WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2001

Message sequence chart specifications with cross verification

Timothy Shawn Boles
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Boles, Timothy Shawn, "Message sequence chart specifications with cross verification" (2001). *Graduate Theses, Dissertations, and Problem Reports*. 1104.
https://researchrepository.wvu.edu/etd/1104

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

MESSAGE SEQUENCE CHART SPECIFICATIONS WITH CROSS VERIFICATION

by

Timothy Shawn Boles

A thesis submitted to the College of Engineering and Mineral
Resources at West Virginia University in partial fulfillment of
the requirements for the degree of

Master of Science in Computer Science

Approved by : Dr. Bojan Cukic, Ph.D.
Chairperson of Supervisory Committee

Professor Ali Mili, Ph.D.

Dr. Vittorio Cortellessa, Ph. D.

Lane Department of Computer Science & Electrical Engineering

Morgantown, West Virginia

2000

Keywords: Message Sequence Charts, Message Flow Graphs, Formal Specification, Software
Engineering, Formal Verification, Software Verification

Abstract

MESSAGE SEQUENCE CHART SPECIFICATIONS
WITH CROSS VERIFICATION

by Timothy Shawn Boles

Chairperson of the Supervisory Committee: Dr. Bojan Cukic
Lane Department of Computer Science & Electrical Engineering at West Virginia University

Current software specification verification methods are usually performed within the context of the specification method. There is little cross verification, pitting one type of specification against another, taking place. The most common techniques involve syntax checks across specifications or doing specification transformations and running verification within the new context. Since viewpoints of a system are different even within programming teams we concentrate on producing an efficient way to run cross verification on specifications, particularly specifications written with Message Sequence Charts and State Transition Diagrams.

In this work an algorithm is proposed in which all conditional MSCs are transformed into an algebraic representations, Message Flow Graphs and by stepwise refinement, a Global State Transition Graph is created. This GSTG has all the properties of a State Transition Diagram and therefore can be analyzed in conjunction with the original STD.

TABLE OF CONTENTS

1. Introduction 1
1.1 Background 1
1.2 Message Sequence Charts 4
1.3 State Transition Diagrams 5
1.4 Objectives 6
1.5 Focus 6
2. Software Engineering 7
2.1 Bridge Building vs. Software Development 7
2.2 How Far Have We Come? 7
2.3 How Much Does This Cost? 8
2.4 This is Not Bridge Building. 8
2.5 The Evolving Software System. 9
2.6 The Three Phases of Software Engineering 10
2.7 Models of Software Engineering 10
2.7.1 Classical Process Models 11
2.7.2 Evolutionary Process Models 12
2.8 Problem Analysis Techniques 14
2.8.1 Functional Oriented Techniques 15
2.8.2 Information Oriented Methods 15
2.8.3 Object Oriented Methods 16
3. Introduction 17
3.1 Background 17
3.2 The Basics of a STD 17
3.3 Beyond the Basics 18
3.3.1 State Transition Tables 18
4. Message Sequence Charts 22
4.1 Background 22
4.2 Basics of Message Sequence Charts 23
4.3 Beyond the Basics 24
4.3.1 Conditions 26
4.3.2 Standard Message Flow Diagram 26
4.3.3 MSC-Composition/MSC-Decomposition 27
4.4 Message Sequence Charts Vs. Message Flow Graphs 29
5. Message Flow Graphs 30
5.1 Description 30
5.2 Simple Message Flow Graphs 31
5.3 Message Flow Graph Definition 33
5.4 The Translation of Message Sequence Chart to Message Flow Graph33
5.5 MFG to a pbMFG 36
5.6 MFGs to Global State Transition Diagrams 36
6. Specifying Evolving Systems 40

6.1 Consistency in Software Specifications ... 41
6.2 The Methodology ... 44
6.3 The Example ... 45
6.4 The Corresponding Functional MSC ... 46
6.5 pbMFG to Global State Transition Graph ... 47
6.6 Abstraction of the GSTG ... 47
7. Concluding Remarks ... 49
Appendix A ... 52
Appendix B ... 54
Derived MSCs ... 54
Derived cMFGs ... 55
pbMFG ... 57
Global System States ... 57
State Transitions ... 58
Global State Transition Graph ... 58

LIST OF FIGURES

Number *Page*
Figure 1 : MSC of a withdraw from an Automatic Teller Machine....5
Figure 2 : State Transition Diagram for a Pop Machine....6
Figure 3: Prototype Life Cycle....12
Figure 4: Spiral Model....14
Figure 5: STD of a CD player....18
Figure 6 : STD of CD player with attached transitions....20
Figure 7 : STD of a CD player with rule expressions....21
Figure 8 : Basic MSC....24
Figure 9 : Messaging....25
Figure 10 : Switching system from [ITU-T Z.120 pg 43-44]....27
Figure 11 : MSC for Play CD....28
Figure 12 : MSC for Pause CD....28
Figure 13 : Combined MSC for Play CD and Pause CD....28
Figure 14: Message Flow Graph....32
Figure 15 : Looping MSC....35
Figure 16 : cMFG....36
Figure 17 : pbMFG....36
Figure 18 : MFG with node labels and messages....37
Figure 19 : Part of an MFG with Branching....38
Figure 20 : Set of Message Flow Graphs....43
Figure 21 : Global State Transition Graph....43
Figure 22 : Abstraction of Global State Transition Graph Figure 21....44
Figure 23 : Algorithm Pictured....45
Figure 24 : CD State Transition Diagram with Messages....46
Figure 25 : Abstraction Z' of CD player GSTG....48
Figure 26 : Singlar path collapsed....50
Figure 27 : Cycle collapsed....50
Figure 28 : GSTG with Condition notation....51
Figure 29 : Global State Graph....53
Figure 30 : MSC Initialize....54
Figure 31 : MSC for CD Found....54
Figure 32 : MSC for No CD found....55
Figure 33 : MFG derived from Figure 30 : MSC Initialize....55
Figure 34 : MFG derived from Figure 31 : MSC for CD Found....56
Figure 35 : MFG derived from Figure 32 : MSC for No CD found....56
Figure 36 : pbMFG....57
Figure 37 : CD player GSTG....58

ACKNOWLEDGMENTS

I wish to thank Dr. Steve Easterbrook for his input and encouragement through the original drafts and research into cross verification of message sequence charts. I would also like to think Dr. Bojan Cukic for taking up the mantel of guidance and chairperson of the supervisor committee after Dr. Easterbrook moved on in his career.

Most of all I need to think my wife, Ginger Boles. Without her encouragement and understanding this thesis might have never been completed. It is easy in life to get sidetracked and off course when working on an extended project such as thesis research tends to be. Ginger has been the ruder that has consistently guided me back to the course of research and completing this work. Thanks for everything you do.

Chapter 1

INTRODUCTION

To be able to understand the need for cross verification methods, some background information is needed. This chapter briefly describes the background, outlines the thesis objectives and gives a brief description of message sequence charts and state transition diagrams .

