence chart.
- Constraint text box: Input the constraints (enforced order). Users have to use

$\ll$ between the events in order to specify which event is enforced by another e.g. Enter S1$\ll$R2 if R2 is enforced by S1. When users finish typing one set of enforced order, they have to click on Add Button in order to input another set of enforced order.

- Add command button: Add constraints (enforced order) of the process. Users can add more than one set of constraints on the process before doing enforced order translation.
- Delete command button: Delete constraints (enforced order) of the process. Users can click this button to delete all the constraints on the process that have been translated to enforced order.
- Visual Order Command Button: Translate the MSC visual order into CCS terms.
- Enforced Order Command Button: Translate the constraints (enforced order) into CCS terms.

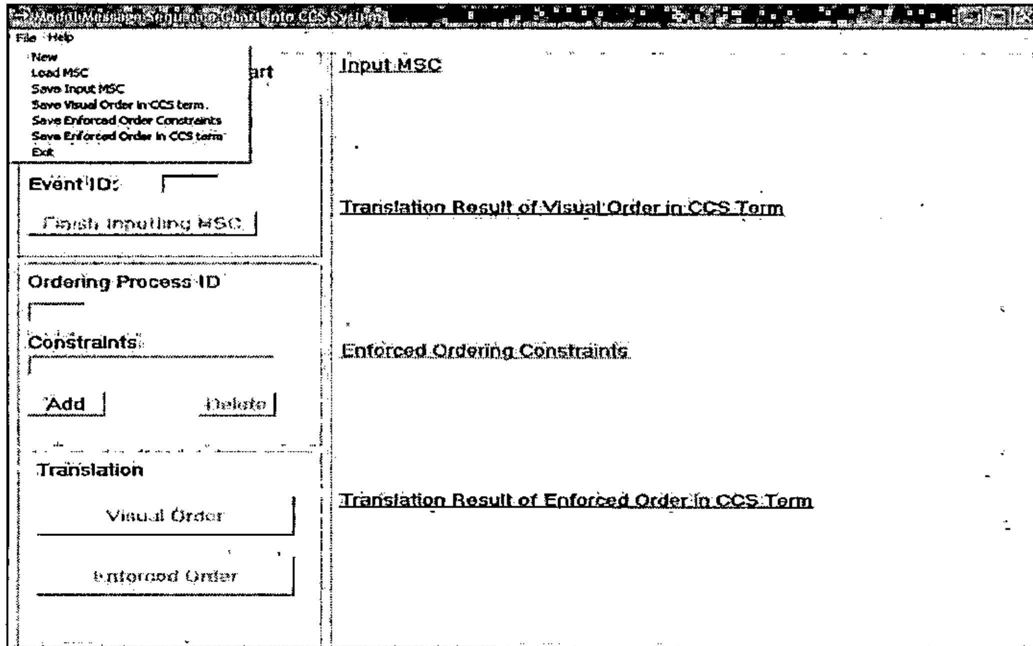
A.3.2 Translation Output

The right side of the window of the main menu gives out the Input MSC in CCS terms, the translation result of visual order, the input constraints of each of the processes of the MSC and the resulting translation of enforced order.



- Input MSC: Display result of translation of MSC into CCS terms.
- Translation Result of Visual Order in CCS terms: Display visual order result of translating MSC into CCS.
- Enforced Ordering Constraints: Display the user input constraints (enforced order) when user clicks on Add Button. If users click on Delete Button, the constraints will be removed from this box.
- Translation Result of Enforced Order in CCS Term: Display the result of translating enforced order of MSC into CCS.

A.3.3 Pull-down Menu



- New: Clear current MSC and create a new MSC.
- Load MSC: Load an MSC from a text file named

  `c:\temp\InputMSC.txt.`

- Save Input MSC: Save the contents of the Input MSC box as a text file named c:\temp\InputMSC.txt.
- Save visual order in CCS term: Save the contents of the Translation Result of Visual Order in CCS Term box as a text file named c:\temp\VisualOrderMSC.CCS.
- Save Enforced Order Constraints: Save the contents of the Enforced Ordering Constraints box as a text file named c:\temp\OrderingConstraint.txt.
- Save Enforced Order in CCS term: Save the contents of the Translation Result of Enforced Order in CCS Term box as a text file named c:\temp\ConstraintsMSC.CCS.

A.4 Conclusion and Future Work

The tool provides a user interface to enter or load MSC data and output the resulting translation to the displayed boxes. Users can verify the result and save the result as text file and that can be verified on Concurrency Workbench. For the future development, the tool can be provided with more features to the user such as change and delete functions for inputting MSC and the constraints (enforced order).

A.5 Notation

Notation	
P	Process
S	Send Event
R	Received Event
'	The sending out signal
$\|$	Concurrency
$\ll$	enforced operator
0	Stop
.	The events are in sequence

Table A.1. *Notation Table*

A.6 Error Message Description

Error Message Please enter the process id

Solution Enter process id in the process id text box

Error Message No process id specified

Solution Enter process id in the process id text box

Error Message Invalid event id

Solution Please type s to specify send message, r to specify received event followed by a number

Error Message Invalid Process ID

Solution Please enter p followed by a number to specify process id

Error Message No enforced order specified

Solution Enter the set of constraint at the enforced order text box

Error Message Event id not exist in the process

Solution Enter the correct event id

Error Message The enforced order violate the visual order

Solution Enter the correct set of constraint

Error Message Race condition, Race Failed!

Solution Deadlock occurs, start again

Error Message Duplicated process id

Solution The process id has been used, enter a new one

Error Message Duplicated event id

Solution The event id has been used, enter a new one

Error Messages The enforced order violate the visual order or the enforced order constraints not match or duplicate event

Solution Re-enter the constraint of enforced order

Error Message Path (76) not found

Solution Create c:\temp directory

If an unexpected result is obtained, click New button in order to start again. If the condition happens again, restart the program.

