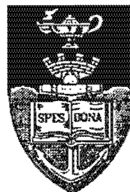
BISIMULATION AS A VERIFICATION AND VALIDATION TECHNIQUE FOR MESSAGE SEQUENCE CHARTS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Philip Wall
April 1998

Supervised by
Prof P.S. Kritzinger



University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Abstract

The complexity of determining whether a system meets the requirements of its designers has increased with the widespread use of real time concurrent systems. This testing process has however been simplified with the emergence of Formal Description Techniques. FDTs not only provide the means for formally specifying a system, but also supply the theoretical basis for conformance testing.

One such FDT is Message Sequence Charts(MSCs). MSCs have evolved out of the need to describe the inter-process flow of communication in a concise, easily understood, graphical format. MSCs originally took the form of system traces, but with the development of, and additions to the specification, the 1996 MSC specification now provides a comprehensive description technique.

An FDT is of little use, unless it can be used in the validation and verification of a system specification. In the case of MSCs, this has usually involved the testing of a trace against a specification. Formal specification with MSCs has however provided the opportunity of testing the equivalence of MSC specifications.

Two equivalence techniques present themselves, namely linear and branch time equivalence. Linear time equivalence provides the assurance that the systems compared give rise to the same event sequences, but with a time complexity of $O(n^3)$ for n states. Branch time equivalence or bisimulation testing, applies a more stringent notion of equivalence, which might exclude systems that are linear time equivalent. Branch time equivalence can however be determined in $O(mlogn)$ for n states and m transitions. It is therefore desirable to make use of this notion of bisimulation wherever possible when deciding equivalence.

This thesis investigates the feasibility of implementing bisimulation testing on the 1996 MSC recommendation. Each of the MSC components is discussed with the intention of converting the primarily graphical representation of MSCs to the finite state form required by bisimulation testing. A state space explosion is an unavoidable consequence of this conversion and is a limiting factor on bisimulation testing of large systems.

University of Cape Town

The feasibility of bisimulation testing in the context of this thesis is illustrated through a tool which allows the specification and implementation of both bisimulation and linear time equivalence testing on systems consisting of the high level MSC structures and basic MSC event components. An example generated and tested by this tool is provided.

University of Cape Town

Acknowledgements

I wish to thank all the people who through encouragement, suggestion and constructive criticism provided me with assistance in the completion of this thesis.

Special thanks to

- My supervisor, Professor Pieter S. Kritzinger.
- Fellow MSc students and the procession of visiting Germans.
- Staff and Lecturers in the Mathematics and Computer Science departments at UCT.

University of Cape Town

Contents

Abstract **ii**

Acknowledgements **iv**

1 Background **1**

1.1 Introduction 1

1.2 Concurrent Systems 1

1.3 System Specification 2

1.4 Modelling Real Time Concurrent Systems 2

1.5 Choice of Model 3

1.6 Verification and Validation 3

1.6.1 Correctness Testing 4

1.6.2 Equivalence of Models 4

1.7 Formal Description Techniques 5

1.7.1 Specification and Description Language (SDL) 5

1.7.2 Temporal Ordering Specification Language (LOTOS) 5

1.7.3 Extended State Transition Language (Estelle) 6

1.7.4 Message Sequence Charts(MSC) 6

1.8 Tools 6

1.8.1 ObjectGeode 6

University of Cape Town

1.8.2 Spin . . . 7
1.8.3 Caesar/Aldebaran Development Package(CADP) . . . 7
1.8.4 Estelle Development Toolset(EDT) . . . 8
1.8.5 Conclusion . . . 8

2 Message Sequence Charts **9**
2.1 Introduction . . . 9
2.2 History . . . 10
2.3 Uses in system engineering . . . 11
2.4 Examples of tests on MSCs . . . 11
2.5 Representations . . . 12
2.5.1 Textual . . . 12
2.5.2 Graphical . . . 12
2.6 Description of Language . . . 13
2.6.1 Basic MSC concepts . . . 13
2.6.2 Structural Constructs . . . 16
2.7 Semantic Requirements . . . 20
2.8 Conclusion . . . 21

3 Bisimulation **22**
3.1 Introduction . . . 22
3.2 Linear and Branch Time Equivalence . . . 22
3.3 Abstraction . . . 23
3.4 Bisimulation . . . 24
3.4.1 Formal Definition of Processes . . . 24
3.4.2 Equivalence types . . . 25
3.4.3 Complexity of determining equivalence . . . 26

University of Cape Town

4 Bisimulation on Message Sequence Charts **28**

4.1 Introduction . . . 28

4.2 An Equivalence for MSCs . . . 29

4.3 Applying Strong Observational Equivalence to MSCs . . . 29

4.4 Notation used by grammar . . . 30

4.5 Message Sequence Chart Document . . . 31

4.6 Basic Message Sequence Chart . . . 32

4.6.1 Message Sequence Chart . . . 32

4.6.2 Instance . . . 32

4.6.3 Message . . . 33

4.6.4 Environment . . . 33

4.6.5 General Ordering . . . 33

4.6.6 Condition . . . 34

4.6.7 Timer . . . 34

4.6.8 Action . . . 34

4.6.9 Instance Creation . . . 35

4.6.10 Instance Stop . . . 35

4.7 Structural Concepts . . . 35

4.7.1 Coregion . . . 35

4.7.2 Instance Decomposition . . . 36

4.7.3 Inline Expression . . . 36

4.7.4 MSC reference . . . 39

4.7.5 High-Level MSC . . . 39

4.8 Optimisations . . . 40

University of Cape Town

5 Bisimulation Algorithms **41**

5.1 Introduction 41

5.2 Algorithm Goal 41

5.3 Relational Coarsest Partitioning 41

5.3.1 Implementation of an O(mn) algorithm 43

5.3.2 Implementation of O(mlogn) algorithm 44

5.4 Trace Equivalence using Bisimulation 46

6 Implementation **47**

6.1 Introduction 47

6.2 Background to Implementation 47

6.3 Displaying an MSC 48

6.4 Editing MSCs 49

6.4.1 Creating an Instance 50

6.4.2 Message Events 50

6.4.3 Instance Creation Event 50

6.4.4 Timer Statements 50

6.4.5 Action Event 52

6.4.6 Instance End and Stop events 52

6.4.7 Shared Conditions 52

6.4.8 Coregions 54

6.4.9 High Level Structures 54

6.5 Bisimulation Testing 54

6.6 Printing 54

6.7 Summary 54

University of Cape Town

7 Examples 56

7.1 Introduction . . . 56

7.2 Example 1 . . . 56

7.2.1 Problem Description . . . 56

7.2.2 Conversion to Finite State Machine . . . 57

7.2.3 Bisimulation Algorithm . . . 57

7.2.4 Refinement Table . . . 61

7.2.5 Results . . . 62

7.3 Example 2 . . . 63

7.3.1 Problem description . . . 63

7.3.2 Conversion to Finite State Machine . . . 63

7.3.3 Bisimulation Algorithm . . . 63

7.3.4 Refinement Table . . . 63

7.3.5 Results . . . 78

8 Conclusion 79

8.1 Introduction . . . 79

8.2 Concerns . . . 79

8.3 MSC and System design . . . 80

8.4 MSCs and Bisimulation . . . 80

8.5 Extensions and Changes . . . 81

8.5.1 Editor . . . 81

8.5.2 Finite State Machine Conversion . . . 81

8.6 Summary . . . 82

Bibliography 83

A Encoded Bisimulation Algorithm 86

University of Cape Town

Chapter 1

Background

1.1 Introduction

Determining whether an implementation of a real time concurrent system meets its specification is a complex task. A problem facing such testing is finding a balance between the detail of the system and a representation that carries enough information to fully describe the system without obscuring other aspects that might be of interest.

Numerous formal description techniques, each emphasising different aspects of a system, have emerged. One such method, concentrating on inter-process communication is Message Sequence Charts. Both graphical and textual representations of MSCs exist. The intuitive nature of system specifications using MSCs has resulted in the use of MSCs across the full spectrum of system engineering, including design, specification and testing.

MSCs are however not a dominant description technique and to understand why, it is important to understand how it fits in with other formal description techniques.

MSCs can be placed in context by examining the factors influencing specification testing, the specification techniques in use and finally how implementations make use of these specifications to automate testing.

1.2 Concurrent Systems

The ability to distribute computational load between processing units has resulted in an increase in the complexity of system design and implementation. Systems making use of

University of Cape Town

distributed processing are known as concurrent systems and can be contrasted with sequential systems, where instructions are executed consecutively. Real time concurrent systems (RTCS) occur in medical, telecommunication and industrial control systems. [MA90]

RTCS often have strict constraints on response times. This has given rise to two system implementations, namely synchronous and asynchronous.

Synchronous systems are clock driven. Processor time is divided into segments, and all system computation is divided to fill these segments. Synchronous systems can guarantee response times.

Asynchronous systems are interrupt driven. After each block of computation completes, the work distributor or scheduler is called for the next computation block. Asynchronous systems have the advantage that no time is wasted if the scheduler has work to be done. Their major disadvantage is the difficulty in constructing scheduling tables.

1.3 System Specification

System specification is a concise description of the behaviour and properties of a system.

System specification can take place at different levels of abstraction. These range from natural language descriptions of problems to the level where system components are so detailed, that implementation can occur.

The greatest difficulty in the specification process is to fully understand the requirements of the system under development. The requirements are the services and constraints the system must operate under. These requirements should be comprehensively documented to ensure compliance to the specification at later stages of development. By describing the requirements using a formal specification language, this compliance can then be tested.

System specification is an incremental process. It is not possible to create a definitive specification for a complex system at the start of the design process. This is because it is often only realised at later stages exactly what the requirements are. The specification technique should therefore lend itself to being constantly updated.

1.4 Modelling Real Time Concurrent Systems

Many modelling techniques exist to simplify the specification process. These techniques attempt to describe real-world concepts in terms of an abstraction. The abstraction is most

University of Cape Town

effective if it has a formal definition.

Finite state machines (FSM) are often used to model RTCS because of their ability to describe a system in terms of states and state transitions [J.H69]. Typically one would examine the inputs, outputs and changes in the system.

Additional constructs have been created to increase the descriptive powers of FSM. This has resulted in a profusion of FSM based modelling techniques. Examples include Petri Nets[Pet81], labelled transition systems, Hoare's Traces[S.S], Synchronization Trees[Rei92], and Pomsets[W.C81].

These models all allow a level of analysis to be done on the system. The drawback of any FSM method is that as complexity increases, a state space explosion occurs[Fer90].

1.5 Choice of Model

The choice of specification model is determined by a number of factors, mostly concerned with how we view the system and at what level of abstraction the specification is aimed. [S.S]

In deciding on a choice of model, the following should be taken into account

- Implicit versus explicit description of system states and state changes.
- Interleaving of instructions versus true concurrent execution of instructions
- Importance of the decision making point, the problem of "Branch time" versus "Linear time" decision making.

Clearly, the most useful modelling approach is the one that makes use of multiple description techniques.

1.6 Verification and Validation

A system that has gone through the entire design cycle can be tested against its specification through the twin processes of verification and validation. Verification tests the conformance of the system to its specification, whereas validation checks whether the program implementation meets the requirements of the user.

University of Cape Town

Applying formal verification techniques to RTCS is made difficult by a number of factors. These include the interaction occurring between processes and the possibility of errors being time dependent.

Often, the only testing done occurs after implementation. Testing for unexpected behaviour during a program's execution, runs the risk that not all errors will be found. Testing by itself can never prove a program's correctness. [I.S89]

Other methods are therefore required to guarantee the discovery and removal of errors and bugs in a program. Two such methods involve the analysis and formal verification of the specification.

1.6.1 Correctness Testing

Much research has gone into the theory of testing RTCS for correctness[MA90]. Determining the correctness of a sequential system is clearly less complex than for a RTCS. The correctness of a sequential terminating system is proved by showing that the program produces the correct output and terminates. For non-terminating systems, which include most RTCS, correctness testing becomes a more involved process. Concepts such as mutual exclusion, deadlock, livelock, starvation and fairness have to be modelled and investigated. This places additional requirements on the modelling technique.

Correctness testing in concert with run time testing can prove very effective in the verification and validation of a system.

University of Cape Town

1.6.2 Equivalence of Models

Another area of investigation is the comparison of systems by means of equivalence testing. This is most meaningful if one model originates from the requirement specification stage and the other model is generated by some means from the implementation. The implementation can then be verified against the requirement description for compliance.

As mentioned earlier, the complexity of RTCS can give rise to large models. These models can either undergo some form of state reduction or depending on their definition, be compared piecewise. Ways of reducing the size of a model include restricting the level of detail or by hiding irrelevant actions.

The many modelling techniques have given rise to many forms of equivalence. Of particular interest to this study are trace equivalence and bisimulation equivalence. Trace equivalence

is used on linear time models and bisimulation testing applies to branch time models. Models equivalent under linear time equivalence exhibit the same execution sequences whereas under branch time equivalence, the branching structure is also considered.

1.7 Formal Description Techniques

The purpose of Formal Description Techniques(FDT) is to provide a specification of a service, protocol or interface. By formalising the description, specifications can be analysed for correctness and efficiency. Four popular FDTs are listed below.

1.7.1 Specification and Description Language (SDL)

University of Cape Town

SDL is an ITU-standardised language, widely used in the telecommunication field [IT88]. This FDT allows one to state what a system should do and can also describe how it does it. Systems are specified according to their structure and behaviour as communicating extended state machines.

The internal structure of a system can be described in an object oriented approach, allowing the system to be developed modularly. This allows redesign of modules while limiting the impact of the changes on other areas of the specification.

Various levels of abstraction are addressed, from overview to detailed design. Although not intended as an implementation language, enough detail can be supplied by an SDL specification to code an implementation.

Testing of SDL specifications is made possible by placing system components in a test environment. This allows components to be tested as they are developed, without requiring the full specification to be available. Testing the full system and the interaction of components must be done later.

Documentation can be included with an SDL specification. This allows version control. After each modification, the entire specification should ideally be retested.

1.7.2 Temporal Ordering Specification Language (LOTOS)

LOTOS uses a process algebra as the basis for system specification. Processes are defined in terms of their ability to communicate with their surroundings. The internal workings of the process itself are not part of the model.

Operators are used to specify temporal orderings between actions and entities. Various sequence of actions can be constructed from such expressions. Complexity in the construction of these expressions often determine that sequences of expressions are not explicitly shown.

Expressions allow components to be reused and provide a natural structuring construct. [G.L92]

1.7.3 Extended State Transition Language (Estelle)

Similar to SDL, Estelle is based on an extended state transition model. The model is augmented by means of the Pascal language.

Systems specified using Estelle consist of interacting modules, represented by a hierarchical structure of communicating automata. Interaction between modules occurs in the form of parameters and variables[S.B87].

1.7.4 Message Sequence Charts(MSC)

MSC describes systems by means of execution traces. A graphical description method exists to describe communication between processes. Compositional operators allow the structuring of traces into structures. MSC are often used together with other FDT.

Although the 1996 MSC recommendation is not in general commercial use, further references to "MSC" can be take to include this recommendation. A comprehensive discussion of MSCs can be found in the next chapter.

University of Cape Town

1.8 Tools

An FDT without the appropriate software tools will remain no more than a concept. Several tools have been developed. The tools illustrate how the specification techniques mentioned in the previous section can be used.

1.8.1 ObjectGeode

ObjectGeode[V.E93] is a graphical toolset allowing verification and validation through simulation, code generation and testing. It provides one of the most comprehensive toolsets

for system specification and description. Systems can be specified through the SDL, Object Modelling Techniques(OMT) and Message Sequence Chart standards.

These techniques can be integrated to provide a unified system description.

Analysis is done by means of a code generator or simulator. The code generator converts an SDL specification to C/C++. Errors such as deadlocks, live locks or dead code can be found.

A simulator allows for verification and validation of SDL specifications. Control of the simulator can either be through a graphical interface or textual commands.

The simulator has three modes, namely random, interactive and exhaustive. Sequences giving rise to errors can be saved as Message Sequence Chart traces for later viewing.

The executable generated by the code generator can be graphically debugged at the SDL level in the execution environment.

Various other tools exist that provide the same basic features. One of these is SDT, which integrates Message Sequence Chart and SDL with the software management and maintenance process [Tel98].

1.8.2 Spin

Spin [D.O95][G.H] provides the means to analyse the logical consistency of a system described in Promela. Promela is a verification modelling language and lacks complex constructs such as abstract data types and also makes no attempt to describe details unrelated to process interaction. It does however provide a comprehensive way of describing the synchronous and asynchronous communication between processes.

A system specified in Promela is analysed either by random simulations or by the generation of a C program when testing correctness properties. Some properties that can be investigated include deadlock, unspecified execution, unexecuted code and system invariants.

Spin overcomes the state space problem using a method known as 'Supertrace' to collapse the state space and thereby reducing the states space.

1.8.3 Caesar/Aldebaran Development Package(CADP)

CADP[lab98] was developed for protocol engineering. Initially systems were described either in LOTOS or labelled transition systems, but subsequent extensions to the initial project allow the CADP tool-set to be used with protocols specified using SDL.

University of Cape Town

The CADP tool-set translates the LOTOS specification to C or a labelled transition system. Analysis is then performed by means of random simulation, testing or verification. Correctness properties and tests include partial and exhaustive deadlock detection, test sequence generation, verification of behavioural specification using bisimulation(equivalence) testing and the verification of branch time temporal logic specification.

Optimisations once again involve reducing the state space generated for the label transition system. Some of the methods used include avoiding the generation of the whole labelled transition system, binary decision diagrams and using a compacted form of storage.

1.8.4 Estelle Development Toolset(EDT)

This toolset for Estelle[S.B] consists of a compiler, simulator, debugger, test driver generator and state table generator.

The compiler serves the same purpose as the ObjectGEODE compiler. It translates a specification into C. The executable can be controlled by a graphical interface or be command driven.

The simulator allows a compiled specification to be checked for errors occurring during execution. The simulator allows the user to guide which properties are observed.

1.8.5 Conclusion

All these toolsets effectively achieve the same purpose, namely a method for the creation and display of a specification and an environment testing.

No equivalent toolset exists solely for MSCs. At present, MSCs are most often used as an extension to other FDTs and are therefore not being used to their full potential. This project sets out to provide an environment for system specification with MSCs and the beginnings of a toolset to encourage the use of MSCs.

Validation of a system using an MSC specification has been neglected by other tools due to the complexity of systems generated by FDTS. This toolset therefore seeks to provide the ability to validate an MSC against a specification by means of bisimulation.