1. Introduction

1.1 Background

The life cycle of software is a complex issue. The Year Two-Thousand (Y2K) problem emphasized the attitudes and prejudices of previous as well as current programmers and system development teams. One of the reasons that Y2K arose is the fact that the system programmers/designer did not conceive that their application would live into the twenty first century. Their view of the life-cycle of the system was one that lived and died within a set time frame. We now know that systems continue to live and grow in a prolonged if not infinite life. Systems are not stagnant. They grow with time. With growth, a system will be updated to incorporate new functions and some older ones to be deleted or at least modified. In other words the requirements of a system change. One of the goals of this thesis is to provide a method for comparing/verifying the specifications of the new functions against the original specification.

System development goes through many phases. One possibility for the sequence of phases follows: The first phase is a high-level analysis and requirement specification. This high-level specification describes the required functionality of the system. The next phase is the design phase, which takes an abstract specification of the design and refines it towards the implementation. Once the implementation is complete the final phase before deployment is testing. In the early years of software development deployment was thought of as the final phase of software development. Now the systems live and grow beyond the initial deployment with patches, updates and new versions

continuously being released. These large continuously evolving software projects often have a myriad of people developing specifications for the project. These specifications are often written in different forms. The methodology within this thesis will provide a means for verifying two of these different forms of specifications against each other.

The requirements of a system often start as natural language description of a problem to be solved. Writing these specifications in natural language leaves the system open to interpretation. Natural language is ambiguous and has a tendency to give incomplete information. One way to better understand and view a problem is by writing the specification in formal languages and/or create non-ambiguous graphical notation of the requirements. There are many useful techniques for describing a system and defining requirements. Formal description techniques (FDT), like State Transition Diagrams (STD), Specification and Description Language (SDL), Structured Analysis and Design Technique (SADT), Object-Oriented Analysis and Design (OOAD) and Message Sequence Charts (MSC), have been developed to ensure unambiguous, concise, complete and consistent specifications. These formal description techniques (FDTs) often allow for some parts of analysis and synthesis activities in the development life cycle to be automated. The automated processes can be anywhere from the formal specification of the requirements to the implementation of the system. The validation of the design specification against formal specification of requirements, the verification of the design specification, stepwise refinement of formal specification towards implementation, and test case generation from the formal specification are just a few activities that can be at least partially automated. [ROBERT]

Structured analysis and design technique (SADT)[1]™ developed by Doug Ross at SofTech, Inc. [DAVIS] uses a model to visualize the problem. The model is composed of hierarchy of diagrams. Each sub-diagram represents some part of the problem space its parent represented. Each diagram is composed of boxes, arrows and text. Each arrow has a specific role (input, control, output or mechanism) that is defined by either a letter (I,C,O,M respectively) or its position on the box (left, top, right, bottom respectively). The arrows represent data control flow between parts of a system.

---

[1] SADT is a registered trademark of SofTech, Inc.

Structured analysis and system specification (SASS) developed by Tom DeMarco [DAVIS], is a pure top-down technique like SADT. The analyst starts by representing the system in a context diagram showing all system inputs and outputs and repeatedly refining the system with more and more detailed data flow diagrams (DFD). The notation finally decomposes into the physical and logical DFD.

Object-oriented problem analysis has its roots in the language Smalltalk. Object-oriented approaches stress the definition and refinement of objects in the real world and classes of objects in the real world. Objects are "an encapsulation of attributes and exclusive services, an abstraction of the something in the problem space, with some number of occurrences in the problem space" [COA89a]. A class of objects can be thought of as an abstraction that represents one or more objects or other classes of objects. Each object possesses attributes and inherits the attributes of classes of which they are members.

Message Sequence Charts are widely used in the telecommunications industry to capture requirements. In comparison to the SASS and SADT, MSC can be comprised of a hierarchy of diagrams. As with Object-Oriented analysis, the concentration is on individual entities. In contrast to SASS and SADT, which are interested in general data flow, MSCs are used to capture explicitly the interactions and the message exchange between processes. MSCs are being used in Object-Oriented analysis to describe the communication between objects/processes.

Often many different formal description techniques are used to develop the specifications of a system. Consistency among these specifications must be verified. One method to ensure the consistency of FDTs is to develop one specification from another. Robert et. al. take this approach in [ROBERT] where a basic message sequence chart (bMSC) is translated into an SDL specification. This translation can be done by translating the MSCs into an intermediate representation as a state transition diagram (a finite state machine) and applying a component based synthesis algorithm to complete the missing transitions to the SDL [ITO]. The purpose is to ensure consistency between the requirement state, represented by the MSC, and the design stage, represented by the SDL. This synthesis approach ensures, by construction, consistency between the SDL specification and the MSC specification; and no further validation is required [ROBERT].

There are tools available that support development and formal verification of many formal description techniques (FDTs) . Increasing the power of a verification system can be done by taking one verified view (i.e. MSC), mapping it to the other view (i.e. SDL), and running verification on it. An example of this can be found in the telecommunications industry, where tools have been developed that take MSC and convert it to SDL and vice-versa.

This methodology is most applicable to new systems and produces a new view of the system to be used in the development process. It tends to lend itself to the view that systems are stagnant and do not evolve. In this thesis we are concerned with evolving systems that are either under development or being re-engineered with new functions being added to the system. A development process includes a complete system with a verified specification, like an STD, and a function to be added to the system is written in another FDT, like a MSC. In this type of development situation, consistency between the formal description techniques must be maintained by validating the system specifications against each other.

The two Formal Description Techniques that we concentrate on in this thesis are Message Sequence Charts and State Transition Diagrams. Much of the work in the area of comparing MSCs to other specification languages arises from the telecommunication industry and concentrates on their relation to the Specification and Description Language. The SDL covers the stimulus behavior/response of state machines [LeBlanc]. We choose to use state transition diagrams for with the right transformation algorithm they can be mapped to many other types of specifications including SDLs. This section briefly explains Message Sequence Charts and State Transition Diagrams and gives some examples of their use. A more through description is included in later chapters.

1.2 Message Sequence Charts

Message Sequence Charts are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment [ANDERSSON]. A MSC essentially consists of set of instance (i.e. Processes) that run in parallel and exchange messages in a one-to-one, asynchronous fashion (see Figure 1). Instances can individually execute internal actions, use timers to enforce timing constraints, create and terminate instances of processes. The messages are presented in an easy way using message sequence charts (see Figure 1). In these charts, the arrows represent

messages, the vertical lines are time axes and the boxes on top represents the instances (Processes) of the system or the environment.