Appendix B

Program Code

Introduction

The followings are the data structure and the main methods of the tool.

a) Declaration

The transitelt is to store the labels of events.

```
Type transitelt
    element As String               'event label
End Type
```

The datatype of localitem is the internal blocking labels of enforced order.

```
Type LocalItem
    intoken() As transitelt         'internal blocking labels from the other events
    outtoken() As transitelt        'sending out blocking labels to the other events
End Type
```

The data type of orders uses to store the basic components of MSC
like process id and events labels.

```
Type Orders
     mark As Boolean            'marked for checking events
     priority As Integer        'checking the ordering of events
     elt As String              'name of events
     signal As String           'signal to the buffer if send/receive happened
     enforced As Boolean        'checking if the local process has been enforced or not
```

```
    token As LocalItem        'checking if the event has internal blocking labels
End Type
```

The buffer data types are the blocking label of event.

```
Type buffer
    Orderevent() As Orders      'no of events per process
    process As String           'process label
    forced As Boolean           'mark for enforced
End Type

Type eventitem
    eltout As String            'blocking labels of buffer(b or c)
End Type

Type buff                       'for send event
   eventelt(2) As eventitem     'send/receive buffering
End Type

Global BufferOut() As buffer    'Process and events MSC
Global buffevent() As buff      'send events buffer blocking label only
Global EnforceEvent() As transitelt     'event label of enforced order
```

b) Methods

This procedure allows the user to enter the process and events of MSC and store the information on a temporary string called visualorderstr. Also, it will display the user input data to the control box resultscreen.

```
Private Sub AddProcess_Click()
Dim templen, i, j As Integer                          'declaration

On Error Resume Next
processid.Enabled = True                              'controls setting
EventID.Enabled = True
processid.SetFocus
```

```
If processid.Text = "" And loaded = False Then           'check user input (process)
    processid.SetFocus
    Exit Sub
End If

templen = 0
If loaded = True Then
    For i = 1 To Len(visualorderstr)
        If Mid(visualorderstr, i, 1) = "S" Then
            maxsendbuffer = maxsendbuffer + 1            'create buffer of each send event
        End If
    Next
End If
If Trim(EventID.Text) <> "" Then                         'check user input (events)
    If Mid(Trim(EventID.Text), 1, 1) = "S" Then
        maxsendbuffer = maxsendbuffer + 1                'create buffer of each send event
    End If                      'in case if any send event left out
    visualorderstr = visualorderstr + Trim(EventID.Text)
                                'store events label for each of process
End If
If Mid(visualorderstr, 1, 1) = "=" Then                  'display each local process and events
    visualorderstr = Mid(visualorderstr, 2, Len(visualorderstr))
End If

i = 1 templen = 0 MaxProcess = MaxProcess + 1 'count the number of process
Do Until i > Len(Trim(visualorderstr))
    If Mid(Trim(visualorderstr), i, 1) = "S" Or Mid(Trim(visualorderstr), i, 1) = "R" Then
        templen = templen + 1                  'count the number of event of each process
    End If
    i = i + 1
Loop
If maxlen < templen Then                       'set the number of event of each process
    maxlen = templen
End If
If loaded = False Then                         'Display the user input result
    ResultScreen.Text = ResultScreen.Text & Trim(processid.Text)
            & "=" & Trim(visualorderstr) & ";" & vbNewLine
Else
    ResultScreen.Text = ResultScreen.Text & Trim(visualorderstr) & ";" & vbNewLine
```

```
End If
processid.Text = ""                                'clear space for user input

EventID.Text = ""
switch = True
visualorderstr = ""
'visual.Enabled = True
processid.SetFocus
finish.Enabled = True
End Sub


This procedure will verify and store the MSC components to array,
translate each process of msc into ccs and display the result at
the control text box called resultscreen.
---------------------------------------------------------------------------------------------
Private Sub finish_Click()
Dim i, j As Integer
'to translate the msc directly into ccs and
'to allocate space for the msc components
'On Error GoTo err_des

If Trim(EventID.Text) <> "" Or Trim(processid.Text) <> "" Then
    Call Command1_Click                'check if there has any leftout data
End If
Call Init_Buffer                       'init the size of array and buffer
Call Add_Event_Process                 'verify/store the process and events into array
ResultScreen.Text = ""                 'free space for the control text box
For i = 1 To MaxProcess                'display the output to the text box
    ResultScreen.Text = ResultScreen & BufferOut(i).process & "="
    For j = 1 To maxlen
        If Trim(BufferOut(i).Orderevent(j).elt) <> "" Then
            If Mid(BufferOut(i).Orderevent(j).elt, 1, 1) = "S" Then
                ResultScreen = ResultScreen & Chr(146)
                & BufferOut(i).Orderevent(j).elt & "."
                BufferOut(i).Orderevent(j).signal = "b"
                & Mid(BufferOut(i).Orderevent(j).elt, 2)
            Else
                ResultScreen = ResultScreen & BufferOut(i).Orderevent(j).elt & "."
```

```
                BufferOut(i).Orderevent(j).signal = "c"
                & Mid(BufferOut(i).Orderevent(j).elt, 2)
            End If
        End If
    Next
    ResultScreen = ResultScreen & "0" & vbNewLine
Next

visual.Enabled = True                        'reset controls setting
processid.Enabled = False
EventID.Enabled = False
Command1.Enabled = False
Command3.Enabled = True
finish.Enabled = False
'ProcessNo.SetFocus
result.Enabled = True
End Sub
```

```
This procedure will verify the MSC based on the semantic of msc.
Afterward, the program will translate the MSC's visual order into
CCS and display the result to the control text box called
visualorderresultscreen.
------------------------------------------------------------------------------------------
Private Sub visual_Click()
Dim i, j As Integer
Dim trace As Boolean
On Error Resume Next
                                             'trace the visual order result
trace = True
ResultScreen.Enabled = True
If Trim(BufferOut(1).process) = "" And Trim(BufferOut(2).process) = "" Then
    MsgBox "No specified Process/event"      'error checking
    processid.SetFocus
    Exit Sub
End If

Call Trace_VisualOrder_Process    'check the ordering is valid according to the visual order
For i = 1 To MaxProcess           'check if each send event having corresponded receive event
```

```
    For j = 1 To maxlen
        If (BufferOut(i).Orderevent(j).mark = False) 'event is being checked
    And (Mid(BufferOut(i).Orderevent(j).elt, 1, 1) = "R") Then
            MsgBox ("race condition, Race Failed!")
            trace = False
            Exit Sub
        End If
    Next
Next
For i = 1 To maxlen              'check if any error occurs
    For j = 1 To 2
        If Trim(buffevent(i).eventelt(1).eltout = "")
    Or Trim(buffevent(i).eventelt(2).eltout = "") Then
            MsgBox ("race condition, Race Failed!")
            trace = False
            Exit Sub
        Else
        End If
    Next
Next
j = 1
If trace = True Then             'Translate and display the result of visual order in CCS
    VisualOrderResultScreen.Text = ""
    For i = 1 To MaxProcess
        Do Until j > maxlen
            If Trim(BufferOut(i).Orderevent(j).elt) <> "" Then
                                          'send events look like 'si.'bi.o
                If Mid(BufferOut(i).Orderevent(j).elt, 1, 1) = "S" Then
                    VisualOrderResultScreen = VisualOrderResultScreen
            & BufferOut(i).Orderevent(j).elt & "."
                    VisualOrderResultScreen = VisualOrderResultScreen & "'"
            & BufferOut(i).Orderevent(j).signal & "."
                Else                      'receive events look like ci.ri.o
                    VisualOrderResultScreen = VisualOrderResultScreen
            & BufferOut(i).Orderevent(j).signal & "."
                    VisualOrderResultScreen = VisualOrderResultScreen
            & BufferOut(i).Orderevent(j).elt & "."
                End If
                j = j + 1
```

```
            Else
                j = j + 1
            End If
            Loop
            'End If
        j = 1
        VisualOrderResultScreen = VisualOrderResultScreen & "nil" & "|"
    Next i                  'next process
    j = 1
    For i = 1 To maxsendbuffer          'display the buffer of events
        Do Until j > 2
            If Mid(buffevent(i).eventelt(j).eltout, 1, 1) = "c" Then
                VisualOrderResultScreen = VisualOrderResultScreen & "'"
        & buffevent(i).eventelt(j).eltout & "."
            Else
                VisualOrderResultScreen = VisualOrderResultScreen
        & buffevent(i).eventelt(j).eltout & "."
            End If
            j = j + 1
        Loop
        j = 1
        If i < maxsendbuffer Then
            VisualOrderResultScreen = VisualOrderResultScreen & "nil" & "|"
        Else
            VisualOrderResultScreen = VisualOrderResultScreen & "nil"
        End If
    Next i
End If
VisualOrderResultScreen = "(" & VisualOrderResultScreen & ")\{"
j = 1
For i = 1 To maxsendbuffer          'display the buffering of each event
    Do Until j > 2
        VisualOrderResultScreen = VisualOrderResultScreen
    & buffevent(i).eventelt(j).eltout & ","
        j = j + 1
    Loop
    j = 1
Next i
VisualOrderResultScreen = Mid(VisualOrderResultScreen, 1,
```

```
            Len(VisualOrderResultScreen) - 1) & "}"
finish.Enabled = False            'reset the controls
ProcessNo.SetFocus
End Sub
```