University of Cape Town

Chapter 2

Message Sequence Charts

2.1 Introduction

An effective formal description technique (FDT) is one where a system description is easily understood and can be analysed to provide meaningful information about the system. A balance is therefore needed between the amount of information provided about the system being specified by and FDT and its presentation.

Message Sequence Charts (MSCs) [IT93] provide an FDT concentrating on the inter-change of messages between processes. In essence, an MSC is a visualisation of the communication occurring during system executions. It can therefore be considered distinct from FDTs such as SDL, which emphasise the computation occurring inside a process rather than interaction between processes.

A limiting factor of FDTs is their manner of presentation. For analysis, the more mathematical the modelling technique the easier it is to draw conclusions about the correctness of the system, but a mathematical model is however not always easily understood.

A graphical form of presentation is therefore important to any FDT intended for general use. MSCs can be specified by means of a graphical or textual format. The importance of simplicity has been stressed throughout the development of MSCs, resulting in a limited number of constructs and events.

University of Cape Town

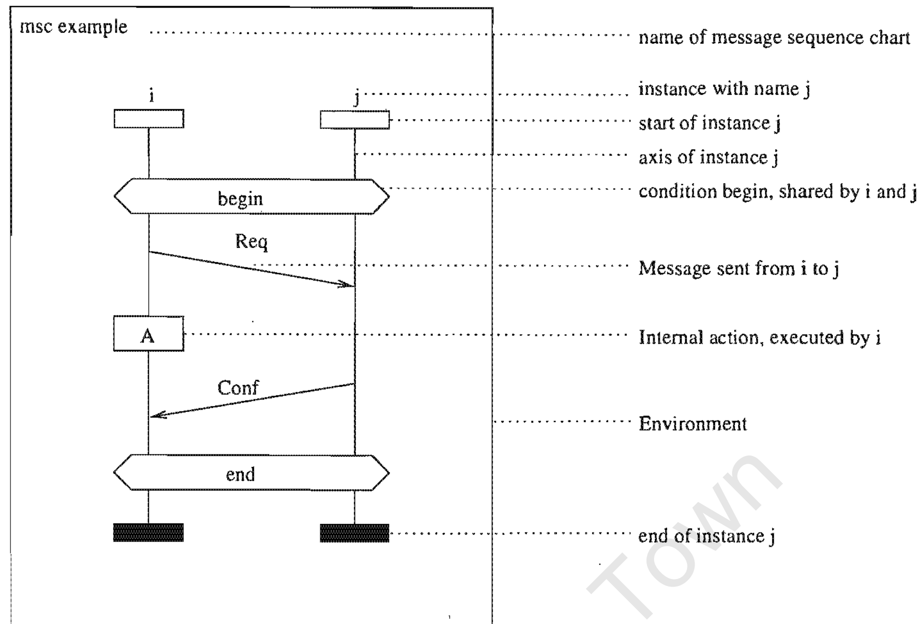


Figure 1: Example of Message Sequence Chart

2.2 History

MSCs started out under various guises in system engineering. Among these are arrow diagrams, message flow graphs and sequence diagrams. These all arose from the need for a finite way of representing communication traces of possibly infinite length. Such traces are commonly generated by systems in the telecommunication industry.

Various companies such as NFT-Ericsson and Siemens, made use of these representations, with each company making use of its own variation. Companies soon realised that a standardised representation would be to their advantage. It would allow the exchange of MSCs between companies, the integration of MSC with other FDTs, remove ambiguity and aid the development of tools for design and analysis.

Standardisation was first suggested in 1989. By the beginning of 1992, the MSC recommendation was approved as MSC recommendation Z.120. A study group within the ITU was created to handle modification and extensions to the standard.

Various approaches were investigated before deciding on a formal semantics for MSC. These included automata, petri nets and process algebra. A process algebra approach was decided on because of the resources available to the relevant study group.

University of Cape Town

2.3 Uses in system engineering

The simplicity and ease of use of MSCs has prompted their use in virtually all aspects of the system engineering process. This has been further bolstered by the standardisation. Some of the areas where MSCs are used include:

- Overview of services offered by several entities
- Requirements Specification
- The generation of a framework for SDL specifications
- Providing test cases for testing specifications
- Object oriented design and analysis
- Use cases
- Specification of communication
- Interface specification
- Documentation
- Trace generation for simulation and validation

University of Cape Town

2.4 Examples of tests on MSCs

A process algebra notation was adopted to specify the formal semantics of MSCs[S.M96]. This has led to the application of process algebra theory in areas such as correctness testing to MSCs.

An MSC is by nature fairly low level. This implies that a particular MSC covers only part of the behaviour of a system. Choosing what behaviour is covered will often determine the effectiveness of the specification. An in depth knowledge of a system and its specification is therefore required before testing.

Tests done on MSCs can take various forms. Some simple tests include:

- Checking whether the behaviour of an MSC is contained in another. This enables one to check systems for desired as well as unwanted behaviour.

- Testing of the requirements document. If the requirements document was created using MSC, it becomes useful to generate MSC traces by simulation of the implemented system. These traces can then be compared with this document.
- Equivalence testing. Such a test could be used to determine whether two MSC specifications exhibit the same behaviour. This differs from requirements testing in that systems are compared, rather than trace comparison of a system.

Although useful on their own, MSCs are most effective when used in combination with formal description techniques such as SDL. Ideally we wish to be able to specify inter-process communication and describe the internal behaviour of a process, allowing one the ability to draw conclusions about the correctness and consistency of the system.

University of Cape Town

2.5 Representations

The representation of a FDT is arguably the most important aspect. FDT with a simple representation will often find more use than a complex mathematical FDT. MSCs have two representations, namely textual and graphical.

2.5.1 Textual

The function of the textual representation is to unambiguously describe a specification. Although not as easy to use as the graphical representation, it is easier to incorporate into tools involved with analysis. This is because the transformation and manipulation of textual representations are easier.

A BNF grammar is supplied by Z.120 Recommendation [IT93] for the textual syntax. Although this grammar ensures syntactic compliance, semantic rules also need to be applied. The automation of general semantic checking has only become feasible since the standardisation.

2.5.2 Graphical

The graphical representation is the format most commonly used by MSCs. Similar to the textual syntax, the graphical syntax is defined by a BNF grammar. An MSC in graphical representation can be transformed automatically into its textual syntax.

The greatest problem facing the grammar is the creation of a visual chart from a graphical syntax, which is by its nature rule based. A solution that has been implemented by some companies is the placing of layout information in comments.

2.6 Description of Language

A general flavour of MSC96 as a specification technique is provided to introduce the reader to MSCs. We define the structures and events to be found in an MSC[EJ96][S.M96].

2.6.1 Basic MSC concepts

An MSC document consists of multiple charts combined using structural constructs. In any chart, the following might be found:

Environment

The system environment is represented by an enclosing rectangle. All other information associated with an MSC must occur inside this rectangle.

Instances

A chart consists of entities known as instances. These may be equated to blocks, processes or systems as found in SDL. Each instance has a beginning and end. An instance is represented by two rectangles joined by a vertical line. The vertical line is a timeline, providing a total ordering on the events occurring during the lifetime of the instance. The top rectangle indicates the start of an instance, and the bottom rectangle its end.

Events on instances occur in parallel and act independently of each other.

Messages

Messages provide the basic events of an MSC. Each message has an input and output, a source and a destination. Choices of destination include the environment and other instances. Messages bring about interaction between instances. As such, they require the additional proviso that a message can not be received by an instance until it has been sent.

University of Cape Town

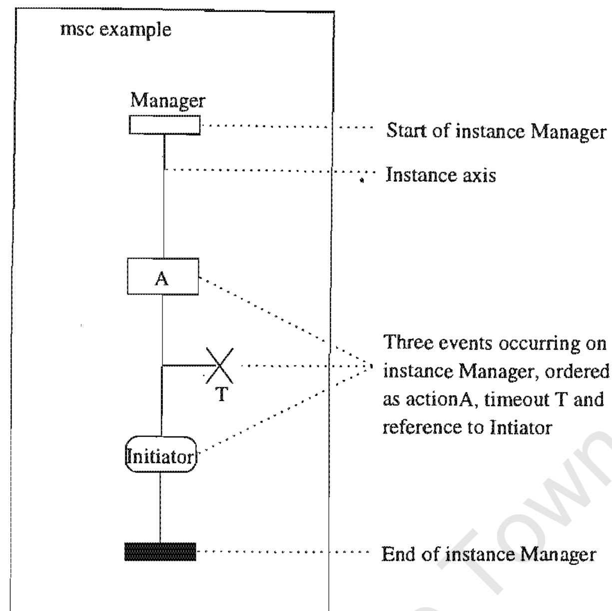


Figure 2: Single Instance with Ordered Events

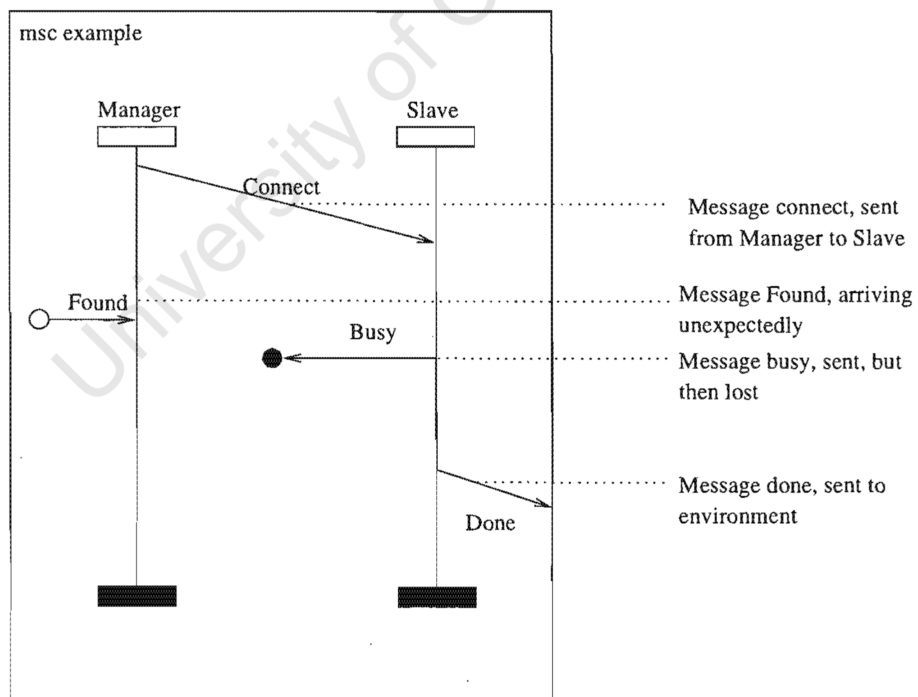


Figure 3: Examples of Message Types

University of Cape Town

Conditions

Conditions in MSCs serve a number of purposes. They can be used for controlling the sequential composition of MSCs or they can also be used to describe system states. Conditions can extend over a limited number of instances, or all instances.

The symbol for a condition is a hexagon which extends over instances involved.

Timers

Three events exist to manipulate the timer. Timers can be set, reset and experience a timeout. The symbol for setting a timer is an hourglass joined to an instance; for resetting a timer, a cross joined to instance and timeout by an hourglass joined to an instance by an arrow.

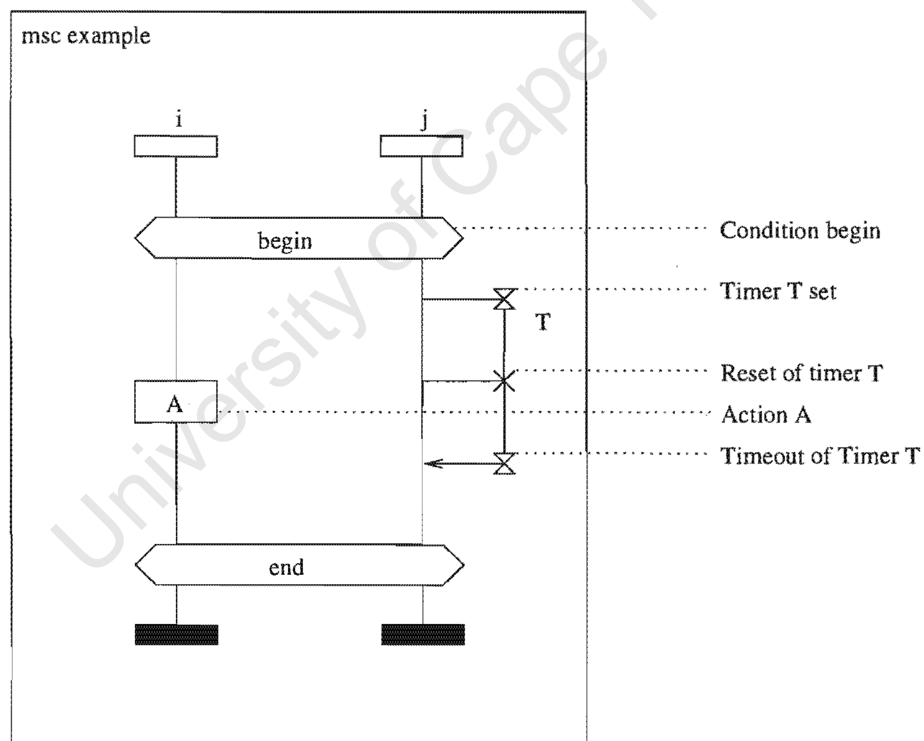


Figure 4: Illustration of Conditions, Action and Timer Events

University of Cape Town

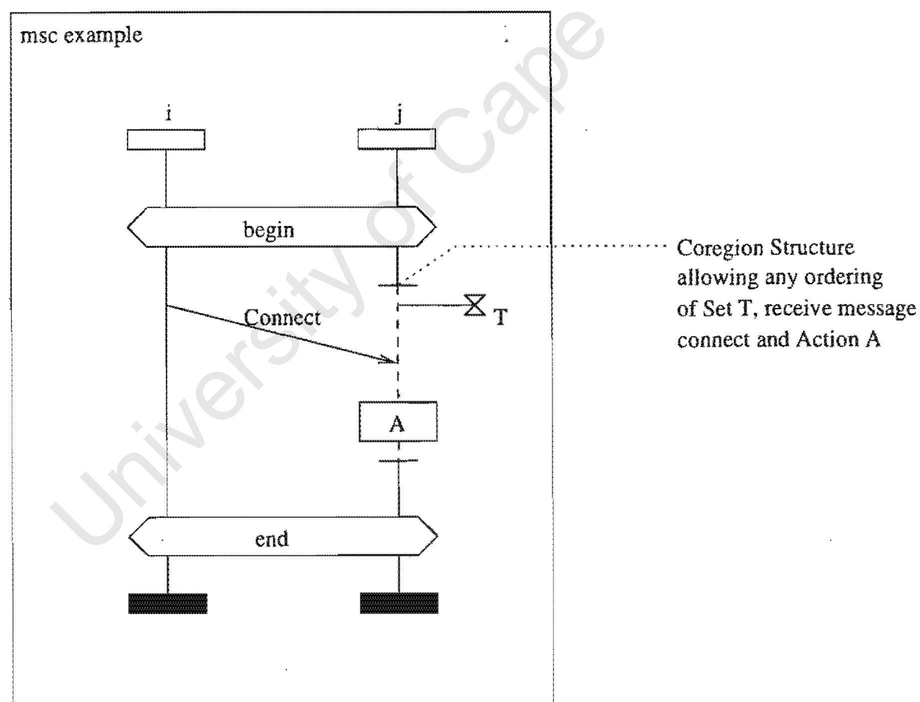
Actions

An action is an independent internal event associated with an instance. Instance Creation and Stop Messages exist to create and stop an instance. Instances that start with a creation message can not execute any events until they are created. The stop event can only be executed by the instance that is terminating and must be the last event on its timeline.

2.6.2 Structural Constructs

Recent additions to the MSC specification have been a number of compositional constructs. Their main purpose was to facilitate the modularization of a specification.

Co-region

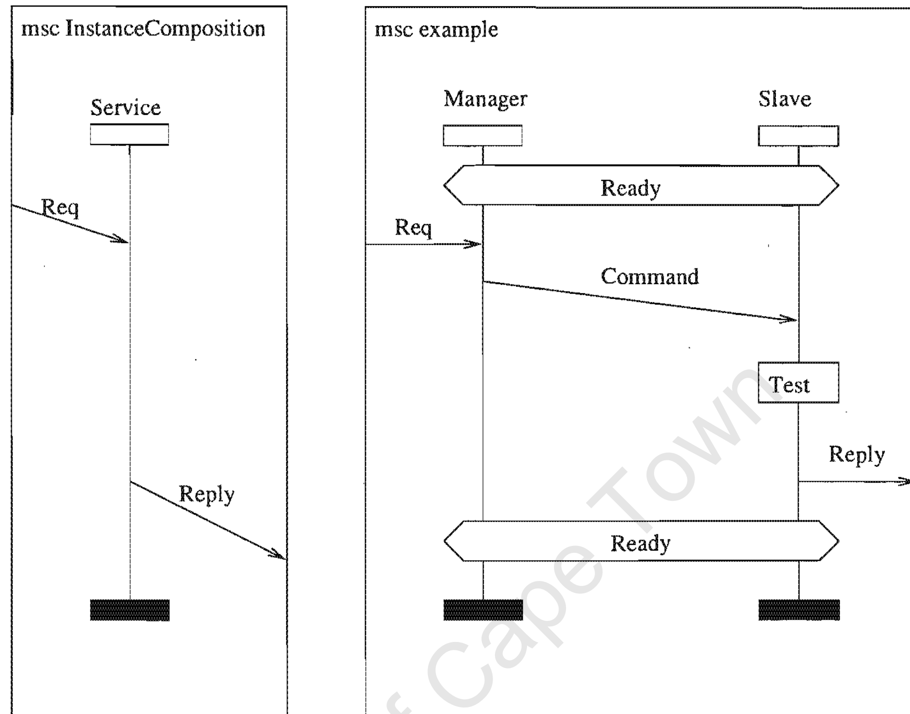


University of Cape Town

Figure 5: Coregion Structure

The events along an instance's axis are strictly ordered. A co-region defines an area along an axis where events can occur concurrently. Such an area is depicted by a dashed timeline.

Instance decomposition



University of Cape Town

Figure 6: Example of Decomposition

A single instance can be used to represent the behaviour of mutiple instances. When decomposing the instance, it is important to maintain the ordering of events.