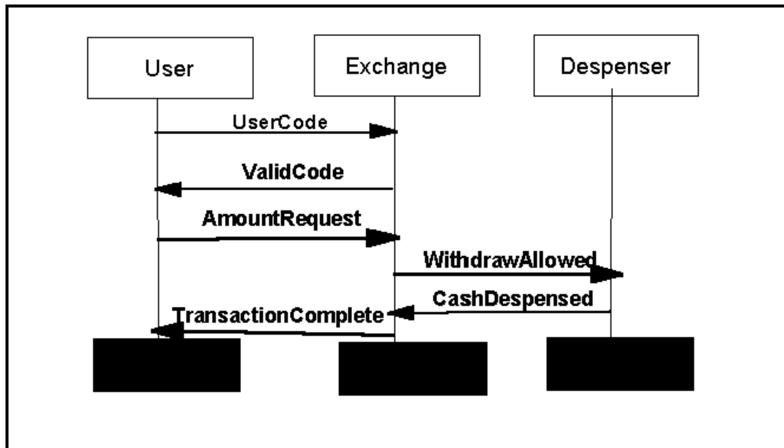


Figure 1 : MSC of a withdraw from an Automatic Teller Machine

There are three major ways in which MSCs are used [LEUE 1]:

1. To visualize actual system execution, during debugging and program understanding
2. As a Language to document early design decisions
3. Document test cases or functional-validation criteria that an implementation must satisfy

1.3 State Transition Diagrams

Many of the formal description techniques (FDTs) are really just extended finite state machines. Many FDTs like the specification language SDL, already have algorithms developed to transform them into finite-state machines which are in essence State Transition Diagrams (STDs). A STD consists of states of a system and the events in the systems that cause transitions between those states. A simple STD for a Pop Dispensing machine can be seen in the Figure 2. The states are represented by circles and the labeled arcs between states represent potential transitions between states they span.

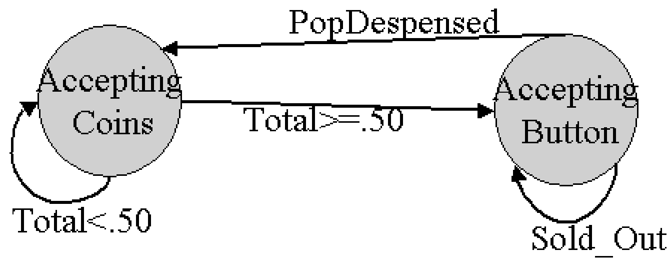


Figure 2 : State Transition Diagram for a Pop Machine

1.4 Objectives

System development methodologies for incremental specifications, often involve trying to integrate the specification for a new section with the previous specification. In order to assure the correctness of a system there has to be a way of checking consistency between the new specification and the previous ones. In addition often during system development more than one type of specification is used to design the system. In the telecommunication industry many development processes involve using MSCs and some other FDT (like SDLs), both which can be broken down into a finite-state machine.

The objectives of this paper are to describe situations where current techniques are inadequate. Describe an algorithm that allows incremental specification of a system by ensuring the consistency between message sequence charts and state transition diagrams.

1.5 Focus

This thesis does not attempt to propose a complete method for system specification. The thesis only outlines a method to incremental specification of a system using Message Sequence Charts and State Transition Diagrams.

Chapter 2

SOFTWARE ENGINEERING

In order to understand the need for cross verification of specifications some background information on past and current engineering methods need to be presented. This chapter draws parallels and contrasts between software engineering techniques and other engineering disciplines.

2. Software Engineering

2.1 Bridge Building vs. Software Development

Comparing bridge building and software development is the premise of a book written by Alfred Spector in 1986 [STANDISH]. Normally bridges are built on-time, on-budget and do not fall down. Software development usually is not on-time or on-budget and has a tendency to fail. Naively, one could spot the nearly 3,000 years of experience in bridge design and only approximately 50 in software design as the major difference. However there are many other reasons why bridge construction is able to do so well and software development does not. The design for building a bridge is extremely detailed and frozen, the contractor has little flexibility in changing the specifications. In software development, design specifications have a tendency to change with the contractor adding new functions or deciding to take advantage of new technology. Some bridges do fail and when this happens, it is investigated and a report is written on the reasons for the collapse. People study this report and try not to make the same mistakes. In software development the same mistakes are made over and over again, because when software fails it is covered up, ignored, and/or rationalized.

2.2 How Far Have We Come?

How far have we come in about a half a decade of building up software engineering practices? Clemetine was a lunar satellite set into orbit by NASA and DoD. In 1994, the United States of America's Department of Defense (DoD) used rigorous and state-of-the-art software engineering

techniques to ensure the correctness of the Clementine project. In support of the space-based missile defense system, a major part of the Clementine mission was to validate and test the systems target software. Clementine was launched successfully and running in orbit, an instruction to fix the moon focus in sight was executed. This command revealed a bug in the real-time software and the program caused the spacecraft to fire off its maneuvering thrusters continuously for 11 minutes. The satellite was then out of fuel and spun widely off into space [ANDERSSON].

It seems that even after about a half a decade we still need to improve our "state-of-the-art" software engineering practices.

2.3 How Much Does This Cost?

Research by Standish Group in 1995 shows that 31.1% of projects started are canceled before they are completed. The same study showed that 53% of projects will cost 189% of their original estimates. This does not include loss of revenue or equipment from software failure. [STANDISH] One example of the loss of revenue due to software failure is baggage handler at the new Denver airport. Though there is a combination of many reasons for the delayed opening of the Denver Airport, the baggage handler was seen to the public as the main cause of the delay and the $360 million dollars in estimated lose of revenue. In addition to lose of revenue the problems with the baggage handler system prompted the installation of an alternative baggage handler. Again it is impossible to state what percentage of the baggage handler problem was pure mechanical vs. pure software but the combination cost the company $89 million for the alternative baggage handling system. [GAO2]

The Standish Group estimated that in 1995 American companies and government agencies spent $81 billion for canceled software projects and an addition $59 billion for software projects that were late in completion. Software projects that were completed on-time and on-budget made up only 16.2% of the sample.

2.4 This is Not Bridge Building.

When building a bridge the design is frozen, the contractor has little flexibility in changing the specifications. Software systems are complex, and complex systems are always changing. Dramatic

decreases in cost and the increasingly powerful, smaller-size and more reliable computers have led to larger and more complex software that only a decade ago would not have been possible. Early developers of software systems saw their product eventually being debunked by new "better:" software. But today there is growing recognition that software, like all complex systems, evolves over a period of time. [PRESSMAN] The development of large and complex (evolving) systems requires constant changes in the specification. The development of systems takes several years and as a result some of the early requirements become obsolete, some change and many new requirements must be added. [VAQUEZ]

In order to do better than developers of the past we must improve and automate the software development process, while much has been done, analysis and specification of requirements (which is the only "true" measurement of software success) remain a relatively untouched area. [DAVIS] No matter what else a system does if it does not meet the customers requirements and expectations then it is a failure.