This procedure gets the enforced order from the user, verifies the orders, adds to the array and displays the valid enforced order on the control text box called OrderingConstraints.

```
Private Sub Add_Click()
Dim i As Integer
Dim temp As String

temp = ""
enforcedOrderstr = enforcedOrderstr & Trim(enforcedorder.Text)
If Trim(enforcedOrderstr) <> "" Then
    For i = 1 To Len(enforcedOrderstr)              'store the user input into a string
        If Mid(enforcedOrderstr, i, 1) = ","
        And Mid(enforcedOrderstr, i + 1, 1) <> " " Then
            temp = Mid(enforcedOrderstr, 1, i) & " "
           & Mid(enforcedOrderstr, i + 1)            'verify the events
            If Check_Event_Process(ProcessNo.Text, Trim(enforcedorder.Text)) = True Then
                Call Add_enforcedevent(Trim(enforcedorder.Text))
                                                     'store the enforced order into array
                If Check_Duplicate_Enforced(enforcedorder.Text) = True
                                                     'verify the data (if duplicated)
          And Check_Priority(ProcessNo.Text, enforcedorder.Text) = True Then
                                                     'check the visual order
                    enforcedOrderstr = temp          'of the enforced order
                Else                                 'if error occurs
                    Call Remove_Event_Process(ProcessNo.Text, Mid(enforcedorder.Text, 1,
             Len(enforcedorder.Text) - 1))           'clear array data
                    Call Delete_string(enforcedOrderstr, enforcedorder.Text)
                                                     'clear the enforcedorder string
                    ProcessNo.SetFocus               'and orderingconstraints
                    ProcessNo.Text = ""
                    enforcedorder.Text = ""
                    enforcedOrderstr = ""
                    Exit Sub
```

```
                End If
            Else
                 MsgBox "Event ID: " & Mid(enforcedorder.Text, 1,
                 Len(enforcedorder.Text) - 1)          'if error occurs
                    & " not in Process ID: " & ProcessNo.Text
                 Call Delete_string(enforcedOrderstr, enforcedorder.Text)
                                                       'clear the enforcedorder string
                 Call Remove_Event_Process(ProcessNo.Text,
                 Mid(enforcedorder.Text, 1,            'and orderingconstraints
            Len(enforcedorder.Text) - 1))              'clear array data
                 ProcessNo.SetFocus
                 ProcessNo.Text = ""
                 enforcedorder.Text = ""
                 enforcedOrderstr = ""
                 Exit Sub
            End If
        End If
    Next
Else
    MsgBox "Please enter set of enforced order"
End If
If Trim(enforcedorder.Text) <> "" Then
                        'display enforced orderings to orderingconstraints text box
    OrderingConstraints.Text = OrderingConstraints.Text & "  "
    & Mid(enforcedorder.Text, 1,
    Len(enforcedorder.Text) - 1) & "  "
End If
nextprocess = True                                       'reset all the controls
result.Enabled = True
enforcedorder.Text = ""
enforcedorder.SetFocus
EnforcedDelete.Enabled = True
End Sub

This procedure will get the user input, clear the enforced
ordering data from the array and update the orderingcontraints
displayed box.
-------------------------------------------------------------------------------------------
Private Sub Delete_Click()
```

```
Dim response As Integer
Dim i As Integer

On Error Resume Next

If Trim(OrderingConstraints.Text) = "" Then     'check if the displayed box is empty
    MsgBox "No Constraint has to be deleted"
    ProcessNo.SetFocus
    Exit Sub
End If

If Trim(ProcessNo.Text) = "" Then              'check user input of process to be enforced
    MsgBox "Please enter the process ID."
    ProcessNo.SetFocus
    Exit Sub
End If
If Trim(enforcedorder.Text) = "" Then           'check the user input of enforced order
    MsgBox "Please enter the constraints of " & ProcessNo.Text
    enforcedorder.SetFocus
    Exit Sub
End If

    response = MsgBox("Do you want to delete this constraint? " 'confirm delete message
    & Mid(enforcedorder.Text, 1, Len(enforcedorder.Text) - 1),
    vbYesNo + vbQuestion, "MSC")
    If response = vbYes Then                        'if confirm
        Screen.MousePointer = vbDefault
        If Check_Event_Process(ProcessNo.Text, enforcedorder.Text) = True Then
            If Delete_Constraints(Mid(enforcedorder.Text, 1,
                                                    'clear data from user input
        Len(enforcedorder.Text) - 1)) = True Then   'clear array if necessary
                If Trim(EnforcedResultScreen.Text) <> "" Then
                    Call Remove_Constraint(enforcedorder.Text)
                End If
            End If
            Call Delete_string(enforcedOrderstr, enforcedorder.Text)
                                        'clear stored string for enforced order
            result.Enabled = True
            enforcedorder.Text = ""
```

```
            enforcedorder.SetFocus
        Else                            'enforced order events not found
            MsgBox "Event ID " & Mid(enforcedorder.Text, 1,
        Len(enforcedorder.Text) - 1) & " not exist in the process "
        & ProcessNo.Text
            ProcessNo.Text = ""
            enforcedorder.Text = ""
            ProcessNo.SetFocus
        End If
    End If

End Sub
```