Inline expressions

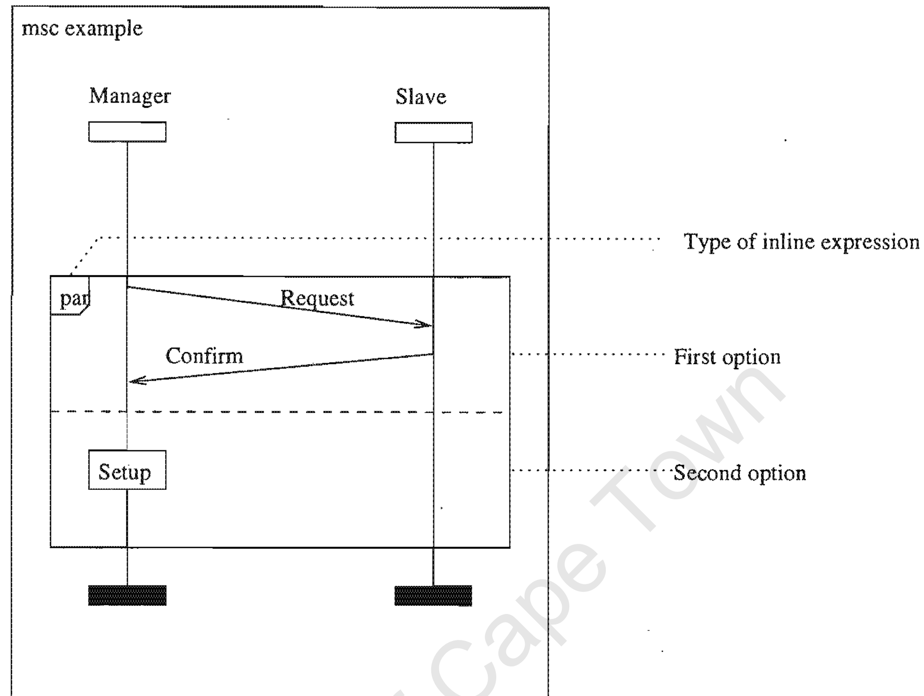


Figure 7: Example of Parallel Inline Expression

Events in an MSC can be structured using inline expressions. These expressions allow events to ordered in parallel, sequentially or in loops. An operator is also provided to optionally terminate the present MSC.

MSC references

MSC references provide the means for referencing other MSCs defined elsewhere in the MSC document. An MSC reference can take the form of an expression on MSCs. The reference MSCs can be linked using parallel, alternate, sequential, looping, exception cases and the substitution of names occurring in the referencing MSC. The graphical representation of a reference is a square with rounded edges, intersecting the axes of the instances occurring in the reference.

University of Cape Town

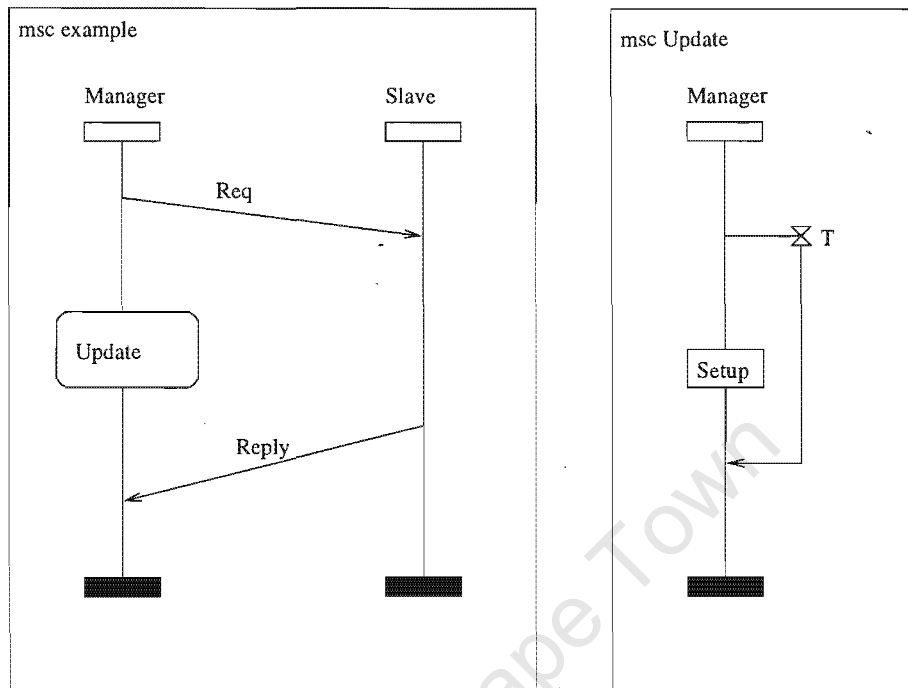


Figure 8: Reference to msc Update in msc Example

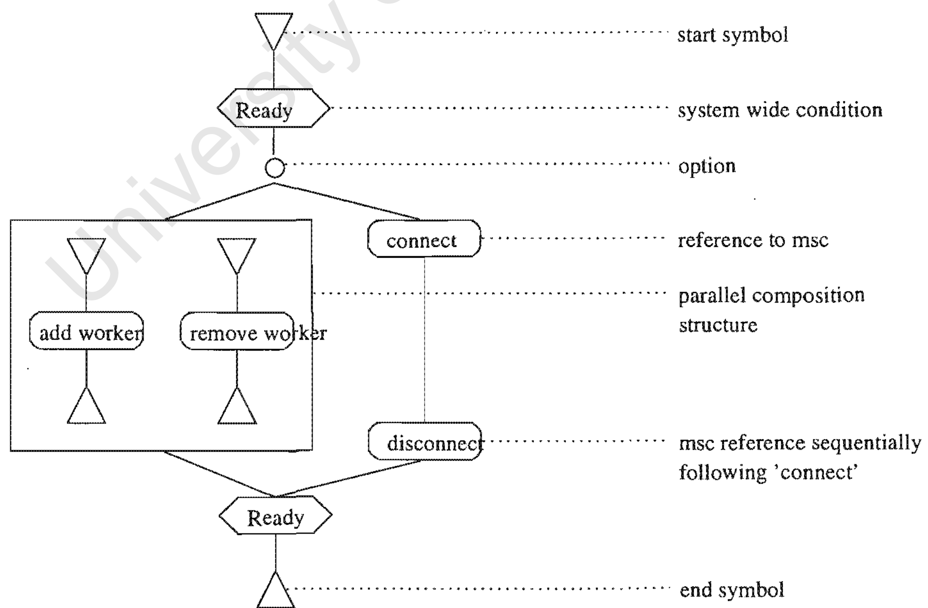


Figure 9: MSC composition with HLMSC

University of Cape Town

High Level MSCs

High level MSCs are used to combine the charts in an MSC document to form a specification. They basically order execution of component MSCs using alternative, sequential and parallel composition.

2.7 Semantic Requirements

Syntactically correct MSCs are not necessarily error free. Certain semantic rules need to be taken into consideration as well. Here follows a brief description of some of these rules as given in [S.M96][M.A94]

- Instance Names: All instances in a MSC must be unique
- Chart Interface Rule: The MSC interface gives information about the contents of an MSC. This information must be consistent with the contents of the document.
- Message input and output:
  - Instances mentioned in the definition of a message event must exist.
  - Message events must be unique to avoid ambiguity.
  - Each message output must have a corresponding input and vice versa.
  - A message output can not be dependent on its input.
- Conditions:
  - Any instance reference by a condition must exist.
  - Corresponding condition declarations must exist on each referenced instance.
- Process Creation:
  - The instance an event creates must be declared.
  - An instance can only be created once.
  - The creation of an instance may not depend on the execution of an event of that process.
- Instance Decomposition:

University of Cape Town

- Within an MSC document there may not be 2 or more charts with the same name.
- All references to charts must be to charts contained in the document.
- Charts should not depend on themselves, directly or through any number of decompositions.
- The decomposed chart should not have any instance names in common with other instances in that MSC.
- The ordering of external communication events in a decomposed instance must be preserved in the corresponding chart.

2.8 Conclusion

MSCs provide a graphical specification technique suited to the description of a system's inter-process communication. A formal semantics exists, allowing the development of tools for specification and testing.

Extensions and variations of MSCs exist, developed by companies before the formal semantics was adopted. This has speeded the development of MSC as a structured FDT.

In its present form, it can be used as a basis for virtually the entire software design and development cycle.

University of Cape Town

Chapter 3

Bisimulation

3.1 Introduction

Comparing specifications is a technique often used in the verification and validation of systems. One might, for example, compare the observed behaviour of a system with its expected behaviour, which allows one to locate unwanted or unexpected behaviour.

Different comparison techniques exist. In this section we will examine how one might do semantic equivalence testing on systems.

3.2 Linear and Branch Time Equivalence

Semantic equivalence can be divided into two categories, namely linear time(LTE) and branch time equivalence(BTE). For processes to be equivalent using LTE, one must examine all possible execution sequences that the processes are capable of. These execution sequences are also known as system traces and LTE is therefore also known as trace equivalence.

BTE, or bisimulation, is similar to LTE in that all possible sequences of events are taken into account. In addition BTE also considers the branching structure of the event sequence. Two systems that produce the same event traces are therefore not necessarily equivalent. Equivalence requires an additional examination of the branching options in each state.

In Figure 10 we see two FSM's, both giving rise to the same traces, ab and ac. LTE would confirm them as equivalent due to this. BTE would not consider them equivalent as no

University of Cape Town

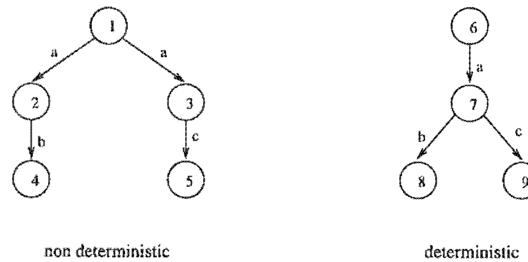


Figure 10: BTE vs LTE Equivalence

comparable state to 7 exists in the non-deterministic FSM where both actions b and c are possible.

The distinction between these two equivalence types originated from the fact that BTE has certain advantages over LTE, the most prominent being that deadlock behaviour cannot be properly modelled using LTE due to the fact that not enough information is kept about possible structures of event sequences.

BTE has however the disadvantage that it distinguishes between systems using the branching structure which is not perceivable to the casual observer. LTE on the other hand, has not been discarded as it has the advantage of simplicity, often a crucial factor in complex systems modelling and test case generation.

Combining the best of LTE and BTE has given rise to various intermediary equivalence types known collectively as decorated equivalence[vG96]. BTE remains however the strongest type of equivalence, meaning that two systems equivalent under BTE will be equivalent for all decorated and linear forms.

University of Cape Town

3.3 Abstraction

Abstraction can play a major role when comparing systems. In the verification and validation process one might wish to ignore certain events or sequences of events to achieve a certain degree of abstraction. These events might however impact on the branching structure of the process, and can therefore not simply be removed. A special transition known as the silent move τ [R.M80], is used to replace these events.

Although τ makes determining equivalence more difficult, the usefulness of being able to hide sequences ensures that new notions of equivalence such as weak bisimulation equivalence and eta bisimulation[vG96] were defined to use it.

3.4 Bisimulation

Before discussing bisimulation, it is useful to formally describe the form a system should be in before comparison. Much equivalence theory is centered around formalisms such as Communicating Concurrent Systems (CCS) [R.M80], Concurrent Sequential Processes (CSP) [F.H85] and Algebra of Communicating Processes (ACP) [BW90].

Of particular interest to this paper is CCS, as theory surrounding it can be applied to non-deterministic automata. We will therefore make use of the notion of processes as described by CCS.

The following section contains a broad overview of the formalisms and equivalence types. Although this thesis makes use of only one equivalence type, namely Strong Observational Equivalence(SOE), the others should not be discarded as they do have uses in validation testing and should be considered to augment the tests available to SOE.

University of Cape Town

3.4.1 Formal Definition of Processes

Finite state processes form the basic building blocks of CCS. By using various operators, these processes can be combined to form more complex processes. Operators include sequential and parallel composition and the Kleene star, all concepts occurring in the theory of regular expressions.

A finite state process is a sextuple $S = (Q, A, T, V, E, q_0)$ [PS90]

- Q is a finite set of states.
- A is a finite set of actions, with τ, a special symbol not in A, called the unobservable action.
- T is the transition function and $T \subseteq Q \times \{A \cup \tau\} \times Q$
- V is a finite set of symbols, different from $A \cup \tau$ called variables.
- $E \subseteq Q \times V$ is a relation called the extension relation.
- $q_0 \in Q$, the start state.

The following representations are used:

- transitions from q : $T(q) = \{< a, q_2 > \mid < q, a, q_2 > \in T\}$
- extensions of q : $E(q) = \{x \mid < q, x > \in E\}$
- destinations of q via a : $T(q, a) = \{q_2 \mid < q, a, q_2 > \in T\}$
- transition between q and q_1 using action a: $q \rightarrow^{s} q_1$

These processes can be graphically represented as labelled transition systems or process graphs. In such graphs, nodes take the form of states and state changes are brought about by transitions under actions.

The use of extensions forms the only difference between FSP and nondeterministic finite state automata (NDFA). One would use the extension relation and variables to represent different types of acceptance. For example, a variable might be used as a counter to indicate progression through a sequence of events and thereby control acceptance. By removing the extension relation, or limiting V to a single acceptance set, we obtain a NDFA with the unobservable transition.

3.4.2 Equivalence types

Equivalence of FSPs is decided by determining state equivalence. 'State equivalence' can have a number of different meanings.

CCS introduces the concept of observationally equivalent processes and an observer. The observer decides if the processes are equivalent if no differences can be 'observed'. In particular, three equivalence types have been defined for CCS, namely observational equivalence, strong observational equivalence and failure equivalence.

k-observationally equivalent

For $p, q \in Q, p, q$ are k-observationally equivalent ($p \approx_k q$) if

1. $E(p) = E(q)$, when $k = 0$. Otherwise $k > 0$
2. For every $s \in A*$
   (a) if $p \rightarrow^{s} p_1$ then $(\exists q_1 | q \rightarrow^{s} q_1$ and $p_1 \approx_{k-1} q_1)$
   (b) if $q \rightarrow^{s} q_2$ then $(\exists p_2 | p \rightarrow^{s} p_2$ and $q_2 \approx_{k-1} p_2)$

University of Cape Town

Observationally equivalent

States p, q are observationally equivalent $(p \approx q)$ if $p \approx_k q \forall k \geq 0$

k-limited observationally equivalent

For $p, q \in Q$, p, q are k-limited observationally equivalent $(p \simeq_k q)$ if

1. $E(p) = E(q)$, when $k = 0$. Otherwise $k > 0$
2. $\forall a \in A \cup \{\epsilon\}$
   (a) if $p \rightarrow^a p_1$ then $(\exists q_1 | q \rightarrow^a q_1$ and $p_1 \simeq_{k-1} q_1)$
   (b) if $q \rightarrow^a q_2$ then $(\exists p_2 | p \rightarrow^a p_2$ and $q_2 \simeq_{k-1} p_2)$

Limited Observationally equivalent

States p, q are limited observationally equivalent $(p \simeq q)$ if $p \simeq_k q \forall k \geq 0$

Strong Observational Equivalence

If p, q are states of FSPs without the τ transition, p, q are strongly equivalent $(p \sim q)$ if $p \simeq q$

Failure Equivalence

If p, q are states of FSPs where every state is an accept state, then p, q are failure equivalent $(p \equiv q)$ if failures(p) = failures(q), where failures(p) is defined as the set of sequences of actions not possible from p.

3.4.3 Complexity of determining equivalence

Before choosing between language and bisimulation equivalence, the relative complexity should be taken into account. This complexity is a function of the number of states and it is therefore infeasible to perform equivalence testing on large systems.

The following diagram lists some of the major complexity results. [RR95]

University of Cape Town

	Bisimulation	Language Equivalence
NDFA	O(mlogn)	PSPACE complete
NDA	polynomial	undecidable
DFA	polynomial	polynomial

Table 1: Complexity of Bisimulation algorithms with m transitions and n states

Bisimulation clearly has a distinct computational advantage over language equivalence and should therefore be used wherever possible. This advantage arises from the fact that when testing for bisimulation, nondeterminism does not give rise to the huge amount of comparative computation. This is because duplicated actions in bisimilar states must lead to bisimilar states.

The limiting factor for bisimulation is the space complexity of $O(m+n)$.

As can be seen from Table 1, deterministic finite state automata, i.e., automata where for each action, states can have at most a single transition, have the property that deciding bisimulation and language equivalence are both polynomial.

As no branching structure exists in a DFA, any bisimulation equivalence test would be the same as a linear time equivalence test. If such a system is branch time equivalent then it would also be trace equivalent.

This makes use of the following result given by [PS90].

If p, q are the start states of DFA accepting languages L(p) and L(q) respectively, then:

1. $p \approx_1 q \Leftrightarrow p \approx q$ (or $p \simeq q$ or $p \sim q$).
2. $p \approx_1 q \Leftrightarrow L(p) = L(q)$

University of Cape Town

Chapter 4

Bisimulation on Message Sequence Charts

4.1 Introduction

MSCs have limited use in system verification and validation unless integrated with other recognised formal description techniques(FDT). The primary reason for this is that MSCs make no attempt to describe internal process behaviour.

It is important for the development of MSCs as a description technique that any verification and validation techniques that can be performed without any reliance on other FDTs be done as efficiently as possible. An example of this would be the automation of equivalence testing by means of toolset, allowing systems specified with MSCs to be tested against their requirements documents.

Various equivalence types exist. Simple linear time equivalence (LTE) is useful for testing traces against a specification, but it cannot identify deadlocking systems because of their branching structures. Another form of equivalence, namely branch time equivalence (BTE) is more restrictive, but does take the branching structure into account as well. Another consideration is the complexity of the algorithms for computing LTE and BTE as shown in Table 1. Although MSCs are deterministic in nature, and even alternative composition should be interpreted as delayed choice [JS95], applying LTE is computationally expensive with complexity polynomially related to the number of states. BTE allows the FSM to be non-deterministic, and can decide equivalence in $mlogn$ for m transitions and n states. The MSC specification designer should be consistent in either using or avoiding the sequential

University of Cape Town

composition where it might give rise to non-determinism in the FSM. This will limit the restrictive effect of BTE.

Although this thesis focuses on the use of BTE on MSCs, LTE is also implemented to illustrate differences in deciding equivalence.

4.2 An Equivalence for MSCs

MSCs originated from the desire to structure system traces into a manageable form. When comparing MSCs, a direct form of trace equivalence is most desirable, but at $O(n^3)$, it is also the most computationally expensive.

Bisimulation algorithms have lower complexity $O(mlogn)$, but have the disadvantage that trace equivalent systems can be found to be not equivalent. This is especially noticeable when dealing with branching structures and concepts such as delayed choice. We however consider the computational advantage of bisimulation sufficient motivation for its use.