Flexibility in the requirements analysis and design phases of the life cycle is a must. The user has to be able to change requirements and this change is reflected in the analysis, design and coding. There are several methods of requirements modeling, the choice of methods depends on the system to be designed, the tools to be used, the system architecture and development method to be used. Each method has its strengths and weaknesses.

2.5 The Evolving Software System.

The market place continually expands, making a need to constantly adapt existing systems. The adaptation of current software to meet new needs is called re-engineering. Software systems normally are re-engineered to incorporate performance improvements or to meet the changing needs of an organization. A study done by G2 Research Incorporated projected that the computer system re-engineering market was expected to double between 1995 to 1997. [MILLER] A larger and larger percentage of information technology resources is being expended on maintaining software systems, (40 to 80 percent) and less and less developing new systems. [MILLER] The maintenance of software falls into the third phase of software engineering (see section 2.5) but also encompasses the first two phases.

2.6 The Three Phases of Software Engineering

It does not matter what you want to build. The project size, application area, or complexity do not matter either. All work that is associated with software engineering can be categorized into three generic phases. [PRESSMAN]

- Definition Phase- The key requirements of the system and software are identified. Three major tasks will occur:
  - Software Project Planning
  - System or Information Engineering
  - Requirements Analysis
- Development Phase – The key methods of building the software are defined. Three specific technical tasks should occur:
  - Software Design
  - Code Generation
  - Software Testing
- Maintenance Phase – Reapplies the steps of the definition and development phases but does so in the context of existing software. Four types of change are encountered:
  - Correction- Changes the software to correct defects
  - Adaptation- Modifications to meet changes in the external environment
  - Enhancement- Extending the software beyond its original functional requirements
  - Prevention- Changing the computer program so that they can be more easily corrected, adapted and enhanced.

2.7 Models of Software Engineering

The work can take on many different forms and is referred to as software engineering paradigms. The choice of the paradigm depends on the nature of the project, the application, the methods and tools to be used. It does not matter which of the paradigms you choose they all will encompass the three above mention phases. Of these phases the definition is the most important. It does not matter how well the code is written, or how much reuse you can get out of the components, or how wonderful the

user interface is, none of this matters unless the software meets the customer expectation and needs (the requirements).

2.7.1 Classical Process Models

Software engineering is relatively a young discipline at 50 years, if you take into account that things like bridge engineering has been around for close to 3,000 years. Much of the early models have their root in other engineering disciplines. These models worked in the previous generation of software development for a number of reasons. Because of the limit in processor speed, memory available and cost of the machines, software had a tendency to be small enough and its functionality limited enough to be written quickly with out much change to the requirements.

The Linear Sequential Model – "waterfall model"

The linear sequential model for software engineering is the oldest and most widely used paradigm for software engineering. [PRESSMAN] It is modeled after the conventional engineering cycle with six activities.

1) System information engineering and modeling
2) Software requirements analysis
3) Design
4) Code Generation
5) Testing
6) Maintenance

Prototyping

As cited above, unless you meet the customers needs and expectations nothing else about the software really matters. What if you have a customer that only has a general idea for the objectives of the software? Then the prototyping paradigm may be the answer. With prototyping there are four steps, with step three being cyclic (see Figure 3):

1. Developer and customer meet and define the overall objectives and define what requirements are known.
2. A quick design of the input approaches and output formats that the customer sees is put together.

3. The Cycle
   a. The prototype of the software is quickly developed
   b. The customer "drives" the prototype to refine the requirements for the software to be developed.
   c. Iteration occurs as the prototype is tuned to the needs of the customer
4. The final product is built.

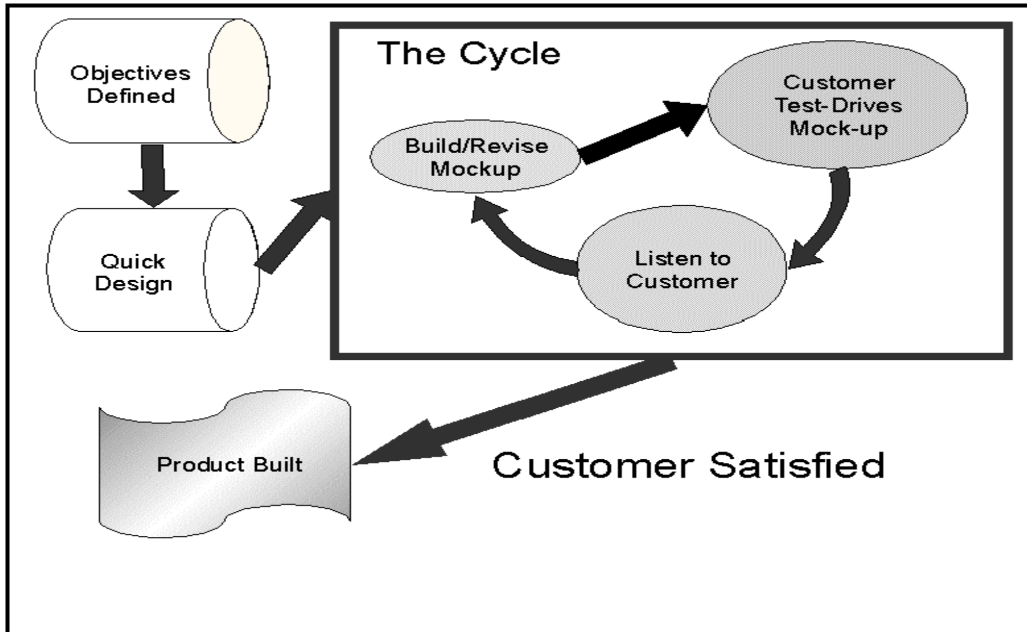


Figure 3: Prototype Life Cycle

As with any approach this has its problems. The major one with prototyping is that the customer sees the design and thinks that it is the finished product, unknowing that the guts of the program are not in place and how much time that it takes to finish. The customer needs to understand from the forthright that this method is only to gather the requirements and nothing else.

The two classical software engineering models above (Prototype and Waterfall), like bridge engineering do not take into account the evolutionary nature of complex systems.

2.7.2 Evolutionary Process Models

It is quite noticeable when you go shopping for a new computer to replace your "out-of-date" computer that the amount of money you spent on the last system will buy you an amazingly more

powerful machine. Dramatic decreases in cost of different computer parts such as memory and processors, combine with increasingly smaller-size, more powerful and more reliable computers gives rise to software which only a decade ago would not have been possible. The market place continually expands, making a need to constantly adapt existing systems. The development of large and complex (evolving) systems requires constant changes in the specification. The next two models take into account the evolutionary nature of software. They are iterative and enable software engineers to develop increasingly more complete versions of the software.

The Incremental Process Model

Rather than producing a product, the prototype model is a requirement elicitation tool. If you combined the prototype model and the linear sequential model you get an approach commonly called the incremental model. Every increment gives a deliverable piece of software to the customer. Generally the increments are more complex and sophisticated as time progresses. Early increments are the core product that provide capability that serves the user but also provide a platform for evaluation by the user. The features for later increments may have already be known or developed from the evaluation of previous increments. This model is particularly useful when staffing is unavailable for a complete implementation. A smaller staff can produce the first few increments and if they are well received then additional staff can be added to implement the later stages.