User can either add or delete an enforced order by entering the enforced order at the control box called enforcedorder. The add and delete program will verify the enforced order and display or update the enforced order list at the control box called orderingconstraints. This procedure will take the content of this control text box and verify if the enforced order events are at the same local process or not. Then, if the user adds the enforced order, it will store the data including internal blocking labels to the array. If the user deletes the enforced order, it will clear the data from the array and update the content of the orderingconstraints control text box. Afterwards, it will call a procedure Display_Enforced_order to translate and display the whole MSC with visual and enforced order translated into CCS at the enfrocedorderresultscreen control text box.

```
Private Sub EnforcedOrder_Click()
Dim i, j, k, m, n As Integer                        'declaration
Dim currprocess As String
Dim tempprocess As String
Dim tempstr, tempforcestr As String
                                                    'verify the enforced order
On Error Resume Next

tempstr = ""                                        'init a temporary string
```

```
If Trim(enforcedOrderstr) = "" Then                 'check input
    MsgBox ("No constraint specified")
    nextprocess = True
    duplicated = False
    Call Display_Enforced_order(ProcessNo.Text)
                        'display the msc into ccs with enforced order
    enforcedorder.SetFocus  'if no enforced order
    Exit Sub
End If

m = 1                   'init counter variables
n = 1
tempforcestr = Trim(enforcedOrderstr)
                        'store the enforced order event labels into a string
Do Until m > Len(tempforcestr)
    If Mid(tempforcestr, m, 1) = "," Then
        tempstr = Trim(Mid(tempforcestr, n, m))
        m = m + 2
        n = m
        If Check_Event_Process(ProcessNo, tempstr) = False Then
                                'check the valid event for the process
            MsgBox "Event ID: " & tempstr & " not in Process ID: " & ProcessNo.Text
            Call Delete_Constraints(Mid(enforcedOrderstr, 1, Len(enforcedOrderstr) - 2))
            For i = 1 To MaxProcess     'delete enforced order from orderingconstraints
                If UCase(ProcessNo.Text) = BufferOut(i).process Then        'clear array
                    For j = 1 To maxlen
                        BufferOut(i).Orderevent(j).enforced = False
                    Next
                End If
            Next
            For i = 1 To enforcedcount          'init the enforced order array spaces
                EnforceEvent(i).element = ""
            Next
            enforcedcount = 0                   'init the controls variables
            enforcedorder.SetFocus
            enforcedOrderstr = ""
            result.Enabled = False
            Exit Sub
```

```
        Else                        'add the internal blocking labels to events
            Call Add_enforcedevent(tempstr)   'store the labels to the array
             If Trace_Enforced_Order = True Then
                                    'and verify the enforced order like visual order
                tempstr = ""
                tempforcestr = Mid(tempforcestr, n)
                m = 0                              'set counter variables
                n = 1
                For i = 1 To MaxProcess
                    If BufferOut(i).process = Trim(ProcessNo.Text) Then
                       BufferOut(i).forced = True 'mark for the process being enforced
                    End If
                Next
            Else            'if not valid order, reject the enforced order
                            'reset the size of array and clear the internal
                            'blocking labels.
            Call Delete_Constraints(Mid(enforcedOrderstr, 1,
        Len(enforcedOrderstr) - 2))  'delete enforced order from orderingcontraints
                result.Enabled := False
                enforcedOrderstr = ""
                For i = 1 To MaxProcess         'init/clear array
                    If BufferOut(i).process = Trim(ProcessNo.Text) Then
                        BufferOut(i).forced = False
                        For j = 1 To maxlen
                            For k = 1 To maxlen
                                BufferOut(i).Orderevent(j).token.intoken(k).element = ""
                                BufferOut(i).Orderevent(j).token.outtoken(k).element = ""
                            Next
                        Next
                    End If
                Next
            End If
        End If
    End If
 m = m + 1
Loop
nextprocess = True                 'reset controls
duplicated = False
Call Display_Enforced_order(Trim(ProcessNo.Text))
```

```
                                    'display the msc into ccs with enforced order
enforcedorder.Text = ""             'the processes
enforcedOrderstr = ""
ProcessNo.Text = ""
enforcedcount = 0
ProcessNo.SetFocus

End Sub


This procedure is to translate and display the whole MSC with
visual and enforced order into CCS.
-------------------------------------------------------------------------------------------

Private Sub Display_Enforced_order(currprocess)
Dim i, j, k, l, m, t, n, o, p As Integer                'declaration
Dim tget, tout As Boolean
Dim enforcedstack As String
Dim NoForced As Boolean

On Error Resume Next
                                                         'init variable
enforcedstack = "" If nextprocess = False Then Exit Sub
'set constraints If duplicated = True Then Exit Sub
EnforcedResultScreen.Text = "" NoForced = False
'the process didn't has enforced order For i = 1 To MaxProcess
'display the result of enforced order
        For j = 1 To maxlen
        If BufferOut(i).forced = False Then NoForced = True     'display the events
            If BufferOut(i).Orderevent(j).elt <> "" Then
                If Mid(BufferOut(i).Orderevent(j).elt, 1, 1) = "R" Then 'for receive event
                    For k = 1 To maxlen                 'display internal blocking labels
                        If BufferOut(i).Orderevent(j).token.intoken(k).element <> "" Then
                            EnforcedResultScreen.Text = EnforcedResultScreen.Text
                & BufferOut(i).Orderevent(j).token.intoken(k).element & "."
                        End If
                    Next                                'display buffering blocking labels
                    EnforcedResultScreen.Text = EnforcedResultScreen.Text &