Different types of bisimulation (or branch time equivalence) exist [PS90]. Selecting a bisimulation type is dependent on the system under consideration. An equivalence notion suited to MSCs is "Strong Observational Equivalence" (SOE), which concentrates on the events and actions an observer of the system might see.

SOE is a type of observational equivalence. A characteristic of SOE differentiating it from observation equivalence is that it removes the possibility of the $\tau-$move. This is a transition between states that is unobservable, and forms part of the process algebra on which Communicating Concurrent Systems (CCS) is based [C.S89]. Eliminating the $\tau-$move reduces the time complexity of determining equivalence from polynomial to O(mlogn)[Fer90], where m is the number of transitions and n is the number of states.

University of Cape Town

4.3 Applying Strong Observational Equivalence to MSCs

Message Sequence Charts (MSCs) provide a formal specification language for modelling inter-process communication. Although graphical and textual representations for MSCs exist, neither are conducive to bisimulation testing.

Bisimulation algorithms for SOE required the system be in the form of a non-deterministic finite state automata (NDFA), of the form $S = (Q, A, T, q_0)$, where

- Q is the set of states,
- A is the set of actions,
- T is the transition function and $T \subseteq Q \times A \times Q$ and
- q_0 is the initial state.

The transition function T, can be written in a number of ways. In each of the following p is mapped to q by a and $p, q \in Q$ and $a \in A$

- $T_a[p] = q|(p, a, q) \in T$
- $T_a^{-1}[q] = p|(p, a, q) \in T$
- $p \rightarrow^a q|(p, a, q) \in T$

The formal definition for SOE of a *system* can be stated as follows:

Given a labelled transition system $S = (Q, A, T, q_0)$, a binary relation $\rho \subset Q \times Q$ is a bisimulation $\iff$:

$\forall(p_1, p_2) \in \rho \wedge \forall a \in A$
$\forall r_1, p_1 \rightarrow^a r_1 \Rightarrow \exists r_2 | p_2 \rightarrow^a r_2 \wedge (r_1, r_2) \in \rho)) \wedge$
$\forall r_2, p_2 \rightarrow^a r_2 \Rightarrow \exists r_1 | p_1 \rightarrow^a r_1 \wedge (r_1, r_2) \in \rho))$

An MSC specification must therefore be converted to this format. This conversion is by no means trivial. For each MSC, an NDFA must be built to contain all event sequences in that MSC. Thereafter one must consider the composition of MSCs by means of structures such as High Level MSCs and MSC references. The resulting NDFA has transition sequences for all possible event execution sequences given by the MSC document.

4.4 Notation used by grammar

The grammar given by the recommendation for the textual representation of MSC takes the form of a Backus Naur Form, or rule based system. Both an instance and event oriented syntax are provided, but for simplicity sake, this thesis will concentrate on the instance oriented syntax.

The following symbols are used:

University of Cape Town

- Words not enclosed by angle brackets are terminal symbols
- Words in angle brackets are either non-terminal symbols or identifiers when containing the word "name".
- If a rule can be expanded in more than one way, the OR operator |, is used between options.
- Brackets are used to group constructs.
- The asterisk indicates zero or any number of repetitions of the preceding sequence.

For example:

```
<non terminal1> ::= terminal <non terminal2> | <non terminal3>
```

Indicates that <non terminal1> is expanded to either "terminal <non terminal2> " or "<non terminal3>"

The following sections examine the contructs of MSC96 and investigate how one might convert a specification using the syntax rules to generate a finite state machine.

4.5 Message Sequence Chart Document

```
<msc document> ::= <mscdocument head> <msc document body>
<msc document head> ::= mscdocument <document name>
          [related to <sdl reference> ] <end>
<msc document body> ::= <message sequence chart>*
```

An MSC document provides the wrapping structure for a specification. Information stored in the MSC document does not impact on the structure of the NDFA and can therefore be omitted. In general information such as references or comments will not be included in the NDFA.

University of Cape Town

4.6 Basic Message Sequence Chart

4.6.1 Message Sequence Chart

```
<message sequence chart> ::= msc <msc head>
          {<msc body> | expr <msc expression> } endmsc <end>
```

A Message Sequence Chart can be considered the basic structural component of an MSC document.

An MSC consists of a head and a body. The head contains information about the MSC. The body can take two forms. It can either consist of the actual instances and their events or an expression combining MSCs into a structure.

The first step in converting an MSC is identifying whether the body is a normal MSC or a structure of MSCs. If it is the conventional collection of instances and events, it can be converted immediately into a NDFA.

If the body consists of a structure of MSCs, each MSC must be converted in turn. The converted NDFAs are then combined using the operators given to create the final NDFA.

4.6.2 Instance

```
<instance head statement> ::= instance [ <instance kind> ]
          [ <decomposition> ]
<instance end statement> ::= endinstance
```

An instance is the primary component of an MSC. Associated with it are the events executed by the component or process the instance represents. Events are ordered by the axis of the instance with higher events occur earlier.

An NDFA is constructed from an instance list by associating and ordering the events of the instance relative to other instances. This is determined by:

- Instance's axis(e.g: higher events occur earlier).
- Structures giving further ordering on events (e.g: inline structures, coregions).
- Rules applying to certain events (e.g: messages output before message input).

After applying these rules, an NDFA is generated for the related collection of instances.

University of Cape Town

4.6.3 Message

```
<message event> ::= <message output> | <message input>
<message output> ::= out <msg identification> to <input address>
<message input> ::= in <msg identification> from <output address>
```

The grammar describes four types of message events, namely normal message input and output and incomplete message input and output. These events provide the basic communication between instances and their environment.

Conversion treats messages as transitions. The MSC recommendation requires enough information to be stored in each message event to ensure no ambiguity and this uniqueness is maintained in the conversion process by transferring this information to the corresponding transition.

4.6.4 Environment

A bounding box around instances represents the environment of an MSC. Messages provide a means of interaction between the instances and their environment by means of gates. These gates have particular importance when combining MSCs. A message may be sent from an instance to a gate in one MSC, and then from a gate of the same name to another instance in some other MSC. This allows information flow between MSCs.

The environment does not in itself provide any ordering on events and does not therefore influence the construction of the NDFA.

4.6.5 General Ordering

```
<orderable event> ::= [ <event name> ] { <message event> |
          <incomplete message event> | <create> | <timer statement>  |
          <action> } [ before <event name list>]
```

The ordering construct allows one to specify an ordering of basic events based on their names. The ordering construct can only specify a list of events that this event must occur before. This necessitates parsing the event structure to generate a list of dependencies prior to conversion to prevent constructing event sequences that are subsequently found to be impermissible. Once the dependency list is constructed, it can easily be determined whether events are permitted by checking whether all dependent events have been executed.

University of Cape Town

4.6.6 Condition

```
<condition> ::= <condition identification>
<shared condition> ::=  <condition identification > shared
          { [<shared instance list>] | all }
<condition identification> ::= condition <condition name list>
```

Conditions define either a global or local system state. The shared name list is a list of those instances to which the condition applies.

The condition serves two roles in the structuring process. The first is in the composition of MSCs. Composition of MSCs is only allowed if the final condition of an MSC is equivalent to the first condition in the following MSC. If the condition is not equivalent, the MSC may not sequentially follow each other.

Conditions can also influence event ordering when used on instance axes. Conditions effectually partition event sequences.

4.6.7 Timer

```
<timer statement> ::= <set> | <reset> | <timeout>
<set> ::= set <timer name> [, <timer instance name>]
         [(<duration name>)]
<reset> ::= reset <timer name> [, <timer instance name>]
<timeout> ::= set <timer name> [, <timer instance name>]
```

The three timer statements set, reset and timeout and be treated as independent events as they do not cause interaction between instances. Timers are uniquely identified by name and calling instance. They are converted, together with their details, to a corresponding transition.

4.6.8 Action

```
<action>::= action <action character string>
```

Actions describe the internal activity of an instance and are independent of all other events. A simple conversion to a transition is therefore possible.

University of Cape Town

4.6.9 Instance Creation

```
<create> ::= create <instance name> [ (parameter list) ]
```

If an instance is to be created, none of its events may be executed until the other instance executes the creation event. This requires the examination of all event lists before the MSC is converted, as no corresponding information is stored on the created instance. This instance is therefore not included in the general interleaving of instances until such time as it is created. The parameter list gives information related to the creation as is serves no structural purpose.

4.6.10 Instance Stop

```
<stop> ::= stop
```

The stop event is the final event on an instance's axis. The name of the instance executing this event is added to the corresponding transition to remove all ambiguity.

4.7 Structural Concepts

4.7.1 Coregion

```
<coregion> ::= concurrent <end> <coevent>* endconcurrent
<coevent> ::= <orderable event> <end>
```

The coregion structure removes the ordering provided by the instance axis. A generalised ordering of the events in the coregion must therefore be integrated into the interleaving of instance events.

The simplest method of doing this is by integrating the coregion structure into the conversion process of the surrounding MSC. The alternative would be to convert the coregion and surrounding MSC separately and then interleave the two, a task that would become extremely complicated as integration with the other constructs are considered.

University of Cape Town

4.7.2 Instance Decomposition

```
<instance head statement> ::= instance [ <instance kind> ]
         [ <decomposition> ]
<decomposition> ::= decomposed
         [ as <message sequence chart name> ]
```

Abstraction can be achieved using the decomposition structure. Decomposition allows a single instance to model the behaviour of multiple instances. The decomposed instances are stored in a separate message sequence chart, which is referenced through the decomposition. The construction of the NDFA requires the substitution of the composed instance with the decomposition. Care must be taken to ensure the preservation of the event ordering of component instances and the integration of the messages of the decomposition.

4.7.3 Inline Expression

```
<inline expr> ::= <loop expr> | <opt expr> |
         <alt expr> | <par expr> | <exc expr>
<shared inline expr> ::= < shared loop expr> |
         < shared opt expr>  |  < shared alt expr> |
         < shared par expr> | < shared exc expr>
```

Inline expressions allow greater control in the structuring of instance events. Composition of groups of events inside MSCs is made possible with operators such as alternative, parallel, iterate, exception and optional regions. These structures can be limited to a single instance or shared across multiple instances.

An inline expression structure is integrated into the conversion of the rest of a MSC and not converted independently. This conversion is complicated communication between an inline structure and the rest of the MSC by means of gates. Care should therefore be taken to ensure that the event ordering is preserved.

Loop

```
<loop expr> ::= loop [ <loop boundary> ] begin
         [ <inline expr identification> ] <end>
         [ <inline gate interface> ] <msc body>
```

University of Cape Town

```
          loop end
<shared loop expr> ::= loop [ <loop boundary> ] begin
          [ <inline expr identification> ] <shared> <end>
          [ <inline gate interface> ] <instance event list >
          loop end
```

The conversion of the loop structure to an NDFA structure presents the greatest challenge. No efficient method of modelling a looped sequence of events for a large, yet limited number of events exists without using an extended NDFA.

An infinite loop is easily modelled in the NDFA with a backward edge.

Optional

```
<opt expr> ::= opt begin
          [ <inline expr identification> ] <end>
          [ <inline gate interface> ] <msc body>
          opt end
<shared opt expr> ::= opt begin
          [ <inline expr identification> ] <shared> <end>
          [ <inline gate interface> ] <instance event list >
          opt end
```

The optional operator allows a choice between the structure given and an empty MSC. In an NDFA this is represented by an edge by passing the event sequence given in the optional structure.

University of Cape Town

Exception

```
<exc expr> ::= exc begin
          [ <inline expr identification> ] <end>
          [ <inline gate interface> ] <msc body>
          exc end
<shared exc expr> ::= exc begin
          [ <inline expr identification> ] <shared> <end>
          [ <inline gate interface> ] <instance event list >
          exc end
```

The exception operator provides the execution flow the option of a sequence of events leading to an end to all events or the continuation with the rest of the MSC. Simply put, a terminating branch is added to the NDFA.

Alternative

```
<alt expr> ::= alt begin
         [ <inline expr identification> ] <end> <msc body>
        { alt <end> [ <inline gate interface> ] <msc body> }*
         alt end
<shared alt expr> ::= alt begin
         [ <inline expr identification> ] <shared> <end>
         [ <inline gate interface> ] <instance event list >
         {alt <end> [ <inline gate interface> ]
         <instance event list> } *
         alt end
```

University of Cape Town

The alternative operator allows the execution sequence the option of any of a number of possible sequences.

The best method of converting the alternative structure is converting each of the alternative sequences and then combining them.

This implies that we are using a non-deterministic choice. Removing the non-determinism would add substantially to the complexity of the algorithm. This is discussed in section 3.4.3.

Parallel

```
<par expr> ::= par begin
         [ <inline expr identification> ] <end> <msc body>
         { par <end> [ <inline gate interface> ] <msc body> }*
         par end
<shared par expr> ::= par begin
         [ <inline expr identification> ] <shared> <end>
         [ <inline gate interface> ] <instance event list >
         {par <end> [ <inline gate interface> ]
```

```
        <instance event list> } *
        par end
```

The parallel operator provides the execution sequence any number of sequence structures to occur in parallel.

Although conceptually simple, the parallel structure has some important implications. Foremost amongst these is the state space explosion caused by interleaving event sequences. This can be limited by using the parallel operator sparingly.

4.7.4 MSC reference

```
<shared msc ref> ::= reference [ <msc ref identification>:]
        <msc ref expr> <shared> [ <ref gate interface> ]
<msc ref> ::= reference [ <msc ref identification>:]
        <msc ref expr> [ <ref gate interface> ]
```

The MSC reference provides a way of describing systems modularly. These references can take the form of an individual MSC or an expression combining MSCs with the following operators: alternative, parallel, sequential, loop, optional, exception and substitute.

4.7.5 High-Level MSC

```
<msc expression> ::= <start> <node expression> *
<start> ::= <label name> { alt <label name> }* <end>
<node expression> ::= <label name> : { <node> seq
        (<label name> { alt <label name> } * ) | end } <end>
<node> ::= ( <msc ref expr> ) | empty | <msc name> |
        <par expression> | condition <condition name list> |
        connect
<par expression> ::= expr <msc expression> endexpr
        { par expr <msc expression> endexpr} *
```

High-Level MSCs (HMSCs) allow the composition of charts, references and conditions. The start state can be one of a number of alternatives, as specified by the label name.

Components of the HMSC can be joined using sequential, alternative or parallel composition operators. Composition can be controlled using the condition statement. Each component

University of Cape Town

of the HMSC is converted independently into an NDFA. The resulting NDFA are then combined using the specified operators.

Combining NDFAs is achieved in the same method described for inline expression operators. The structures are convered independently and then combined.

4.8 Optimisations

The resulting NDFA can clearly be considerably optimised. The possibility of long chains of equivalent transition sequences exists. Collapsing these to minimise states could dramatically reduce the number of states in the NDFA.

At this stage no work has been done by this project on state reduction of the resultant FSM.

University of Cape Town

Chapter 5

Bisimulation Algorithms

University of Cape Town

5.1 Introduction

As mentioned in the chapter on bisimulation, equivalence exists in various forms. Of the equivalence forms mentioned, bisimulation has algorithms with the best time complexity. This chapter examines the work of various people in the development of bisimulation algorithms. The algorithm with best time and space complexity will then be applied to MSCs.

5.2 Algorithm Goal

The aim of bisimulation algorithms is to find states in the systems under consideration which meet the conditions for bisimulation as described in Sec 4.3. Once again, states are only equivalent if transitions exist to further states, which are also equivalent. For systems to be equivalent, one needs to determine whether the start states are equivalent.

5.3 Relational Coarsest Partitioning

Early work by Kanellakis and Smolka [PS90] found a relation between bisimulation testing and a combinatorial problem known as relational coarsest partitioning.

Partitioning is the act of dividing a set into disjoint subsets, such that the union of all subsets give the initial set, and elements appear only once in all subsets. The subsets making up a partition are also known as blocks. The process of partitioning is determined

using some relationship between elements. For NDFAs, this relationship takes the form of the transition function.

The relational coarsest partition problem is described in [PS90], and can be defined as finding for a family of transition relations between states T, and initial partition P over a set U, the partition $\rho = \{B_i | i \in J, \text{the subscript of the sets of} \rho\}$ with the fewest blocks such that the following conditions hold:

$\forall i, j \in J, \forall p, q \in B_i$

$$T_a[p] \cap B_j \neq \emptyset \Leftrightarrow T_a[q] \cap B_j \neq \emptyset$$

The initial partition is refined into ρ by means of splitting the initial partition until the above condition is true for each $a \in A$.

Partitioning is an incremental process, with each partition a refinement of the previous. This refinement process always terminates with a unique coarsest partition, or partition with the fewest blocks. This can be seen from the fact that there are a finite number of states, and after each refinement, the number of sets in the partition increases. A refinement increases the number of sets in a partition, however the number of sets can clearly never exceed the number of elements in the NDFAs. This implies the refinement process terminates after at most n-1 steps, where n is the number of elements.

The relationship between the relational coarsest partitioning problem and bisimulation testing follows from the fact that bisimulation partitions the elements of NDFAs into sets of equivalent states. The relations between elements are equivalent to transitions, mapping states onto states, and in the context of the partition, blocks onto sets of blocks.

Formally, bisimulation and the relational coarsest partitioning problem can be seen to be logically equivalent by means of the following simple transformation:

$$T_a[p] \cap B_j \neq \emptyset \equiv \exists r \in B_i | (p \rightarrow^a r)$$

The result of repeatedly refining a partition until the coarsest partition is reached, is a partition consisting of blocks that contain equivalent elements from the NDFAs.

University of Cape Town

5.3.1 Implementation of an O(mn) algorithm

Introduction

The following algorithm is an adaptation by J-C Fernandez[Fer90] of the Smolka-Kanellakis algorithm[PS90].

Given an initial partition ρ_i, the algorithm identifies blocks in the partition that break the bisimulation condition and then partitions these blocks until the conditions for bisimulation are met. Repeatedly applying this step leads to the unique coarsest partition of the set. [RR87]

A set W is used by the algorithm to keep track of the blocks (B_i) that have been used to refine the partition. The reason for keeping this set is that refining more than once with a particular B_i has no effect on the partition [PS90].

The starting partition (ρ_I) is the set of all states.