The incremental process model as well as the classical process models generally end with the completion of the project. Similar to the incremental model is the spiral process model, which can be used during the whole life of a piece of software.

The Spiral Process Model

The spiral model is divided into a number of section activities. The complexity and details of these sections depend on the size of the project and those who have set it up. Typically they contain some if not all the following sections (as can be seen in Figure 4: Spiral Model).

Requirements Development
Planning
Risk Analysis
Engineering

Construction & Release

Customer Evaluation

As the evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the core. As the project progresses around and out the spiral, the results are more and more complex. The first circuit may produce the initial specifications, the second a prototype, each additional circuit may produce more sophisticated versions of the software. Each pass through the planning sections results in adjustments to the project plan. Cost and schedule are adjusted based on the feedback derived from the customer evaluation. [PRESSMAN]

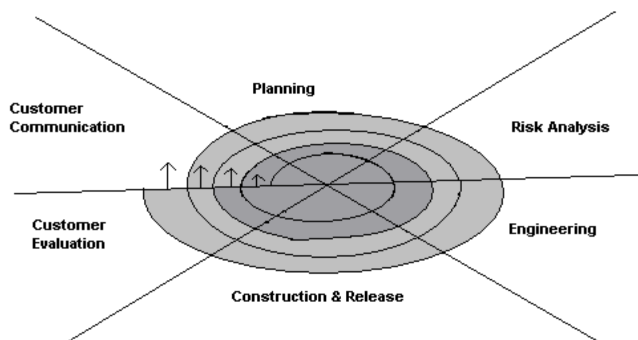


Figure 4: Spiral Model

Unlike the classical process models that end when the software is delivered, the spiral model can be adapted to the whole life of the software. The starting point of a project within the model depends on the type of project. Where a concept of a project starts at the core and continues through multiple iterations along the spiral pathway a piece of software that only needs enhancements will start further out along the axis (see Figure 4: Spiral Model)

2.8 Problem Analysis Techniques

The first step in any of the above mentioned software engineering methods, is to understand the "problem" to be solved.

Problem analysis involves learning about the problem to be solved, understanding the needs of potential users, and knowing any constraints (hardware or otherwise) on the solution. There are many

techniques for surveying a problem and gathering the system requirements, the use of which will be based on the engineers, problem size and type of problem.

2.8.1 Functional Oriented Techniques

A functional model shows what the system is to do, without showing how or when it is done. A technique that is often used is called Structured Analysis/Structured Design (SASD). Basically SASD breaks down a system into a network of processes or functions. The processes are connected by data-flow messages. Each process transforms its input data into output data. The output of one process can be the input to another. Data flow diagrams (DFDs) are used as graphical representations of the functional model. The structure of a DFD is a directed graph with three types of nodes (processes, agents and datastores) and oriented arcs connecting pairs of nodes. As with all notations and techniques, they are often changed to meet the projects needs giving more details where needed. More recent versions have added state transitions diagrams (often referred to as control flow diagrams) and bottom-up analysis driven by event identification.

2.8.2 Information Oriented Methods

Information Engineering (IE) encourages object modeling of data components of a system. A common definition is "…an interlocking set of formal techniques in which enterprise models, data models and process models are built up in a comprehensive knowledge base and are used to create and maintain data-processing systems" [DAAE]. Often many methods are used collectively to develop a system. Information Engineering can be used for the entire information systems development process. Between stating the business goals and using the production system are steps that focus on business processes, (technical) design, construction and transition to the new system. Many parts of the design and construction stages can be automated by the use of CASE tools. Information Engineering (IE) is an integrated method, with all aspects of planning, analysis, design and construction of information systems interrelated. Once the business goals and processes are defined in natural language, specifications are written in FDTs to be a precise, cohesive and complete. The function specification uses process dependency and action diagrams. Cross-referencing of functions to entities is provided for and state-transition diagrams explicitly associate event-creating operations with entities, producing a data oriented specification.

2.8.3 Object Oriented Methods

One description of Object-oriented systems analysis (OOSA) models is as an object relationship network with subclasses. State-transition specifications are constructed for each object and functions are modeled with data-flow diagrams. The method produces a composite activity-data model, but is achieved by attachment of activity to the data model, essentially merging data-flow diagrams with state-transition models with entities.

Chapter 3

STATE TRANISITION DIAGRAMS

Cross verification of specifications requires that there be at least two different types of specifications available for reference. Many of the current formal description language techniques have a finite state machine as their underlying architecture and therefore a state transition diagram is a natural choice as one of the types of specifications to use. In this chapter we look at the history of state transition diagrams as well give a detail description.

3. Introduction

3.1 Background

State Transition Diagrams (STD) have been used in the computer industry for many years. They have gone under various names and are applicable to many tasks. State Transition Diagrams, Harel Diagrams, State Charts, Transition Networks, State Diagrams, Finite State Machines, and Transition Diagrams are variants with slightly different semantics. They are used in fields such as natural language processing, finite automata, compiler construction, lexical analyzer construction and requirements engineering. The state-transition diagram (STD) indicates how the system behaves in response to events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. [PRESSMAN 97]

3.2 The Basics of a STD

A basic STD consists of a finite set of circles connected by arrows (arcs) (see Figure 5). The circles represent possible states and the arrows the events that cause transitions. To understand what a state-transition diagram represents we need to define some basic terms and underlining principles. A state is an observable mode of behavior. A STD indicates how the system moves from state to state. The movement from state to state is based on events. An event may or may not change the state of the

system. On this CD player (Figure 5) when the door is open, the event of pressing the "Play Button" has no effect on the system. Therefore it is not shown in Figure 5.