                        BufferOut(i).Orderevent(j).signal & "."
```

```
      EnforcedResultScreen.Text = EnforcedResultScreen.Text &
          BufferOut(i).Orderevent(j).elt & "."
      For k = 1 To maxlen
          If BufferOut(i).Orderevent(j).token.outtoken(k).element <> "" Then
              EnforcedResultScreen.Text = EnforcedResultScreen.Text & "'" &
  BufferOut(i).Orderevent(j).token.outtoken(k).element & "."
          End If
      Next                                   'next event
  End If
  If Mid(BufferOut(i).Orderevent(j).elt, 1, 1) = "S" Then 'for send event
      For k = 1 To maxlen                    'display internal blocking labels
          If BufferOut(i).Orderevent(j).token.intoken(k).element <> "" Then
              EnforcedResultScreen.Text = EnforcedResultScreen.Text &
  BufferOut(i).Orderevent(j).token.intoken(k).element & "."
          End If
      Next 'input token                      'display buffering blocking labels
      EnforcedResultScreen.Text = EnforcedResultScreen.Text &
          BufferOut(i).Orderevent(j).elt & "."
      EnforcedResultScreen.Text = EnforcedResultScreen.Text & "'" &
          BufferOut(i).Orderevent(j).signal & "."
      For l = 1 To maxlen                    'display internal blocking labels
          If BufferOut(i).Orderevent(j).token.outtoken(l).element <> "" Then
              EnforcedResultScreen.Text = EnforcedResultScreen.Text & "'"
  & BufferOut(i).Orderevent(j).token.outtoken(l).element & "."
          End
      Next                                   'next event
  End If
  If j <> maxlen Then                        'if next process or not
      If BufferOut(i).Orderevent(j + 1).elt <> "" Then
          EnforcedResultScreen.Text = EnforcedResultScreen.Text & "nil" & "|"
      Else
          EnforcedResultScreen.Text = EnforcedResultScreen.Text & "nil"
      End If
  Else
      EnforcedResultScreen.Text = EnforcedResultScreen.Text & "nil"
  End If
 Next                                        'event elt
If NoForced = False Then                     'add enforcedstack token
  For m = 1 To maxlen                        'list the restricted labels
```

```
        If m <> maxlen Then 'check last label
            If BufferOut(i).Orderevent(m + 1).elt <> "" Then
                For t = 1 To maxlen
                    If BufferOut(i).Orderevent(m + 1).token.intoken(t).element
                        <> "" Then
                        enforcedstack = enforcedstack & "d" &
        Mid(BufferOut(i).Orderevent(m + 1).elt, 2) & ","
                    End If
                Next
            End If
        End If
    Next
End If

If Trim(enforcedstack) <> "" Then
    enforcedstack = GetTokenList(enforcedstack, maxlen)
End If                          'store internal blocking label for each process

Select Case i 'display the bracket   'for display brackets only
Case MaxProcess 'the last process
        If Trim(enforcedstack) <> "" Then
            EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")\{" &
Mid(enforcedstack, 1, Len(enforcedstack) - 1) & "})"
        Else
            EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")"
        End If
Case 1 'the first process
    If Trim(enforcedstack) = "" Then
        If BufferOut(i + 1).forced = True Then
            EnforcedResultScreen.Text = "(" & EnforcedResultScreen.Text & ")|(("
        Else
            EnforcedResultScreen.Text = "(" & EnforcedResultScreen.Text & ")|("
        End If
    Else
        If BufferOut(i + 1).forced = True Then
            EnforcedResultScreen.Text = "((" & EnforcedResultScreen.Text & ")\{"
& Mid(enforcedstack, 1, Len(enforcedstack) - 1) & "})|(("
        Else
           EnforcedResultScreen.Text = "((" & EnforcedResultScreen.Text & ")\{"
```

```
        & Mid(enforcedstack, 1, Len(enforcedstack) - 1) & "})|("
                End If
            End If
        Case Else 'other processes
            If Trim(enforcedstack) = "" Then
                If BufferOut(i + 1).forced = True Then
                    EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")|(("
                Else
                    EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")|("
                End If
            Else
                If BufferOut(i + 1).forced = True Then
                    EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")\{"
        & Mid(enforcedstack, 1, Len(enforcedstack) - 1) & "})|(("
                Else
                    EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")\{"
        & Mid(enforcedstack, 1, Len(enforcedstack) - 1) & "})|("
                End If
            End If
        End Select
    NoForced = False
    enforcedstack = ""
Next 'process id

If Trim(VisualOrderResultScreen.Text) = "" Then
    Call Trace_VisualOrder_Process
End If
EnforcedResultScreen.Text = EnforcedResultScreen.Text & "|("
o = 1
For p = 1 To maxsendbuffer                    'display each buffer's blocking
    Do Until o > 2                            'labels
        If Mid(buffevent(p).eventelt(o).eltout, 1, 1) = "c" Then
            EnforcedResultScreen.Text = EnforcedResultScreen.Text & "'"
    & buffevent(p).eventelt(o).eltout & "."
        Else
            EnforcedResultScreen.Text = EnforcedResultScreen.Text
    & buffevent(p).eventelt(o).eltout & "."
        End If
        o = o + 1
```

```
        Loop
        o = 1                                          'end of process
        If p < maxsendbuffer Then
            EnforcedResultScreen.Text = EnforcedResultScreen.Text & "nil" & "|"
        Else
            EnforcedResultScreen.Text = EnforcedResultScreen.Text & "nil"
        End If
    Next p
   EnforcedResultScreen.Text = EnforcedResultScreen.Text & ")"
    EnforcedResultScreen.Text = "(" & EnforcedResultScreen.Text & ")\{" 'display brackets
    t = 1
    For n = 1 To maxsendbuffer
        Do Until t > 2
            If Trim(buffevent(n).eventelt(t).eltout) <> "" Then
                EnforcedResultScreen.Text = EnforcedResultScreen.Text
        & buffevent(n).eventelt(t).eltout & ","
            End If
            t = t + 1
        Loop
        t = 1
    Next n                                             'brackets for end of display
    EnforcedResultScreen.Text = Mid(EnforcedResultScreen.Text, 1,
    Len(EnforcedResultScreen.Text)- 1) & "}"
    nextprocess = False                                'reset the controls
    duplicated = False
    result.Enabled = False

End Sub
---------------------------------------------------------------------------------------------
```