University of Cape Town

Pseudo Code

$W, \rho = \rho_I, \rho_I$

repeat

choose and remove any B in W

for each $a \in A$

$I_{a,B} = \{X \in \rho | X \cap T_a^{-1}[B] \neq \emptyset \wedge X \not\subseteq T_a^{-1}[B]\}$

$I_{a,B}^{1,2} = \{X \cap T_a^{-1}[B] | X \in I_{a,B}\} \cup \{X - T_a^{-1}[B] | X \in I_{a,B}\}$

$\rho = \rho - I_{a,B} \cup I_{a,B}^{1,2}$

$$W = W - I_{a,B} \cup I_{a,B}^{1,2}$$

until $W = \emptyset$

5.3.2 Implementation of O(mlogn) algorithm

Introduction

Paige and Tarjan improved on the Kanellakis/Smolka algorithm to achieve a time complexity of O(mlogn). The improvement in complexity is achieved by intelligently choosing the blocks used to refine a partition and keeping track of how blocks of the partition are split.

Once again, a set W is kept of blocks that will be used to refine a partition. The elements of this set can however take two forms, namely, simple or compound. Simple blocks are single blocks. Compound blocks consist of the union of compound and/or simple blocks.

Pseudo Code

$W, \rho = \rho_I, \rho_I$

repeat

Remove the block B from W

For each action $a \in A$

If B is compound,

For each set X such that $X \in \rho \wedge X \subseteq T_a^{-1}[B]$

Replace X in ρ by X_1, X_2 and X_3 where

$$X_1 = (X \cap T_a^{-1}[B_1]) - T_a^{-1}[B - B_1]$$

University of Cape Town

$X_2 = (X \cap T_a^{-1}[B - B_1]) - T_a^{-1}[B_1]$

$X_3 = X \cap T_a^{-1}[B_1] \cap T_a^{-1}[B - B_1]$

Update W as follows:

If X is part of a compound block in W,

replace X by $(X_1 \cup X_2 \cup X_3)$

Else add $X_1 \cup X_2 \cup X_3$ to W

If B is simple

For each set X such that $X \in \rho \wedge X \not\subseteq T_a^{-1}[B] \wedge X \cap T_a^{-1}[B] \neq \emptyset$

Replace X by X_1 and X_2 where

$X_1 = X \cap T_a^{-1}[B_1]$

$X_2 = X - T_a^{-1}[B_1]$

Update W as follows

If X is part of a compound block in W, replace X by $(X_1 \cup X_2)$

Else add $X_1 \cup X_2$ to W

until $W = \emptyset$

University of Cape Town

5.4 Trace Equivalence using Bisimulation

Bisimulation testing can be extended to trace equivalence testing. This involves converting the finite state machine fed to the bisimulation algorithm, from a non-deterministic to deterministic finite state machine.

This has been done using a algorithm supplied by [A.A85]. *Dstates* is the set of states of the new DFA. *Dtran* is the transition table for the new DFA.

$move(T, a)$ is the set of NDFA states to which there is a transition on input symbol a from some NDFA state S in T

Let S_0 be the start state. It is initially the only state in *Dstates*, and is unmarked.

While there is an unmarked state T in Dstates do

    mark T

    for each input symbol a do

        $U = move(T, a)$

        if U is not in *Dstates* then

            add U as an unmarked state to *Dstates*

        $Dtran[T, a] = U$

    end

end

After the finite state machine has been made deterministic, the bisimulation algorithms mentioned in the previous section can be applied. The additional complexity of applying the deterministic conversion algorithm places limits on the use of trace equivalence.

University of Cape Town

Chapter 6

Implementation

6.1 Introduction

The bisimulation algorithm consists of repeatedly applying the refinement step for sets generated by previous iterations. Although this in itself is not a complicated procedure, applying the algorithm by hand is time consuming and prone to human error.

This project therefore provides a tool to apply the testing as described in earlier chapters, and thereby confirm the applicability of bisimulation testing on MSCs. In addition, this tool allows limited editing of an MSC specification.

University of Cape Town

6.2 Background to Implementation

The tool was developed on a Windows 95 platform using C++. The zApp application framework was used to produce the graphical user interface, and thereby ensure portability to other operating systems with a minimal number of changes.

The MSC grammar used was taken directly from the 1996 Z.120 Recommendation and therefore contains the complete rule set and all structures and constructs such as high level MSCs, decomposition, inline expressions and MSC references.

A parser was produced for this grammar using a lexical analyser and a parser generator known respectively as "LEX" and "YACC". This was done prior to knowledge of a similar parser created by the University of Erlangen. The parser checks syntax. Due to constraints

on time, it does not check whether the information given makes sense semantically within an MSC specification.

The parser was then integrated into the GUI, allowing textual MSC documents to be loaded from files and stored in a data structure for manipulation and editing.

If syntactic errors are found in the document, an error box is displayed informing the user of the line in the textual specification where the error is found. The user can then externally edit and correct the error by means of any text editor.

No semantic checking is performed during parsing. The user should therefore ensure the semantic correctness of the document.

University of Cape Town

Figure 11: Loading the MSC

6.3 Displaying an MSC

Once an MSC is loaded, the individual charts found in the document are displayed in a dropdown listbox just below the main pull-down menus. Selecting an MSC from this list will display the MSC in the main display window. Multiple MSCs from the same document may be displayed at any particular time. These windows may be resized, maximised and minimised to allow visual comparison of the respective MSCs.

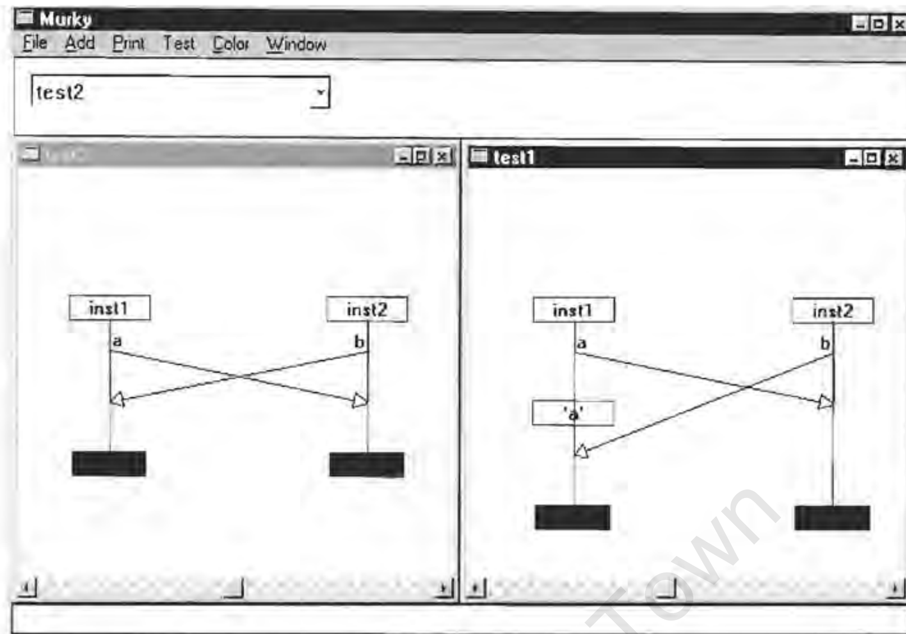


Figure 12: Displaying MSCs

At present, only MSCs consisting of the basic set of MSC structures and events in an instance-orientated representation can be displayed. Such a representation can include all orderable and non-orderable events, but omits inline expressions and reference expressions. Structured MSCs is achieved using high level MSCs, which are catered for.

6.4 Editing MSCs

After selecting and displaying an MSC using the dropdown list-box, editing can take place. These changes can then be saved using the *Save* option in the *File* menu.

Left clicking the item while holding down the keyboard's "Control" key allows the event to be deleted. Deletion of structures such as instances and coregions is only possible once all events have been removed from them.

Once any event or structure is created, a routine is called to update the screen location of all events. This is to ensure that the graphical representation retains its integrity. This however has the disadvantage that the user cannot place items freely.

Care should be taken when creating all of these events as no semantic checking is done, and an invalid MSC can be created.

University of Cape Town

The following explanation of the creation of MSC events depicts how one might go about designing an MSC. These events can be created via the *Add* menu.

6.4.1 Creating an Instance

To create an instance, the user first selects the MSC to which it will be added. The user then selects a name, unique to this MSC. The instance is then automatically added.

6.4.2 Message Events

All messages types can be added to a basic MSC. The user is prompted to select which instances are involved in the message exchange, the name of the message and the message type. A user might therefore create a message from "inst1" to the environment called "mess1." After making the relevant selections, the user has to place the message in the desired location on the instance axis of the involved instances. This is achieved by clicking on the axis of the transmitting axis, and while the mouse button is still depressed, dragging the cursor to the input position of the receiving instance. The mouse button is then released and the message is drawn. Should the user attempt to select an incorrect instance axis or a point off the axis, an error message is displayed offering another placement attempt.

University of Cape Town

6.4.3 Instance Creation Event

An instance can be created from another instance. This prevents the created instance from performing any actions until after the creation event is received. An instance is created by simply specifying the instances doing the creation and being created. After selecting "OK", the user selects desired position on the creation instance.

6.4.4 Timer Statements

Three types of timer statements exist. All are created by means of the same menu option. The user is asked to select the instance the timer will occur on, the name of the timer and the timer type. Once again the user finally selects the location on the timer instance axis to position it. The display algorithm then relates the timer to any other timer with the same identity on this axis.

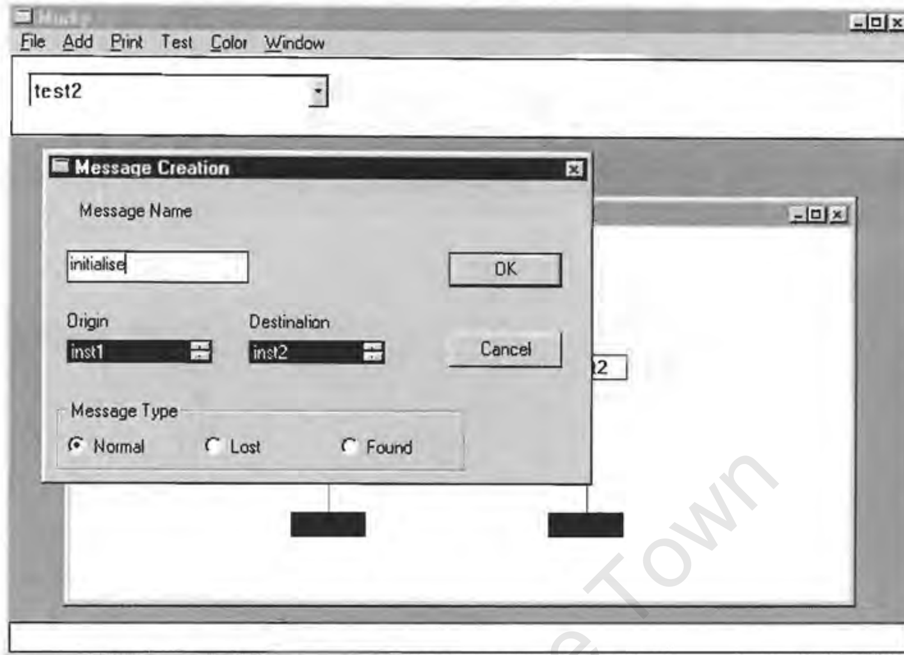


Figure 13: Choices involved when adding a message

University of Cape Town

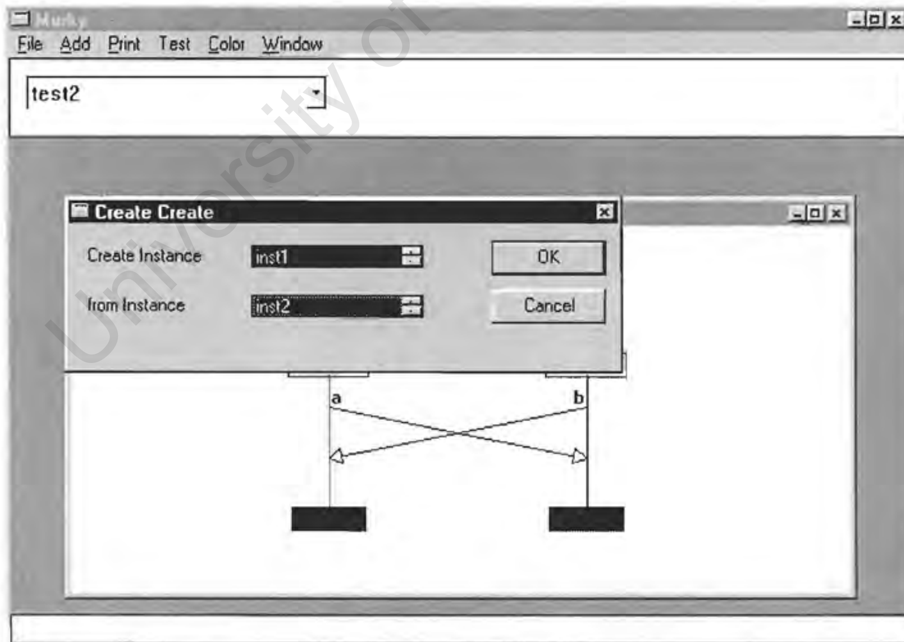


Figure 14: Instance Creation Event

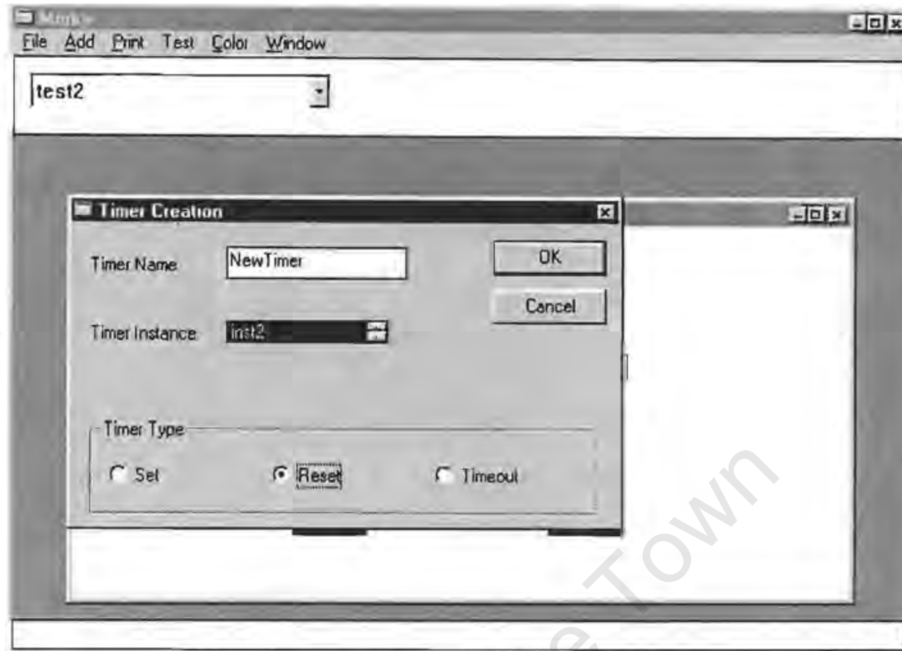


Figure 15: Adding a Timer

6.4.5 Action Event

The action event follows the same creation sequences as the previous events. The user is prompted for a name and instance for the action. Once completed, the action is position on the selected axis.

6.4.6 Instance End and Stop events

These events terminate an instance axis and their creation is identical. The user selects from the *Add* menu the type of termination event wanted, and then clicks on the existing termination event. This event is then replaced by the new selection.

6.4.7 Shared Conditions

Creation of a condition event follows a number of steps. Firstly the condition option is selected from the add menu. From this dialog, only the name of the condition is selected. The user then has to select a point on each of the instance axes sharing this condition. Once all points have been selected, the user presses the right click mouse button to indicate completion. The condition is then added to each of the selected instances.

University of Cape Town

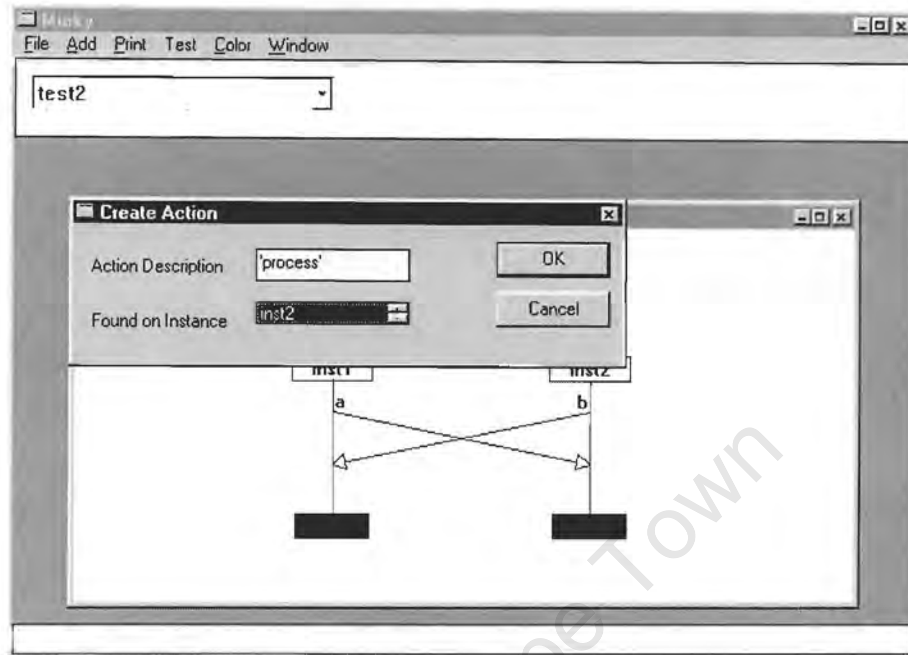


Figure 16: Adding an Action

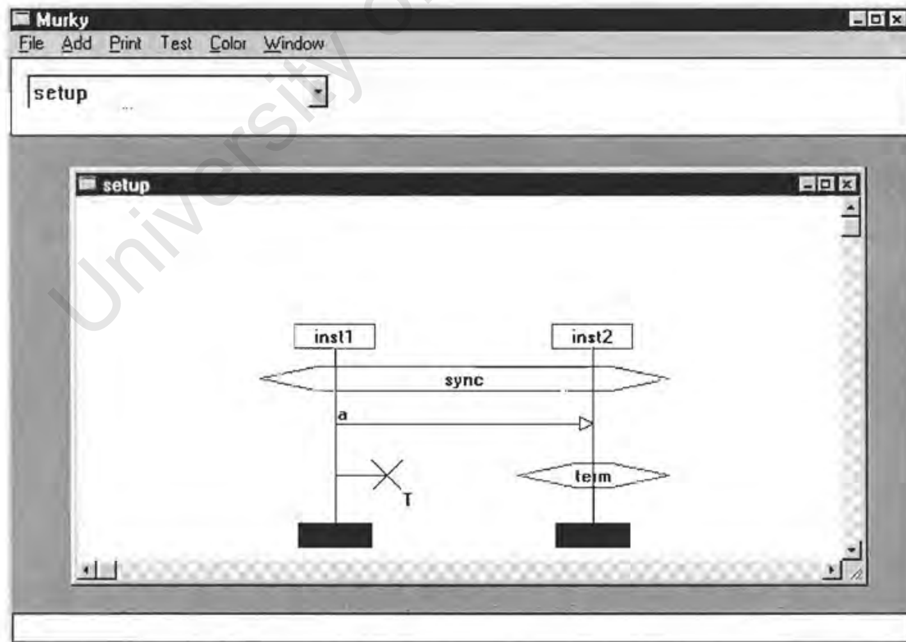


Figure 17: Condition Representation

University of Cape Town

6.4.8 Coregions

A coregion is a section of an instance axis where events are not strictly ordered. A coregion can be added by selecting the coregion option from the pull-down menu and then selecting the first and last events on the relevant axis. The coregion structure is then drawn in to reflect the choice.