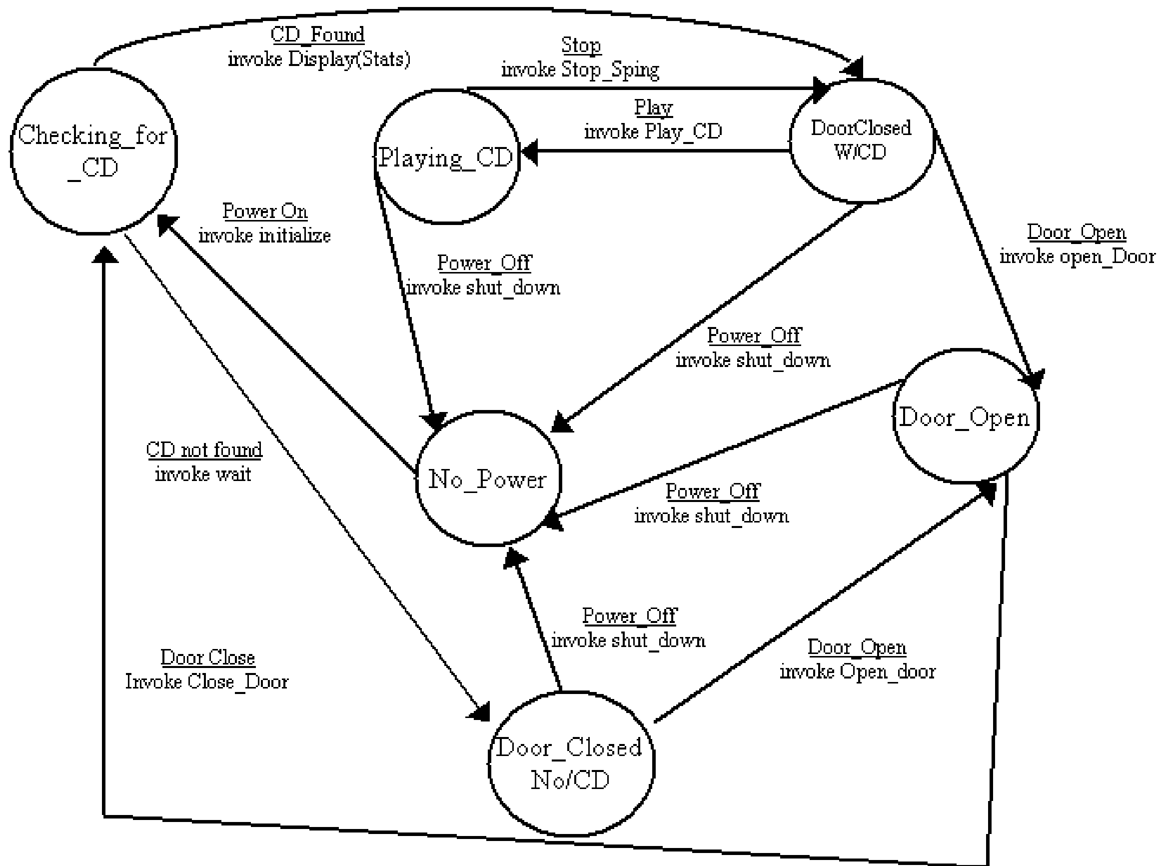


Figure 5: STD of a CD player

3.3 Beyond the Basics

State transition diagrams are used in many object oriented analysis, design and coding methods. Transition tables are another way to view the same information that is contained within a state transition diagram and is often used to generate code.

3.3.1 State Transition Tables

A transition table is a two-dimensional array with a row for each state in the transition diagram and a column for each message (event). The entry found in the mth row & nth column of the table is the

state that would be reach in the transition diagram by leaving state m via arc n. The transition table for Figure 5 is given in Table 1.

	MESSAGE/EVENT	Door Button	Power Button	Play Button	Stop Button
STATE	No Power	No Power	Door Open	No Power	No Power
	Door Closed no CD	Door Open	No Power	No CD	No CD
	Door Closed with CD	Door Open	No Power	Playing	With CD
	Door Open	No CD : With CD	No Power	Door Open	Door Open
	Playing	With CD	No Power	Playing	With CD

Table 1 : Transition Table

This example of a CD Player is non-deterministic, e.g. if in the 'No Power' state, the event 'power button' is received, the diagram offers 2 choices (No CD; With CD) for the next state. This non-determinacy indicates a flaw in the model. The system will need a way of detecting whether a CD is present. The notation doesn't help with this issue. This type of problem often occurs when using any diagramming technique. Situations arise that the technique was not developed to handle. One possible solution is to expand the diagramming technique to include syntax to overcome the non-determinacy. One variant of state transition diagrams has conditions attached to the transitions. The STD of a CD player, Figure 5 is re-diagrammed with this type of notation in Figure 6.

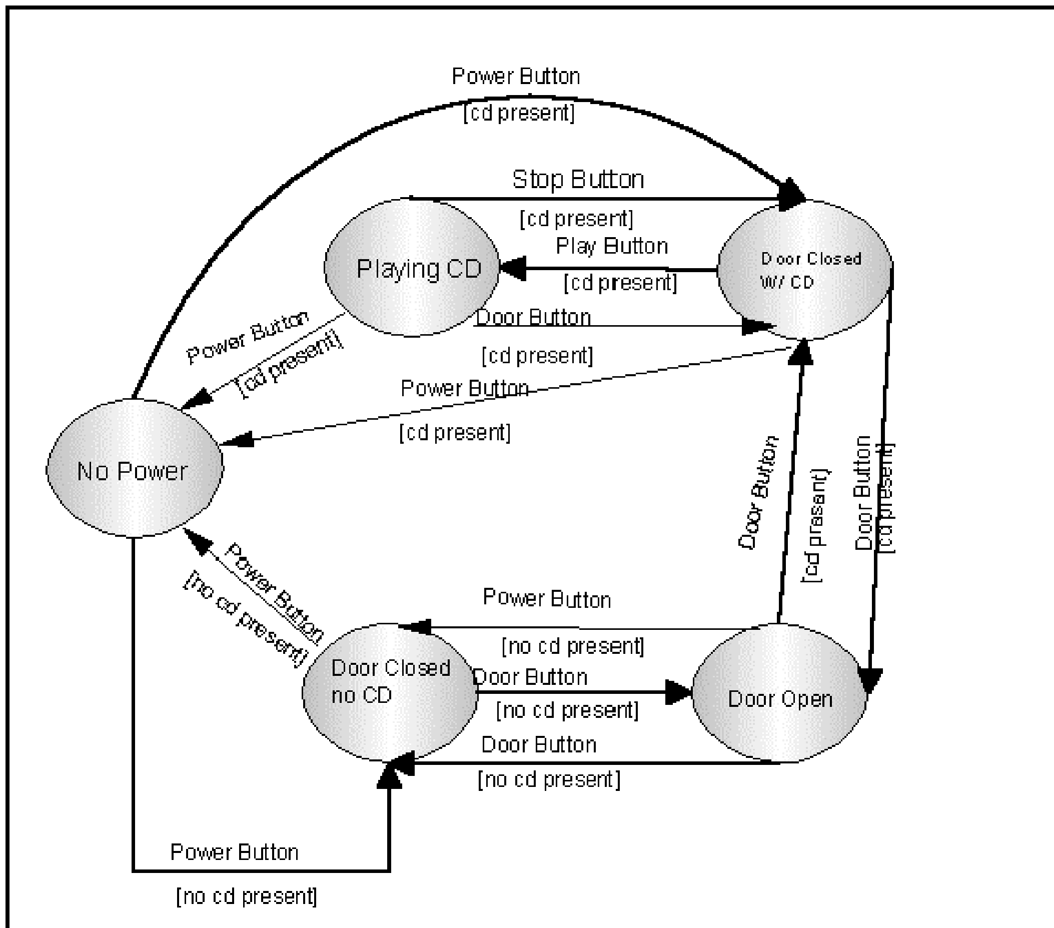


Figure 6 : STD of CD player with attached transitions

As with any modeling technique there are variations, many of which add more information through symbols or syntax. Some variations display every possible event even if it just returns the system back to the same state. Although that would explicitly show all possible scenarios it would add visual clutter and make it harder to see the "important" events. The State Transition Diagramming style recommended by Roger S. Pressman [PRESSMAN 97], involves having each arrow labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. The non-determinacy found in the CD player in Figure 5 is over come by adding a new state "Checking for CD" as can be seen in Figure 7.