Appendix C

Using the tool output and CWB-NC to verify inferred orderings

In the examples we show the results of the Concurrency Workbench when it is given output from our tool for the input. We check to see if different inferred orderings are consistent with the visual or the enforced ordering of the MSC by using GCTL model checking feature in the CWB.

For the first example we use a simple visual-order MSC. The MSC is shown in Figure C.1

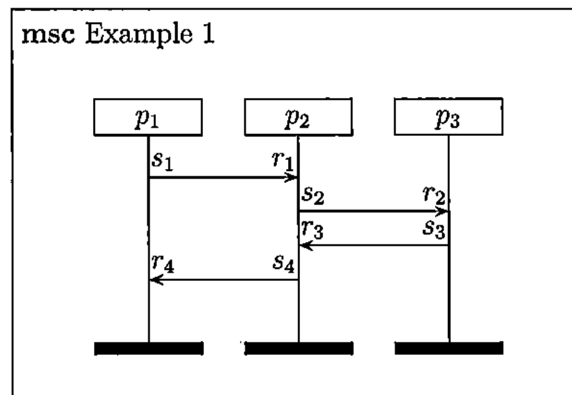


Figure C.1. *Example 1*

We stored the output from our tool in the file `MSC1Visual.ccs`[1]

[1]The output from the current version of the tool does not exactly meet the input format for

```
%cat MSC1Visual.ccs
proc m = (S1.'b1.c4.R4.nil |
          c1.R1.S2.'b2.c3.R3.S4.'b4.nil |
          c2.R2.S3.'b3.nil |
          b1.'c1.nil|b2.'c2.nil|b3.'c3.nil|b4.'c4.nil)
           \{b1,c1,b4,c4,b2,c2,b3,c3}
%cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load MSC1Visual.ccs
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.003)
```

The first check shows that the user can infer that a receive event always happens after the corresponding send event. Here it shows that r_1 happens after s_1 ($s_1 \sqsubset r_1$).

```
cwb-nc> chk -L gctl m A(~{R1}U{S1})
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 5 states.
Starting ABTA model checker.Table size: 0
State #0: Not -> 1 in {}
State #1: \/ -> {2/3} in {}
State #2: <R1> -> {4/} in {} (TREE)
```

CWB but it is very easy to put it in the right format

```
State #3: <<-S1>> -> {/1} in {}
State #4: True in {} (TREE)

Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.009)
```

Another inferred order that we always get is every one in a local process that is consistent with the visual order. Here it shows that $r_1 \sqsubset s_2$.

```
cwb-nc> chk -L gctl m A(~{S2}U{R1})

...

Model checking completed.
Expanded state-space 13 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.008)
```