6.4.9 High Level Structures

The use of these structures allows a user to combine MSCs in a representation similar to a flow graph. MSCs defined elsewhere in the MSC document can be referenced and then combined using sequential, alternate and parallel composition operators.

Adding a structure requires the user to select the add option, select the type of structure to be added and then select the manner in which it is added.

6.5 Bisimulation Testing

The bisimulation process is totally automated. The user is asked to select any two MSCs in the currently loaded document and then does the conversion to FSM. The bisimulation algorithm of Paige/Tarjan[RR87] is then run and the user is informed whether the respective start states are equivalent.

6.6 Printing

Any MSC that can be display in a window can be printed. This gives the designer or tester a hardcopy in addition to the textual representation.

6.7 Summary

The tool developed, allows for limited editing and specification of MSCs via the graphical representation, while illustrating the viability of bisimulation automation.

University of Cape Town

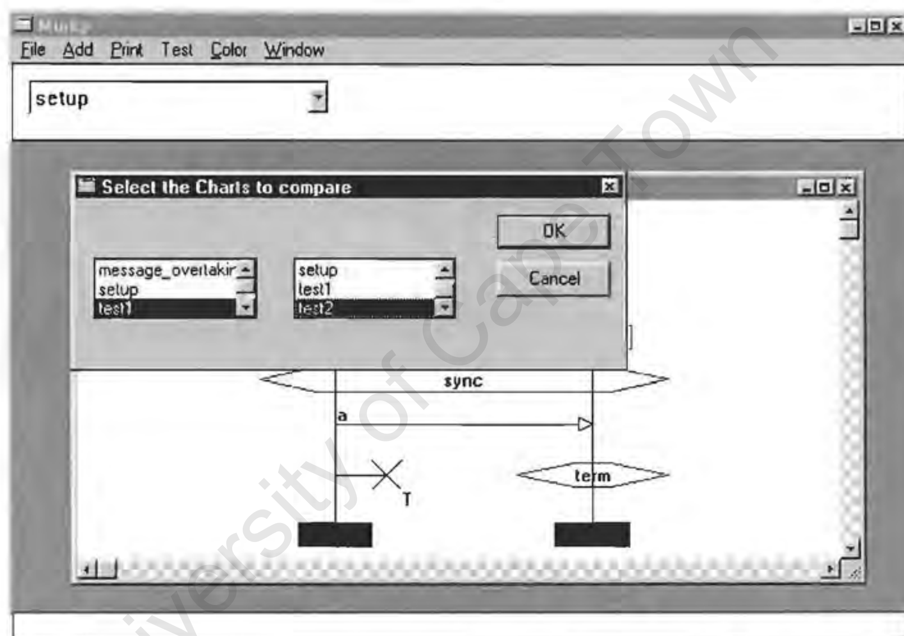


Figure 18: Selecting Charts for the Bisimulation Test

University of Cape Town

Chapter 7

Examples

7.1 Introduction

The chapter contains two examples, showing the step-by-step application of bisimulation to MSCs. Conversion to FSM is achieved using a self-written recursive algorithm. The bisimulation algorithm itself, is an implementation of the Paige/Tarjan [RR87] algorithm, with extensive use of the Fernandez description [Fer90]. Both examples were generated and tested using a bisimulation automation tool constructed during the course of this study. This tool is described in greater detail in the previous chapter.

7.2 Example 1

7.2.1 Problem Description

The MSCs that will be tested for equivalence are provided below in textual (Figure 20) and graphical format (Figure 19). Both MSCs contain actions labelled 'b' and a message 'a' from 'inst1' to 'inst2'. The only difference is an additional occurrence of action 'b' on 'inst2' that MSC 'test_case1' contains and which 'test_case2' does not. The purpose of this example is therefore to show that these MSCs are not equivalent.

University of Cape Town

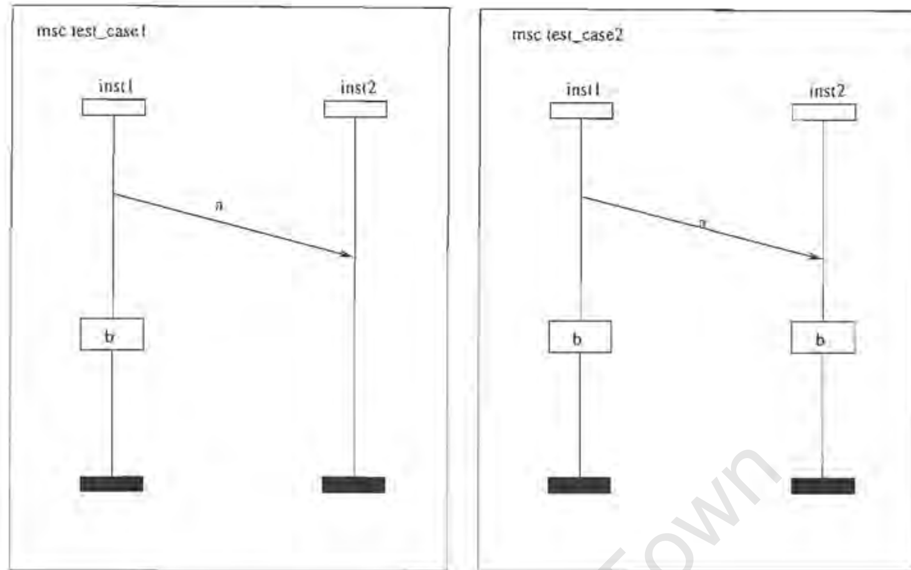


Figure 19: Graphical Specification of Test Case

7.2.2 Conversion to Finite State Machine

The graphical MSC representation is converted to a finite state machine as seen in Fig 21. States of the MSCs have no representation under MSC, and therefore a unique identifying number is assigned during conversion. This particular example gives two FSMs with states numbered from 0 to 15. The starting states for each FSM are assigned 0 and 6 respectively.

Transitions exist for each of the orderable events in the MSCs. Transitions exist for 'out a to inst2', 'in a from inst1' and 'action a'. No transition is needed for the instance header or end.

7.2.3 Bisimulation Algorithm

The bisimulation test uses the implementation described by Fernandez. As described in the algorithm given in the previous chapter, two partitions are used. A partition 'Q' contains the refined partition and 'W' contains sets to be used for further refinement of partition Q.

The steps of the algorithm are

- select and remove a set from W
- for each action

University of Cape Town

```
mscdocument bisimulation_example;

msc test_case1;
   instance inst1;
      out a to inst2;
      action 'b';
   endinstance;

   instance inst2;
      in a from inst1;
      action 'b';
   endinstance;
endmsc;

msc test_case2;
   instance inst1;
      out a to inst2;
      action 'a';
   endinstance;

   instance inst2;
      in a from inst1;
   endinstance;
endmsc;
```

Figure 20: Textual Specification of Test Case

University of Cape Town

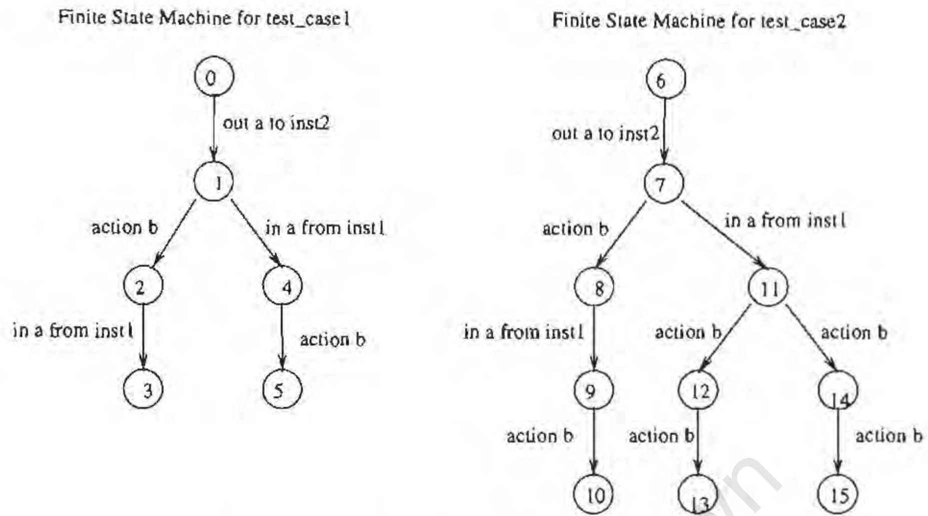


Figure 21: Finite State Machines Generated

- refine Q based on the set selected
- update Q and W

The states of W and Q for each step of the algorithm are as follows

Initially Q contains [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
and W contains [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

University of Cape Town

Iteration	Q	W	Refinement block	Action
1	[0,1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15]	[0,1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15]		
			[0,1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15]	
	[0,6] [1,2,3,4,5,7,8, 9,10,11,12,13,14,15]			1
	[2,3,5,8,10,13,15] [0,6] [1,4,7,9,11,12,14]			8
	[3,5,10,13,15] [0,6] [2,8] [4,9,11,12,14] [1,7]			0
2		(([3,5,10,13,15], [2,8]), ([4,9,11,12,14], [1,7])), [0,6])	(([3,5,10,13,15], [2,8]), ([4,9,11,12,14], [1,7])), [0,6])	
				1
				8
				0
3		(([3,5,10,13,15], [2,8]), ([4,9,11,12,14], [1,7]))	(([3,5,10,13,15], [2,8]), ([4,9,11,12,14], [1,7]))	
				1
	[3,5,10,13,15] [0,6] [4,9,12,14] [2,8] [1,7] [11]			8
	[3,5,10,13,15] [0,6] [4,9,12,14] [2] [8] [1,7] [11]			0
4		(([3,5,10,13,15],([2], [8])) (([4,9,12,14], [11]), [17])	(([3,5,10,13,15],([2], [8]))	
				1
				8
				0
5		(([4,9,12,14], [11]), [1,7]) ([2], [8])	(([4,9,12,14], [11]), [1,7])	
				1
				8
				0

University of Cape Town

Iteration	Q	W	Refinement block	Action
6		([2], [8]) ([4,9,12,14], [11])	([2], [8])	
				1
	[3,5,10,13,15] [0,6] [4,9,12,14] [2] [8] [1] [7] [11]			8
				0
7		([4,9,12,14], [11]) ([1], [7])	([4,9,12,14], [11])	
				1
				8
				0
8		([1], [7])	([1], [7])	
	[3,5,10,13,15] [0] [6] [4,9,12,14] [2] [8] [1] [7] [11]			1
				8
				0
9		([0], [6])	([0], [6])	
				1
				8
				0
10	[3,5,10,13,15] [0] [6] [4,9,12,14] [2] [8] [1] [7] [11]	Empty set		

7.2.4 Refinement Table

This table shows how a partition Q is refined using the Paige/Tarjan algorithm. Sets are enclosed by square brackets, sets of sets by normal brackets. [0,6] therefore represents a set containing two elements, namely '0' and '6'. The set ([0], [6]) indicates two sets which are subsets of their union. Sets of sets are used by the algorithm to reduce the complexity of determining bisimulation.

The algorithm starts with partition Q containing all states, the partition is refined for each

University of Cape Town

set of partition W. Each set is refined across the partition for each action. Actions are indicated in this table by the integers 0, for the output, 1 for the input and 8 for the action. The algorithm generates new sets for W, which are in turn applied to Q for each action of the FSM.

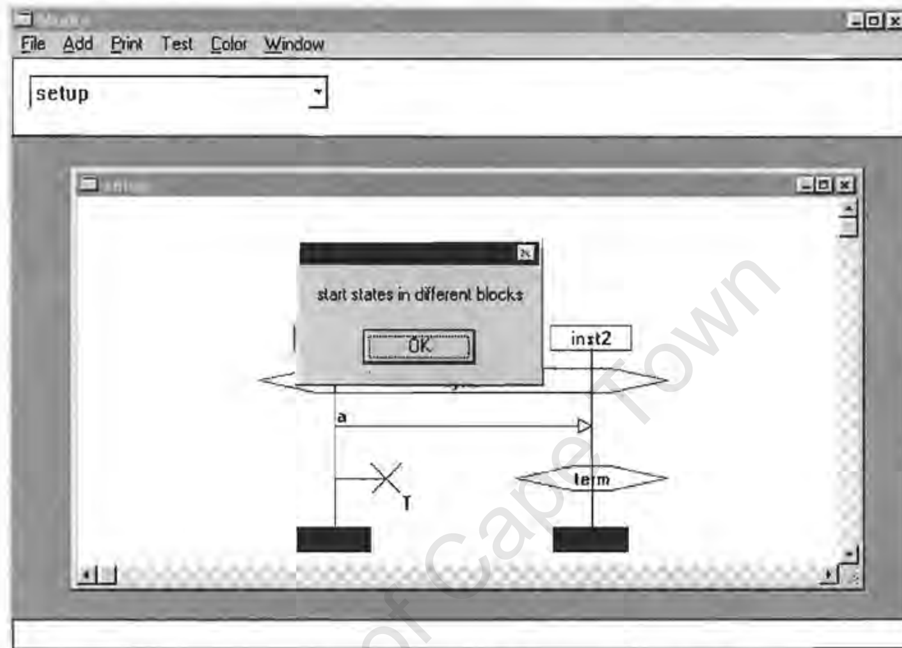


Figure 22: Result of the Bisimulation Test

7.2.5 Results

The algorithm took 9 refinement steps to find the coarsest partition. Each of these refinement steps requires a refinement for each action. It is therefore desirable to limit the number of actions where possible.

The state of Q over each of these iterations shows that the initial steps effect the most changes. The algorithm could have been terminated one iteration earlier, when the start states (0, 6) were split. This implies that the algorithm found a sequence of steps possible from one start state that was not possible from another. Similarly, other single state sets of Q hold the same properties. Event sequences from these states can then be examined to identify the differences between the MSCs.

University of Cape Town

7.3 Example 2

7.3.1 Problem description

This example illustrates the concepts of delayed choice and its impact on equivalence testing. Two high-level MSCs are compared using bisimulation and trace equivalence. The high-level MSCs are provided in textual format in Figure 23 and in the graphical representation in Figure 24.

Figure 24 illustrates the difference between the high-level MSCs named miscon1 and miscon2. Miscon1 consists of the sequential composition of *msc1* and the alternate choice of *msc2* and *msc3*. Miscon2 however allows the alternate choice of the sequential composition of *msc1.msc2* and *msc1.msc3*.

Although the same sequences of events are possible in both high-level MSCs, the equivalence tests give different results.

7.3.2 Conversion to Finite State Machine

The conversion from high-level MSCs to FSM is done as described in Chapter 4.

A shorthand notation is used to represent transitions. The conversion is given in Figure 25.

The resulting FSMs are given in Figures 26 and 27. From these figures it can be seen that the finite state machine generated for trace equivalence contains no non-deterministic states. As states are created, numbers are assigned to them. From the figure it can be seen that a total of 78 states were generated for the DFA and only 44 for the NDFA. Although modifications to the algorithms can reuse older states, this value gives an indication of how many states are used during the conversion from non-determinism to determinism.

7.3.3 Bisimulation Algorithm

The bisimulation algorithm is applied as before, with the initial partitions once again set to the union of all states. No further differences exist between the implementation of trace equivalence and bisimulation.

7.3.4 Refinement Table

Both refinement tables are given for comparison purposes. The first table is generated by the deterministic FSM, the second by the NDFA.