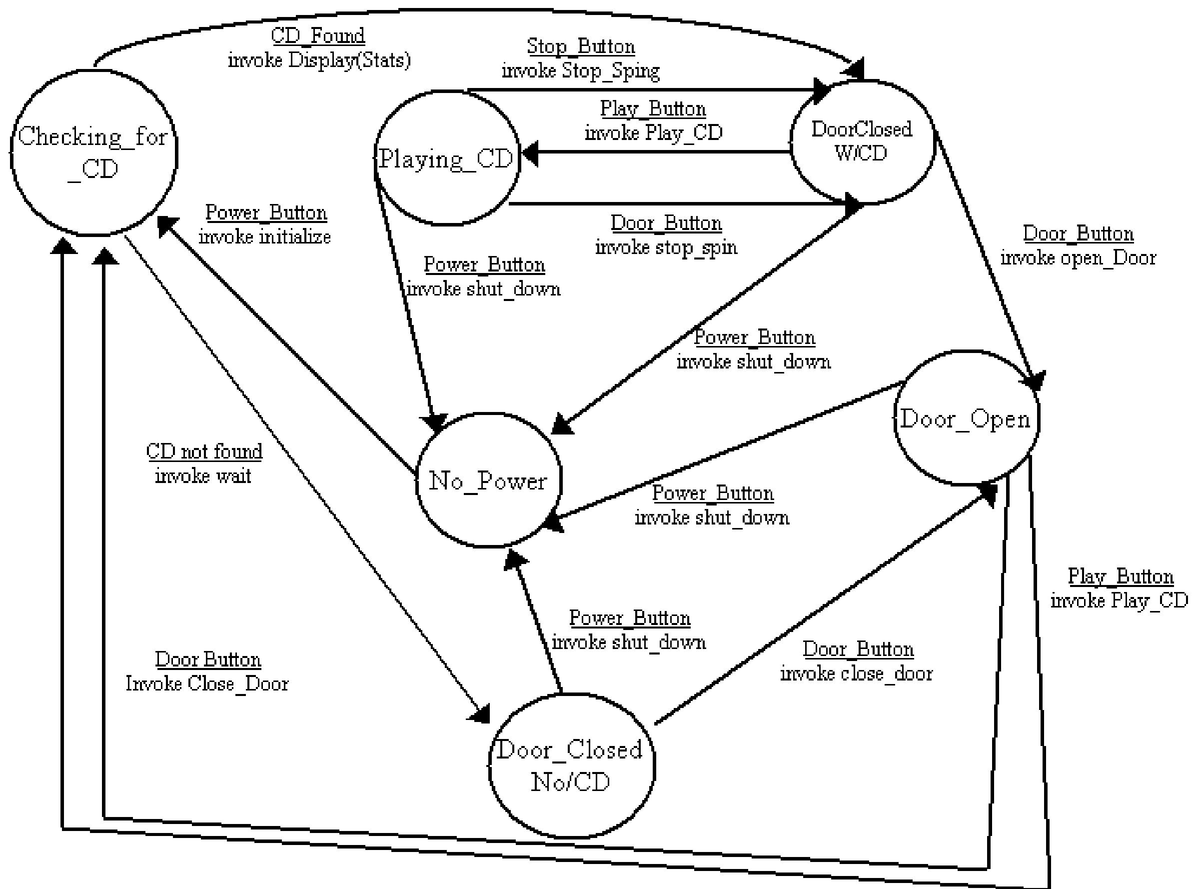


Figure 7 : STD of a CD player with rule expressions

There are many diagramming techniques for requirements modeling. Unlike Message Sequence Charts, there has been no standardization of State Transition Diagrams. It is a necessary part of requirements modeling to decide what information to display and be consistent through out the project.

Chapter 4

MESSAGE SEQUENCE CHARTS

The prevalence of the World Wide Web in our culture shows the importance of the telecommunications industry and its software. One of the most widely used types of specification within the telecommunications industry is a message sequence chart. This lends it to naturally be one of the specifications we use in our cross verification. This chapter focuses on the history, description and a summary of the standardization of Message Sequence Charts as found in ITU-T Recommendation Z.120 [ITU-T Z.120].

4. Message Sequence Charts

4.1 Background

Specification languages vary in what they describe about a system. Some FDTs describe the process behavior explicitly, leaving message flows to be inferred. On the other hand MSCs specify explicit message flow while other process behaviors must be inferred from the specification. MSCs are being used in many software engineering methodologies and tool-sets, object-oriented methodologies, design pattern methods, and early life-cycle design analysis of message exchanges. Some tool-sets that use MSCs are ObjecTime, Cinderallla, ObjectGeode, Ubet and Rhapsody.

Message Sequence Charts have been used for a long time by industry and academia under various names such as Signal Sequence Chart, Information Flow Diagram, Message Flow and Arrow Diagram. Message Sequence Charts (MSC) have been standardized by the ITU-T (International Telecommunication Union Telecommunication Standardization Sector) in recommendation Z.120 [ITU-T Z.120]. This standardization of MSCs makes recommendation Z.120 the authority on the Message Sequence Charts and their representation. An MSC essentially consists of a set of instance (i.e. Processes) that run in parallel and exchange messages in a one-to-one, asynchronous fashion. Instances can individually execute internal actions, use timers to enforce timing constraints, create and

terminate instances of processes. MSCs are used to document system requirements that guide the system design, describe test cases and scenarios, to express system properties that are verified against other FDT specifications, visualize sample behavior of simulated system specification and to express legacy specifications in an intermediate representation that helps in software maintenance and re-engineering. [ABDALLA 96]

4.2 Basics of Message Sequence Charts

This section summarizes the basics of Message Sequence Charts (MSCs) as found in [ITU-T 120]. Message sequence charts are used for the visualization of the communications between system components. They are mainly used in specification, simulation and validation of real-time systems, in particular telecommunication systems. MSCs can be used in connection with other specification languages, in particular Specification and Description Language (SDL). Often the purpose of using MSC is to provide a trace language for the specification and description of the communication behavior of system components and their environment by means of message interchange. Since in the message sequence chart the communication behavior is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC-language is easy to learn, use and interpret. In connection with other languages, it can be used to support methodologies for system specification, design, simulation, testing and documentation.