The next example shows that $\sqsubset$ is transitive. Since we showed that $s_1 \sqsubset r_1$ and $r_1 \sqsubset s_2$, and we could also show $s_2 \sqsubset r_2$, we can show $s_1 \sqsubset r_2$.

```
cwb-nc> chk -L gctl m A(~{R2}U{S1})

...

Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.006)
```

The following example also shows transitivity for $\sqsubset$. The next example shows that from $r_2 \sqsubset r_4$ it doesn't always follow that $r_4 \sqsubset r_2$.

```
cwb-nc> chk -L gctl m A(~{R4}U{R2})

...

Model checking completed.
Expanded state-space 25 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.009)

cwb-nc> chk -L gctl A(~{R2}U{R4})

...

Model checking completed.
Expanded state-space 25 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.000,0.010,0.000,0.008)
```

We can again take the MSC from Figure C.1 but now we only have enforced order for sends that are blocked by receives. This enforced order is $r_1 << s_2$, $r_2 << s_3$ and $r_3 << s_4$. We translate this enforced order MSC with our tool and copy the result to the file MSC1Enforced.ccs.

```
%cat MSC1Enforced.ccs
proc m = ((S1.'b1.nil|c4.R4.nil) |
          ((c1.R1.'d2.nil|d2.S2.'b2.nil|c3.R3.'d4.nil|d4.S4.'b4.nil)\{d2,d4}) |
          ((c2.R2.'d3.nil|d3.S3.'b3.nil)\{d3}) |
          (b1.'c1.nil|b2.'c2.nil|b4.'c4.nil|b3.'c3.nil))
              \{b1,c1,b2,c2,b4,c4,b3,c3}
```

In this example, the user may believe that $s_1 \sqsubset r_4$ and $s_2 \sqsubset r_3$ even though it is not true that $s_1 << r_4$ and $s_2 << r_3$ are part of the enforced order. We can check this with CWB.

```
cwb-nc> load MSC1Enforced.ccs
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.004)
cwb-nc> chk -L gctl m A(~{R4}U{S1})

...

Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.000,0.010,0.000,0.008)

cwb-nc> chk -L gctl m A(~{R3}U{S2})

...

Model checking completed.
Expanded state-space 19 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.000,0.010,0.000,0.009)
```

For the second example we use another simple visual-order MSC. The MSC is shown in Figure C.2

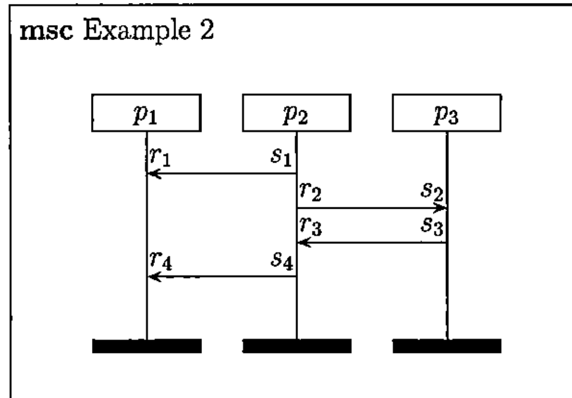


Figure C.2. *Example 2*

We take the MSC from Figure C.2 with the enforced order $r_1 \ll r_4$. We translate this enforced order MSC with our tool and copy the result to the file MSC2Enforced.ccs.

```
%cat MSC2Enforced.ccs
proc t =(((c1.R1.'d4.nil|d4.c4.R4.nil)\{d4})|
         (S1.'b1.nil|S2.'b2.nil|c3.R3.nil|S4.'b4.nil)|
         (c2.R2.nil|S3.'b3.nil)|
         (b1.'c1.nil|b2.'c2.nil|b4.'c4.nil|b3.'c3.nil))
              \{b1,c1,b2,c2,b4,c4,b3,c3}

> cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)


cwb-nc> load MSC2Enforced.ccs
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.012)
```

The enforced order $r_1 \ll r_4$ of the receives does not impose an ordering on the corresponding sends s_1 and s_4, if assume asynchronous message delivery without any guarantee of timeliness. We can check that $s_1 \sqsubset s_4$ does not hold:

```
cwb-nc> chk -L gctl t A(~{S4}U{S1})

...

Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.006)
```

We can again take the MSC from Figure C.2 with the enforced order $r_2 \ll s_3$. We translate this enforced order MSC with our tool and copy the result to the file MSC3Enforced.ccs.

```
%cat MSC3Enforced.ccs
proc t=((c_1.R_1.nil|c_4.R_4.nil)|
        (S_1.'b_1.nil|S_2.'b_2.nil|c_3.R_3.nil|S_4.'b_4.nil)|
        ((c_2.R_2.'d_3.nil|d_3.S_3.'b_3.nil)\{d_3})|
        (b_1.'c_1.nil|b_2.'c_2.nil|b_4.'c_4.nil|b_3.'c_3.nil))
           \{b_1,c_1,b_2,c_2,b_4,c_4,b_3,c_3}
> cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load MSC3Enforced.ccs
Execution tim (user,system,gc,real):(0.010,0.000,0.000,0.012)
```

Now $r_2 \ll s_3$, so we can show that $s_2 \sqsubset r_3$.

```
cwb-nc> chk -L gctl t A(~{R3}U{S2})

 ..

Model checking completed.
```

```
Expanded state-space 76 times.
Stored 57 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.020,0.000,0.000,0.023)
```

The next example shows that even though $R_2 << S_3$, it doesn't show that $s_1 \sqsubset r_3$.

```
cwb-nc> chk -L gctl t A(~{R3}U{S1})

...

Model checking completed.
Expanded state-space 37 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.000,0.020,0.000,0.016)
```