University of Cape Town

```
mscdocument example2;
   msc  msc3  ;
      instance  inst1  ;
         action 'c'  ;
      endinstance  ;
      instance  inst2  ;
           out  a  to lost  inst2  ;
      endinstance ;
   endmsc  ;
   msc  msc2  ;
      instance  inst1  ;
         in  b  from  env  ;
         condition  Sync  shared  inst1  ,  inst2  ;
      endinstance  ;
      instance  inst2  ;
         condition  Sync  shared  inst1  ,  inst2  ;
      endinstance ;
   endmsc  ;
   msc  msc1  ;
      instance  inst1  ;
         out  a  to  inst2  ;
      endinstance  ;
      instance  inst2  ;
         set T , inst2  ;
         in  a  from  inst1  ;
         timeout T , inst2  ;
      endinstance ;
   endmsc  ;
   msc  miscon2  ;
      expr L1 alt L2 ;
      L1 : msc1 seq  ( L3 )  ;
      L2 : msc1 seq  ( L4 )  ;
      L3 : msc2 seq  ( L5 )  ;
      L4 : msc3 seq  ( L5 )  ;
      L5 :  end  ;
   endmsc  ;
   msc  miscon  ;
     expr L1 ;
     L1 : msc1 seq  ( L2 alt L3 )  ;
     L2 : msc2 seq  ( L4 )  ;
     L3 : msc3 seq  ( L4 )  ;
     L4 :  end  ;
   endmsc  ;
```

Figure 23: Textual MSC representation of Example2

University of Cape Town

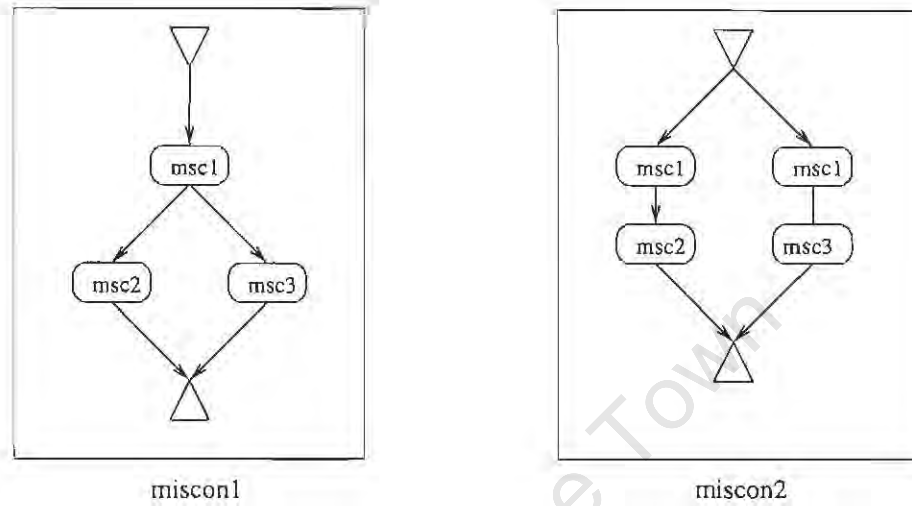


Figure 24: High-level MSC representation of Example2

University of Cape Town

Transition	Value
out a to inst2	1
set T , inst2	5
in a from inst1	0
timeout T , inst2	6
in b from env	11
out a to lost inst2	3
action 'c'	8

Figure 25: Key for Example2 Refinement table

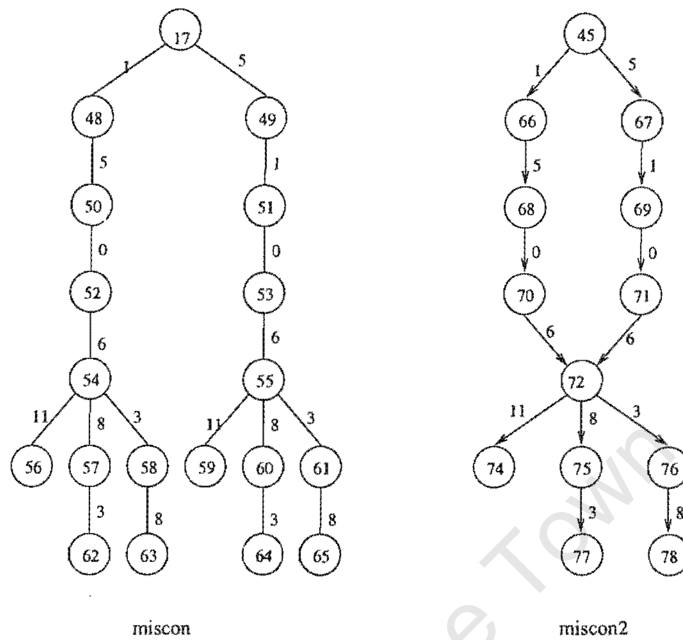


Figure 26: Finite State Machines Generated for Trace Equivalence

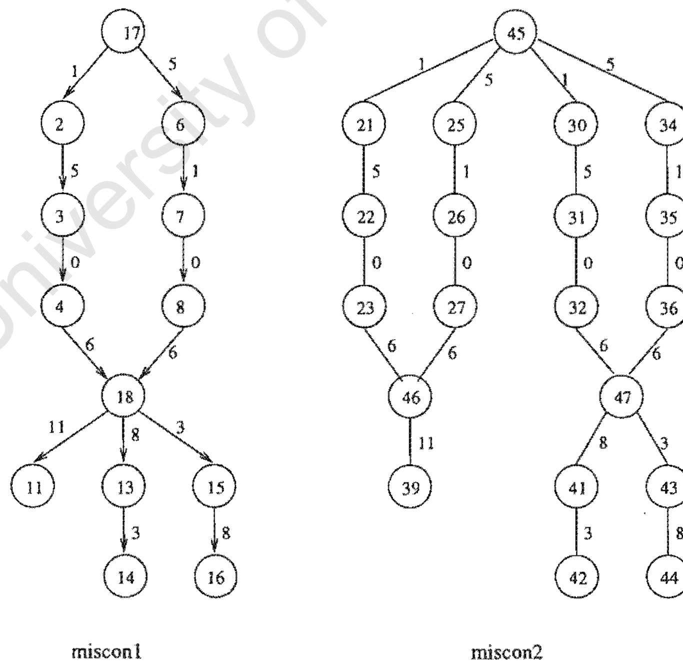


Figure 27: Finite State Machines Generated for Bisimulation

University of Cape Town

Iteration	Q	W	Refinement block	Action
1	[17,48,49,50,51,52,53,54, 55,56,57,58,59,60,61,62, 63,64,65,45,66,67,68,69, 70,71,72,74,75,76,77,78]	[17,48,49,50,51,52,53,54, 55,56,57,58,59,60,61,62, 63,64,65,45,66,67,68,69, 70,71,72,74,75,76,77,78]		
	[48,50,51,52,53,54,55, 56,57,58,59,60,61,62,63, 64,65,66,68,69,70,71,72, 74,75,76,77,78] [17,49,45,67]			1
	[50,51,52,53,54,55,56,57, 58,59,60,61,62,63,64,65, 68,69,70,71,72,74,75,76, 77,78] [49,67] [17,45] [48,66]			5
	[52,53,54,55,56,57,58,59, 60,61,62,63,64,65,70,71, 72,74,75,76,77,78] [49,67] [17,45] [48,66] [50,51,68,69]			0
	[54,55,56,57,58,59,60,61, 62,63,64,65,72,74,75,76, 77,78] [49,67] [17,45] [48,66] [50,51,68,69] [52,53,70,71]			6
	[56,57,58,59,60,61,62,63, 64,65,74,75,76,77,78] [49,67] [17,45] [48,66] [50,51,68,69] [52,53,70,71] [54,55,72]			11
	[56,57,59,60,62,63,64,65, 74,75,77,78] [49,67] [17,45] [48,66] [50,51,68,69] [52,53,70,71] [54,55,72] [58,61,76]			8
	[56,59,62,63,64,65,74,77, 78] [49,67] [17,45] [48,66] [50,51,68,69] [52,53,70,71] [54,55,72] [58,61,76] [57,60,75]			3

University of Cape Town

Iteration	Q	W	Refinement block	Action
2	[56,57,59,60,62,63,64,65, 74,75,77,78] [49,67] [17,45] [48,66] [50,51,68,69]	[[[[[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]], [54,55,72]], [52,53,70,71]], [50,51,68,69]], [48,66]], [[49,67], [17,45]]]		
				1
				5
				0
				6
				11
				8
				3
3		[[[[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]], [54,55,72]], [52,53,70,71]], [50,51,68,69]], [48,66]] [[49,67], [17,45]]	[[56,59,62,63,64,65, 74,77,78], [57,60,75]]	
				1
				5
				0
				6
				11
				8
				3
4		[[49,67]; [17,45]] [[[[[56,59,62,63,64, 65,74,77,78], [57,60,75]], [58,61,76]], [54,55,72]], [52,53,70,71]], [50,51,68,69]]		
				1
				5
				0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
5		[[[[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]], [54,55,72]], [52,53,70,71]], [50,51,68,69]]		
				1
				5
				0
				6
				11
				8
				3
6		[[[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]], [54,55,72]], [52,53,70,71]]		
				1
				5
				0
				6
				11
				8
				3
7		[[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]], [54,55,72]]		
				1
				5
				0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
8		[[[56,59,62,63,64,65, 74,77,78], [57,60,75]], [58,61,76]]		1
				5
				0
				6
				11
				8
				3
9		[[56,59,62,63,64,65, 74,77,78], [57,60,75]]		1
				5
				0
				6
				11
				8
				3
10	[56,59,62,63,64,65,74, 77,78] [17,45] [49,67] [48,66] [50,51,68,69] [52,53,70,71] [58,61,76] [54,55,72] [57,60,75]	Empty set		

University of Cape Town

Iteration	Q	W	Refinement block	Action
	[2,3,4,6,7,8,11,13,14, 15,16,17,18,21,22,23,25, 26,27,30,31,32,34,35,36, 39,41,42,43,44,45,46,47]	[2,3,4,6,7,8,11,13,14, 15,16,17,18,21,22,23,25, 26,27,30,31,32,34,35,36, 39,41,42,43,44,45,46,47]		0
1	[2,3,4,7,8,11,13,14, 15,16,18,21,22,23,26,27, 30,31,32,35,36,39,41,42, 43,44,46,47] [17,6,45,25,34]			1
	[3,4,7,8,11,13,14,15,16, 18,22,23,26,27,31,32,35, 36,39,41,42,43,44,46,47] [6,25,34] [2,21,30] [17,45]			5
	[4,8,11,13,14,15,16,18, 23,27,32,36,39,41,42,43, 44,46,47] [6,25,34] [2,21,30] [17,45] [3,7,22,26,31,35]			0
	[11,13,14,15,16,18,39,41, 42,43,44,46,47] [6,25,34] [2,21,30] [17,45] [3,7,22,26,31,35] [4,8,23,27,32,36]			6
	[11,13,14,15,16,39,41,42, 43,44,47] [6,25,34] [2,21,30] [17,45] [18,46] [3,7,22,26,31,35] [4,8,23,27,32,36]			11
	[11,13,14,16,39,41,42,44] [6,25,34] [17,45] [2,21,30] [46] [18] [3,7,22,26,31,35] [4,8,23,27,32,36] [15,47,43]			8
	[11,14,16,39,42,44] [6,25,34] [17,45] [46] [18] [2,21,30] [13,41] [3,7,22,26,31,35] [47] [4,8,23,27,32,36] [15,43]			3

University of Cape Town

Iteration	Q	W	Refinement block	Action
2		[[[[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]], [4,8,23,27,32,36]], [3,7,22,26,31,35]], [2,21,30]], [[6,25,34], [17,45]]]		1
				5
				0
				6
				11
				8
				3
3		[[[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]], [4,8,23,27,32,36]], [3,7,22,26,31,35]], [2,21,30]] [[6,25,34], [17,45]]		1
				5
				0
				6
				11
				8
				3
4		[[6,25,34], [17,45]] [[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]], [4,8,23,27,32,36]], [3,7,22,26,31,35]]		1
				5
				0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
5		[[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]], [4,8,23,27,32,36]], [3,7,22,26,31,35]]		1
				5
				0
				6
				11
				8
				3
6		[[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]], [4,8,23,27,32,36]]		1
				5
				0
				6
				0
				11
				8
				3
7		[[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]], [[46], [18]]]		1
				5
				0
	[11,14,16,39,42,44] [17,45] [6,25,34] [2,21,30] [3,7,22,26,31,35] [46] [15,43] [13,41] [18] [47] [32,36] [4,8,23,27]			6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
8		[[[11,14,16,39,42,44], [13,41]], [[15,43], [47]]] [[46], [18]] [[32,36], [4,8,23,27]]		1
				5
				0
				6
				11
				8
				3
9		[[46], [18]] [[32,36], [4,8,23,27]] [[11,14,16,39,42,44], [13,41]] [[15,43], [47]]		1
				5
				0
	[11,14,16,39,42,44] [17,45] [6,25,34] [2,21,30] [3,7,22,26,31,35] [32,36] [46] [15,43] [13,41] [18] [47] [23,27] [4,8]			6
				11
				8
				3
10		[[32,36], [[23,27], [4,8]]] [[11,14,16,39,42,44], [13,41]] [[15,43], [47]]		1
				5
	[11,14,16,39,42,44] [17,45] [6,25,34] [2,21,30] [32,36] [46] [15,43] [13,41] [18] [47] [23,27] [4,8] [31,35] [3,7,22,26]			0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
11		[[11,14,16,39,42,44], [13,41]] [[15,43], [47]] [[23,27], [4,8]] [[31,35], [3,7,22,26]]		1
				5
				0
				6
				11
				8
				3
12		[[15,43], [47]] [[23,27], [4,8]] [[31,35], [3,7,22,26]]		1
				5
				0
				6
				11
				8
				3
13		[[23,27], [4,8]] [[31,35], [3,7,22,26]]		1
				5
	[11,14,16,39,42,44] [17,45] [6,25,34] [2,21,30] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7]			0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
14	[11,14,16,39,42,44] [17,45] [2,21,30] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7] [34] [6,25]	[[31,35], [[22,26], [3,7]]]		1
				5
				0
				6
				11
				8
				3
15	[11,14,16,39,42,44] [17,45] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7] [34] [30] [2,21] [25] [6]	[[22,26], [3,7]] [[34], [6,25]] [[30], [2,21]]		1
	[11,14,16,39,42,44] [17,45] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7] [34] [30] [25] [6] [21] [2]			5
				0
				6
				11
				8
				3
16		[[34], [[25], [6]]] [[30], [[21], [2]]]		1
	[11,14,16,39,42,44] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7] [34] [30] [25] [6] [21] [2] [17] [45]			5
				0
				6
				11
				8
				3

University of Cape Town

Iteration	Q	W	Refinement block	Action
17		[[30], [[21], [2]]] [[25], [6]] [[17], [45]]		1
				5
				0
				6
				11
				8
				3
18		[[25], [6]] [[17], [45]] [[21], [2]]		1
				5
				0
				6
				11
				8
				3
19		[[17], [45]] [[21], [2]]		1
				5
				0
				6
				11
				8
				3
20		[[21], [2]]		1
				5
				0
				6
				11
				8
				3
21	[11,14,16,39,42,44] [23,27] [4,8] [31,35] [46] [15,43] [13,41] [32,36] [18] [47] [22,26] [3,7] [34] [30] [25] [6] [21] [2] [45] [17]	Empty set		

University of Cape Town

7.3.5 Results

The first refinement table illustrates the effect of the algorithm on trace equivalence. The coarsest refined partition is found after 10 iterations. The second refinement table was generated for the non-deterministic FSM and gives bisimulation equivalence. The algorithm takes 21 iterations to find the coarsest partition and could terminated after 16 iterations, when the start states of the two FSMs were placed in separated blocks.

University of Cape Town

Chapter 8

Conclusion

8.1 Introduction

The purpose of this study was the investigation of Message Sequence Charts (MSCs) as a specification technique and the use of bisimulation in testing systems for equivalence. MSCs describe the communication between system components in a graphical form. Before a formal description was accepted various forms of MSCs existed. A formal specification for MSCs was therefore important to standardize specification and facilitate exchange.

8.2 Concerns

The MSC specification however is lacking in several regards. Of major concern is ambiguity in the specification [PL95]. Future revisions to the specification will hopefully reduce these. A further problem is that MSC messages are asynchronous. Describing synchronous message exchange is simply not catered for. A global system clock would be a possible solution to this, but this changes the emphasis of MSCs.

Another concern is the lack of any way for specifying the co-ordinates of the graphical layout of a specification. This particular problem has been sidestepped in some quarters by storing layout information in the form of textual comments. As this is not part of the formal specification, not all tools sharing these MSCs can make use of this layout information.

All of these factors have limited the support and use of MSCs by software engineers notwithstanding the fact MSCs in their various forms predate most other specification techniques.

University of Cape Town

8.3 MSC and System design

System design can be simplified and accelerated by means of formal description techniques. These allow a system to be described and tested for correctness before implementation. Popular FDTs include SDL, LOTOS and ESTELLE. Of these, SDL in particular, makes use of MSCs to aid the designer by illustrating system communication. MSCs are also used in conjunction with FDTs to check the consistency between an MSC specification and the system as described by the FDT.

Design based solely on MSCs is limited by the lack of tools providing the same facilities found for other FDTs. The formal definition for MSCs provides access to a wealth of theory, which has gone largely unused.

8.4 MSCs and Bisimulation

This study investigated bisimulation as a verification and validation technique for MSCs.

System design should take a number of factors into consideration before attempting to apply bisimulation. The crucial factor is the impact of branch time and linear time equivalence. Bisimulation tests branch time equivalence, which is stricter than linear time or trace equivalence. Designers should therefore avoid creating non-deterministic branching structures where possible. Another consideration is the looping construct of MSCs. The looping construct in conjunction with other constructs such as parallel composition can not be efficiently converted to a finite state representation. A loop in particular would require either the unrolling of the loop, which is impracticle for large loops, or an to the FSM using variables. Designers should either strictly limit the looping construct or avoid it completely.

University of Cape Town

Implementation of Bisimulation on MSCs

The steps in applying bisimulation testing include converting the MSC to a finite state machine and then applying an appropriate algorithm. The algorithm used to convert the MSC is recursive and therefore slow for large systems. A big impact would be made on the speed and efficiency of the whole algorithm if state space reduction was implemented. A small increase in the size of an MSC can be responsible for a state space explosion. Until this is done, the bisimulation process is limited to fairly small MSC systems.

The algorithm used is the Paige/Tarjan[RR87] bisimulation algorithm. It has complexity O(mlogn) with m transitions and n states. Extensive use was made of a description of this

algorithm given by Fernandez which extends the algorithm to allow more than one type of transition without an additional increase in complexity.

Automation of Bisimulation

The FSM conversion and bisimulation algorithm were implemented in the form of a simple tool. The tool consists of two components, namely an MSC editor and the bisimulation algorithm. It provides the beginnings of a full specification and testing tool for MSCs. The implementation code of the algorithm is given at the end.

The editor allows the user to graphically design the MSC. All orderable events are catered for, and the design process ensures the consistency of resulting MSCs. It is therefore impossible to design an MSC that is semantically or syntactically incorrect. The tool accepts an instance oriented textual description or overview of an MSC specification in the form of a high level MSC. The event-oriented description, inline events and referencing are not presently allowed. These constructs can be implemented using equivalent high level constructs and the instance orientated technique.

Applying the bisimulation test to MSCs designed by the editor is a simple matter of selecting the MSCs needing comparison. The whole testing process is automated and the result is reported to the user. This tool illustrates the usefulness of bisimulation testing on systems designed for this purpose.

University of Cape Town

8.5 Extensions and Changes

8.5.1 Editor

The editor exists in the form of a beta version. Before it reaches the status of a commercial tool, context sensitive help, a more intuitive interface, better manipulation of MSCs and support of the full MSC specification would be a requirement.

The most important of these extensions is to allow the alternate textual representation of event oriented MSCs. This will allow the use of inline expressions.

8.5.2 Finite State Machine Conversion

The conversion algorithm consists of two stages. The initial stage translates MSCs to FSMs. The second stage examines the high level MSC to combine individual MSCs using

the operators of the high level MSC. This process can result in a huge number of states for a small number of events. Finding ways to reduce the state space is therefore of importance.

8.6 Summary

It is quite feasible to apply bisimulation testing to MSCs. This is illustrated by a tool developed during the course of this project, which allows graphical design of MSCs and testing of generated MSCs.

It is not however appropriate to apply bisimulation testing to any two MSCs. The system designer should ensure that it is bisimulation rather than trace equivalence under consideration.

The bisimulation algorithm used has the best complexity O(mlogn) for this type of test . For large MSC specifications the time required to test systems will become impractical unless the state space is collapsed.

University of Cape Town

Bibliography

[A.A85] J.Ullman A.Aho, R.Sethi. *Compilers: Principle, Techniques and Tools.* Addison-Wesley Publishing, 1985.

[BW90] J.C.M Baeton and W.P.Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, 18, 1990.

[C.S89] C.Stirling. Introduction to modal and temporal logics for CCS. *Concurrency: Theory, Language and Architecture, Lecture Notes Computer Science*, 391, 1989.

[D.O95] D.Ouimet. *Spin, Promela.* http://wwwtios.cs.utwente.nl/lnews/lnews03/node29.html, 1995.

[EJ96] P.Graubmann E.Rudolph and J.Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[Fer90] J.C Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1990.

[F.H85] F.Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[G.H] G.Holzmann. *Using SPIN.* http://plan9.bell-labs.com/plan9/doc/spin.html.

[G.L92] G.Leduc. A framework based on implementation relations for implementing lotos specification. *Computer Networks and ISDN Systems*, 25:23–41, 1992.

[I.S89] I.Sommerville. *Software engineering.* Addison-Wesley, 1989.

[IT88] ITU-TS. *ITU-TS Recommendation Z.100: Specification and Description Language(SDL).* ITU-TS, 1988.

[IT93] ITU-T. *ITU-T Recommendation Z.120 : Message Sequence Charts (MSC).* ITU-T, 1993.

University of Cape Town

[J.H69] J.Hopcroft. *Formal Languages and their relation to Automata.* Addison-Wesley, 1969.

[JS95] J.C.M.Baeten and S.Mauw. Delayed choice: an operator for joining message sequence charts. In D.Hogrefe and S.Leue, editors, *Formal Description Techniques VII*, pages 340–354. Chapman and Hall, 1995.

[lab98] Verimag laboratory. *CADP.* http://www.inrialpes.fr/vasy/pub/cadp.html, 1998.

[MA90] M.Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[M.A94] M.A.Reniers. *Syntax Requirements of Message Sequence Charts.* Study Group Meeting SG10, Geneva, TD59, 1994.

[Pet81] J.L. Peterson. *Petri Nets and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[PL95] P.Ladkin and S Leue. Comments on a proposed semantics for Basic Message Sequence Charts. *The Computer Journal*, 37(9), 1995.

[PS90] P.Kanellakis and S.Smolka. CCS expressions, finite state processes and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[Rei92] W. Reisig. *A Primer in Petri Net Design.* Springer-Verlag, 1992.

[R.M80] R.Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[RR87] R.Paige and R.Tarjan. Three partiton refinement algorithms. *Siam Journal of Computing*, 16(6):973–989, 1987.

[RR95] R.Paige and R.Tarjan. On the computational complexity of bisimulation. *ACM Computing Surveys*, 27(2):287–289, 1995.

[S.B] S.Budkowski. *ESTELLE Development Toolset (EDT).* http://alix.int-evry.fr/ stan/edt.html.

[S.B87] P.Dembinski S.Budkowski. An introduction to estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[S.M96] S.Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996.

University of Cape Town

[S.S] S.Smolka. *ACM SDCR Workshop, Draft of Concurrency Working Group Report.* http://www.cs.sunysb.edu/ sas/sdcr/report/draft2.html.

[Tel98] Telelogic. *SDT Toolset.* http://www.telelogic.se, 1998.

[V.E93] V.Encontre. *Geode Simulator Reference Manual.* Verilog, Toulouse, France, 1993.

[vG96] R.J. van Glabbeek. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[W.C81] W.Clinger. Foundation of actor semantics. Technical Report AI-TR-633, MIT A.I.Laboratory, Cambridge, MA, 1981.

University of Cape Town

Appendix A

Encoded Bisimulation Algorithm

University of Cape Town

```
#ifndef __TRANSITION_PAIR
#define __TRANSITION_PAIR

#include"list.hpp"

class block;
class transition_pair;
class transition;
class partition;
class state;

class transition_pair
{
   private:
      transition *p_transition; //the R in xRy
      Linked_List<state *> *p_ll_state; //the x in xRy

   public:
      transition_pair(transition *p_t, state *p_s);

      Linked_List<state*> *give_ll_state(void);
      transition *give_transition(void);
      void display();
```

```
      void write(FILE *fp){};
};
#endif


#ifndef __TRANSITION
#define __TRANSITION

class transition
{
   private:
      instance_event *p_instance_event; //contains the event

   public:
      transition(instance_event *i);
      instance_event* give_instance_event(); //give the action
      void display();
      void write(FILE *fp){};
};

#endif __TRANSITION


#ifndef __STATE
#define __STATE

#include "list.hpp"
#include "stdlib.h"
#include "mscclass.h"
class block;
class transition_pair;
class transition;
class partition;
class state;
```

University of Cape Town

```
class state
{
   private:
      //states that map into this state
      Linked_List<transition_pair *> *p_ll_transition_pair;
      //states that this state maps to
      Linked_List<transition_pair *> *p_ll_to_transition_pair;
      //holds all the states making up this one
      Linked_List<state *> *p_ll_condense_state;

      block *p_block; //the block this state is in
      int id;
      int b, b1;

   public:
      state(block *p_b, int i);
      state(int i);

      void move_state(block *p_dest);
      void add_transition(instance_event *p_ie, state *p_s);
      void add_to_transition(instance_event *p_ie, state *p_s);

      block *give_block(void);
      void set_block(block *p_b);
      Linked_List<transition_pair *> *give_ll_transition_pair();
      Linked_List<transition_pair *> *give_ll_to_transition_pair();
      Linked_List<state *> *give_ll_condense_state();
      void set_ll_condense_state(Linked_List<state *> *p_ll);

      int mark;
      int give_info();
      transition_pair *find_transition_pair(instance_event *p_ie);
      transition_pair *find_to_transition_pair(instance_event *p_ie);
      void increment_b();
      void increment_b1();
      void reset_info();
```

University of Cape Town

```
      int give_id();

      void display();
      void write(FILE *fp){};
};

#endif

#ifndef __BLOCK
#define __BLOCK
#include "mscclass.h"
#include "list.hpp"

class block;
class transition_pair;
class transition;
class partition;
class state;

enum splitter_type { SIMPLE=0, COMPLEX_LEAF, COMPLEX_NODE, ABSENT};

class block
{
   private:                        //used when this is a complex_node
      block *p_X1,
            *p_X2,
            *p_parent_block,        //used when COMPLEX NODE or LEAF
            *p_associated_block;    //used when refining a simple splitter
      splitter_type type;
      Linked_List<state *> *p_ll_state;
      Linked_List<block *> *p_ll_block_info;


   public:
      block(); //used to create a simple splitter
      block(Linked_List<state *> *p_ll);  //create a block when we have a ll
```

University of Cape Town

```
    block(Linked_List<state *> *p_ll_X1, Linked_List<state *> *p_ll_X2);
    block(block *X1, block *X2);

    void set_child(block *old_child, block *new_child);

    block *give_associated_block(void);
    void set_associated_block(block *p_b);

    block *give_parent_block();
    void set_parent_block(block *p_b);

    void set_type(splitter_type st);
    splitter_type give_type(void);

    block *give_X1(void);
    block *give_X2(void);

    Linked_List<state *> *give_ll_state();
    Linked_List<state *> *give_all_states();
    Linked_List<block *> *give_ll_block_info();

    int give_count(void);
    void set_info(instance_event *);
    void reset_info(void);

    void increment_states(Linked_List<state *> *p_ll, int choice);
    void consistent_block();
    void display();
    void write(FILE *fp){};
};

#endif


#ifndef __PARTITION
#define __PARTITION
```

University of Cape Town

```
#include "list.hpp"

class block;
class transition_pair;
class transition;
class partition;
class state;


class partition
{
   private:
      Linked_List<block *> *p_ll_block; //all blocks in the partition

   public:
      partition(block *p_b);
      partition();

      Linked_List<block *> *give_ll_block(void);
      block *give_and_remove_block(void);
      void display(void);
      void write(FILE *fp);
};

#endif


partition *Q, *W;

void construct(block *X,block * X1,block * X2)
{
   switch(X->give_type())
   {
      case SIMPLE       :  W->give_ll_block()->remove(X);
      case ABSENT       :  {
```

University of Cape Town

```
                                    block *new_block=new block(X1, X2);
                                    W->give_ll_block()->add(new_block);
                                    break;
                                 }
        case COMPLEX_LEAF :      {
                                    block *p_parent=X->give_parent_block();
                                    //create a new block
                                    block *new_block=new block(X1, X2);
                                    //update all pointers
                                    p_parent->set_child(X, new_block);
                                    //remove X from everywhere
                                    break;
                                 }
        default              :   break;
    }
};


void algorithm(Linked_List<transition *> *p_ll_transition, block *p_block)
{

    //setup finite state machine

    //initialise structures
    Q=new partition(p_block); //create Partition Q (block contains all elements)
    W=new partition(p_block); //for splitter partition init

    block *B;
    block *B_split;

    while (W->give_ll_block()->give_first()!=NULL)
    {
        B=W->give_and_remove_block(); //Select block from W

        if (B->give_type()==SIMPLE)
        {
```

University of Cape Town

```
B->set_type(ABSENT);
//Copying the linked list of states to a new linked list
B_split=new block(B->give_all_states());
B_split->set_type(ABSENT);

//loop through all the actions in the action list
for(Link<transition *> *p_l_transition=p_ll_transition->give_first();
   p_l_transition!=NULL;
   p_l_transition=p_l_transition->give_next())
{
   instance_event *p_ie=
p_l_transition->give_object()->give_instance_event();
   Link<state *> *p_l_state=B_split->give_ll_state()->give_first();
   Linked_List<block *> *I=new Linked_List<block *>;


   //loop through all states of B_split
   //Construct I
   while(p_l_state!=NULL)
   {
      //take the state (which has a linked list of transition pairs)
      //loop through its preset, looking for transition pair where the
      //transition is the same as the action up above

      transition_pair *p_tp=
p_l_state->give_object()->find_transition_pair(p_ie);

      //found a transition into this element of B, using action A
      if (p_tp!=NULL)
      {
         //loop through the elements of the tp's origin list
         Link<state *> *p_l_loop_state=
            p_tp->give_ll_state()->give_first();

         //for each element in its preset
         while(p_l_loop_state!=NULL)
```

University of Cape Town

```
        {
           //check if the block assoc with the element has any ASSOC
           if (p_l_loop_state->give_object()->
              give_block()->give_associated_block()!=NULL)
           {
              //if it does move the state to that block
              p_l_loop_state->give_object()->
                 move_state(p_l_loop_state->give_object()->
                 give_block()->give_associated_block());
           }
           else
           {  //if not,
              //create a new block,
              block *p_new_block=new block();
              //associate the old block with this new one
              p_l_loop_state->give_object()->give_block()->
                 set_associated_block(p_new_block);
              p_new_block->set_associated_block(p_l_loop_state->
                 give_object()->give_block());

              //add the new block to the partition
              I->add(p_new_block);
              //move the state to the new block
              p_l_loop_state->give_object()->move_state(p_new_block);
           };
           p_l_loop_state=p_l_loop_state->give_next();
        };
     };
     p_l_state=p_l_state->give_next();
  };

  //Update all structures, by looping through the blocks of I

  for(Link<block *> *p_l_loop_block=I->give_first();
     p_l_loop_block!=NULL;
     p_l_loop_block=I->give_first())
```

University of Cape Town

```
{
   block *X1=p_l_loop_block->give_object();
   block *X=p_l_loop_block->give_object()->give_associated_block();

   //Case where X=empty set (all elements in X1)
   if (X1->give_associated_block()->give_count()==0)
   {
   if (X1->give_associated_block()->give_count()==0)
   {

      X=X1->give_associated_block();

      //move the list of elements to the associated block
      //return elements moved previously
      X->give_ll_state()->set_first(X1->
         give_ll_state()->give_first());
      X->give_ll_state()->set_last(X1->give_ll_state()->
         give_last());

      //undo the association
      X->set_associated_block(NULL);
      //remove this block
      I->remove(X1);
   }
   else
   {
      //loop through the elements of X1, making them point to X1
      for(Link<state *> *p_l_loop_state=
         X1->give_ll_state()->give_first();
         p_l_loop_state!=NULL;
         p_l_loop_state=p_l_loop_state->give_next())
      {
         p_l_loop_state->give_object()->set_block(X1);
      };

      //updating X1 and X associations
```

University of Cape Town

```
X1->set_associated_block(NULL);
X->set_associated_block(NULL);

//remove X1 from I
I->remove(X1);

//add X1 to partition
X1->set_type(ABSENT);
Q->give_ll_block()->add(X1);

//update W as follows:
//if X simple in W, add X1 to W
if (X->give_type()==SIMPLE)
{
   X1->set_type(SIMPLE);
   W->give_ll_block()->add(X1);
}
else
{
   //if X is a leaf, replace by X->X12<-X1
   if (X->give_type()==COMPLEX_LEAF)
   {
      //collect parent
      block *p_parent=X->give_parent_block();
      //create X12
      block *X12=new block(X, X1);

      //add X12 into W
      p_parent->set_child(X, X12);
   }
   else //X is not in W at all
   {
      block *X12=new block(X,X1);
      W->give_ll_block()->add(X12);
   };
};
```

University of Cape Town

```
            }; //the case where X has been modified - fixed Q, W, I
         } //end looping through the blocks of I
      } //end for this action
   } //end for B simple
   else
   {
      //so B must be a COMPLEX NODE
      //Create B'
      B->set_type(ABSENT);
      B_split=new block(B->give_X1()->give_all_states(),
         B->give_X2()->give_all_states());

      if (B->give_X1()->give_type()==COMPLEX_NODE)
      {
         B->give_X1()->set_parent_block(NULL);
         W->give_ll_block()->add(B->give_X1());
      }
      else B->give_X1()->set_type(ABSENT);

      if (B->give_X2()->give_type()==COMPLEX_NODE)
      {
         B->give_X1()->set_parent_block(NULL);
         W->give_ll_block()->add(B->give_X2());
      }
      else B->give_X2()->set_type(ABSENT);

      //loop through all the transitions
      for(Link<transition *> *p_l_transition=p_ll_transition->give_first();
         p_l_transition!=NULL;
         p_l_transition=p_l_transition->give_next())
      {
         instance_event *p_ie=p_l_transition->give_object()->
            give_instance_event();
         B_split->set_info(p_ie);
         B_split->give_X1()->set_info(p_ie);
```

University of Cape Town

```
//we now have the list I of sets that are subsets of T-1a[b]
Linked_List<block *> *I=B_split->give_ll_block_info();
//for each X in I, split based on 3 rules
block *X1, *X2, *X3, *X;

Link<block *> *p_l_block=I->give_first();

while(p_l_block!=NULL)
{
   X1=new block;
   X2=new block;
   X3=new block;

   X=p_l_block->give_object();
   //for each element in the block
   Link<state *> *p_l_state;
   p_l_state=p_l_block->give_object()->give_ll_state()->
      give_first();
   while(p_l_state!=NULL)
   {
      state *p_state=p_l_state->give_object();

      //use rules here
      switch (p_state->give_info())
      {
         case 1 : p_state->move_state(X1);
            break;
         case 2 : p_state->move_state(X2);
            break;
         case 3 : p_state->move_state(X3);
            break;
         default :
         {
               writeToFile("main exit 322");
               exit(1);
         }
```

University of Cape Town

```
      };
      p_l_state=p_l_block->give_object()->give_ll_state()->
         give_first();
   };

   //states have now been moved from X to X1, X2, X3
   //problem if X is in W, and all states are just moved to Xi

   X1->reset_info();
   X2->reset_info();
   X3->reset_info();

   //examine the states in each of X1, X2, X3
   int option=0;

   if (X1->give_count()) option+=1;
   if (X2->give_count()) option+=2;
   if (X3->give_count()) option+=4;

   switch(option)
   {
      case 0 :
      {
         //impossible for all sets to have no elements
         writeToFile("main exit 347");
         exit(1);
         break;
      }
      case 1 :
      {
         //all the elements went into X1
         //move list of states back to X
         X->give_ll_state()->set_first(X1->give_ll_state()->
            give_first());
         X->give_ll_state()->set_last(X1->give_ll_state()->
            give_last());
```

University of Cape Town

```
        //add X to Q
        Q->give_ll_block()->add(X);
        delete X1;
        delete X2;
        delete X3;
        break;
    }
    case 2 :
    {
        //all the elements went into X2
        //move list of states back to X
        X->give_ll_state()->set_first(X2->
            give_ll_state()->give_first());
        X->give_ll_state()->set_last(X2->
            give_ll_state()->give_last());
        //add X to Q
        Q->give_ll_block()->add(X);
        delete X1;
        delete X2;
        delete X3;
        break;
    }
    case 3 :
    {
        //elements split between X1 and X2

        construct(X, X1, X2);
        Q->give_ll_block()->add(X1);
        Q->give_ll_block()->add(X2);
        delete X3;
        break;
    }
    case 4 :
    {
        //all the elements went into X3
        //move list of states back to X
```

University of Cape Town

```
        X->give_ll_state()->set_first(X3->give_ll_state()->
            give_first());
        X->give_ll_state()->set_last(X3->give_ll_state()->
            give_last());
        //add X to Q
        Q->give_ll_block()->add(X);
        delete X1;
        delete X2;
        delete X3;
        break;
    }
    case 5 :
    {
        //elements split between X3 and X1
        construct(X, X1, X3);
        Q->give_ll_block()->add(X1);
        Q->give_ll_block()->add(X3);

        delete X2;
        break;
    }
    case 6 :
    {
        //elements split between X3 and X2
        construct(X, X2, X3);
        Q->give_ll_block()->add(X2);
        Q->give_ll_block()->add(X3);

        delete X1;
        break;
    }
    case 7 :
    {
        //elements split between X1, X2, X3
        block *new_block=new block;
        construct(X, X1, new_block);
```

University of Cape Town

```
                construct(new_block, X2, X3);
                Q->give_ll_block()->add(X1);
                Q->give_ll_block()->add(X2);
                Q->give_ll_block()->add(X3);
                break;
             }
             default:
             {
                writeToFile("main exit 425");
                exit(1);
             }
          };
          //get the next block in I
          p_l_block=p_l_block->give_next();

          //remove X from I
          I->remove(X);
       };

       Link<block *> *p_lb=Q->give_ll_block()->give_first();
       while(p_lb!=NULL)
       {
          p_lb->give_object()->consistent_block();
          p_lb=p_lb->give_next();
       };
    }; //end of this action, next please
 } //end 'else' complex refinement step
 delete B_split;
 //will not delete B, as we might just have included it into the partition
}
}
```

University of Cape Town