For the last example we use another visual-order MSC. The MSC is shown in Figure C.3

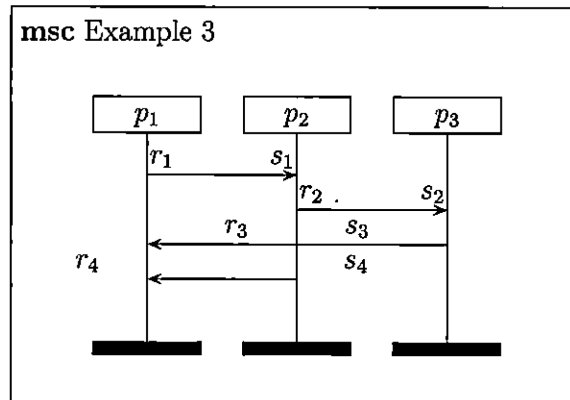


Figure C.3. *Example 3*

We take the MSC from Figure C.3 with the enforced order $S_1 << S_4$.We translate this enforced order MSC with our tool and copy the result to the file `MSC4Enforce.ccs`.

```
%cat MSC4Enforced.ccs
proc n = (((S_1.'b_1.'d_4.nil|c_3.R_3.nil|d_4.S_4.'b_4.nil)\{d_4})|
         (c_1.R_1.nil|S_2.'b_2.nil|c_4.R_4.nil)|(c_2.R_2.nil|S_3.'b_3.nil)|
         (b_1.'c_1.nil|b_4.'c_4.nil|b_2.'c_2.nil|b_3.'c_3.nil))
         \{b_1,c_1,b_4,c_4,b_2,c_2,b_3,c_3}
> cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)


cwb-nc> load MSC4Enforced.ccs
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.012)
```

Here we cannot show that $r_1 \sqsubset r_4$.

```
cwb-nc> chk -L gctl n A(~{R4}U{R1})

...
```

```
Model checking completed.
Expanded state-space 49 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.010,0.010,0.000,0.019)
```

We can again take the MSC from Figure C.3 with the enforced order $R1 \ll S_2 < < R_4$.We translate this enforced order MSC with our tool and copy the result to the file MSC5Enforced.ccs.

```
%cat MSC5Enforced.ccs
proc n = ((S1.'b1.nil|c3.R3.nil|S4.'b4.nil)|
          ((c1.R1.'d2.'d4.nil|d2.S2.'b2.'d4.nil|d4.d4.c4.R4.nil)\{d2,d4})|
          (c2.R2.nil|S3.'b3.nil)|
          (b1.'c1.nil|b4.'c4.nil|b2.'c2.nil|b3.'c3.nil))
             \{b1,c1,b4,c4,b2,c2,b3,c3}
> cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)


cwb-nc> load MSC5Enforced.ccs
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.012)
```

This example shows that we can infer the ordering $s_1 \sqsubset r_4$.

```
cwb-nc> chk -L gctl n A(~{R4}U{S1})

...

Model checking completed.
Expanded state-space 37 times.
Stored 25 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.030,0.000,0.010,0.025)
```

We can again take the MSC from Figure C.3 with the enforced order $S1 \ll R3$ and $R1 \ll S2 \ll R4$.We translate this enforced order MSC with our tool and copy the result to the file MSC6Enforce.ccs.

```
%cat MSC6Enforced.ccs
proc n = (((S1.'b1.'d3.nil|c3.d3.R3.nil|S4.'b4.nil)\{d3})|
          ((c1.R1.'d2.'d4.nil|d2.S2.'b2.'d4.nil|d4.d4.c4.R4.nil)\{d2,d4})|
          (c2.R2.nil|S3.'b3.nil)|(b1.'c1.nil|b4.'c4.nil|b2.'c2.nil|b3.'c3.nil))
          \{b1,c1,b4,c4,b2,c2,b3,c3}
> cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load MSC6Enforced.ccs
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.012)
```

Now we cannot show that $s_1 \sqsubset s_4$.

```
cwb-nc> chk -L gctl n A(~{S4}U{S1})

...

Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.008)
```

Again, we cannot show that $r_2 \sqsubset s_3$.

```
cwb-nc> chk -L gctl n A(~{S3}U{R2})

...

Model checking completed.
```

```
Expanded state-space 4 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.000,0.010,0.000,0.007)
```

Index

mu-calculus, 34

An Interpreted Message Sequence Charts, 9

atomic action propositions, 35

Blocking Labels, 25

Buffering, 13

CCS, 16
- Agent, 16, 19
- Agent Expression, 16
- Alternation, 16
- Composition, 17, 20
- Expansion Law, 32
- LTS, 18
- Nondeterministic, 17
- Prefix, 16, 19
- Relabelling, 21
- Renaming, 17, 18
- Restriction, 17, 20
- Selection, 19
- Silent Action, 20
- Silent action, 16
- Transition Rules, 17

Concurrency Workbench, 2

CTL, 34

GCTL, 34, 37
- temporal operators, 35

Implementation, 2

Message Sequence Charts, 2, 7, 8, 13, 14
- Enforced Ordering, 2, 9
- Inferred Ordering, 3, 9
- Local Ordering, 9
- Total Ordering, 8
- Visual Ordering, 2, 7, 8

Modeling, 1

Path formulas, 34, 35

Realizability, 6

Specification, 1

State formulas, 35

Temporal logic, 33
- Branching time, 33
- Linear time, 33

Testing, 2

Verification, 1

VITA

Surname: Chiu | Given Names: Wai Han

Place of Birth: Macau

Educational Institutions Attended:

University of Victoria, BC	1993 to 1995
University of Victoria, BC	1998 to 2003

Degrees Awarded:

B.Sc. (Computer Science)	University of Victoria	1995

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Modeling and Verification of Message Sequence Charts using Process Algebras and Temporal Logic Model Checking

Author:

[illegible] Aug 2003

Date