Due to standardization MSCs may serve as:

1. An overview of a service as offered by several entities;
2. A statement for requirements specification;
3. A basis for elaboration of SDL specifications;
4. A basis for system simulation and validation;
5. A basis for selection and specification of test cases;
6. A specification of communication;
7. An interface specification;
8. A formalization of use cases within object-oriented design and analysis;

A message sequence chart usually only covers a partial behavior. Further cases are generally built on them and cover exceptional behaviors.

A basic MSC diagram consists of four parts as seen in Figure 8.

1. Instance – The parts of a message sequence chart that interact are instances of entities. An object with the properties of an entity is an instance of that entity. The instance heading must contain the entity name, e.g. process name, and may specify the instance name as well. The instance body contains the ordering of events.
2. Message – The relation between an output and input is called a message. Outputs and inputs come from either the environment or an instance. A message passed between instances consists of two events:
   A. Message Input
   B. Message Output
3. Environment/Gates – Gates represent the interface between the MSC and its environment. Any message or order relation attached to the MSC frame constitutes a gate.
4. Timer – In MSC's, you can specify the setting of a timer and the subsequent time-out due to timer expiration or the subsequent timer reset.

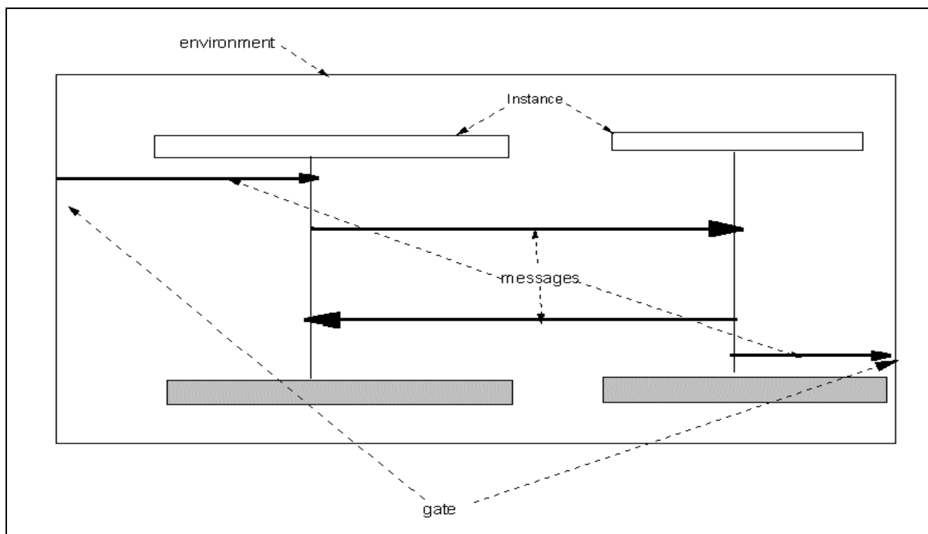


Figure 8 : Basic MSC

4.3 Beyond the Basics

Communication between system components can be described using a Message Sequence Chart. The Basic Message Sequence Chart in Figure 9 : Messaging defines communication behavior between

instances **process_a, process_b, process_c** and the **environment**. An instance has the properties of being abstract and has observable interactions with other instances or the environment. The MSC has an instance axis (vertical) for each system component covered. The time along each instance axis runs from top to bottom, but there is no time correlation between instances. With no global time axis being assumed, message events with the environment do not have an ordering.

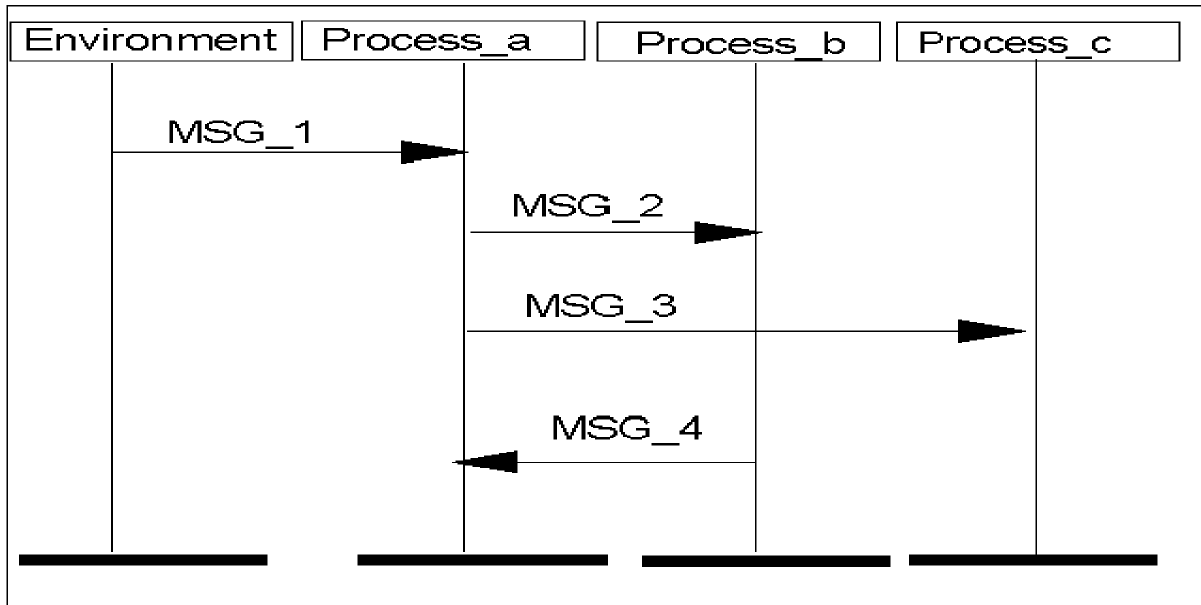


Figure 9 : Messaging

Messaging is represented by an arrow starting with the sending instance and end at the consuming instance. A very intuitive rule that a message must be sent before it is consumed leads to a generalized ordering mechanism between instances. In Figure 9 this implies for example **MSG_4** can only be received by **Process_a** after it has been sent by **Process_b** and consequently after the reception of **MSG_2** by **Process_b**. Therefore **MSG_1** and **MSG_4** are ordered in time, but for **MSG_3** and **MSG_4** no order exists. Sine we have asynchronous communication [ITU-T Z.120], it is possible for **MSG_3** to be sent, then send and consume (receive) **MSG_4**, and finally receive **MSG_3**. Even with asynchronous communication there is an imposed ordering upon the system:

Using message inputs [labeled by in(mi)] and output [labeled by out(mi)] together with the transitive closure:

out(MSG_2)<in(MSG_2)

out(MSG_4)<in(MSG_4)

in(MSG_1)<out(MSG_2)<out(MSG_3)<in(MSG_4)

in(MSG_2)<out(MSG_4)

4.3.1 Conditions

A condition describes either a global system state (global condition) referring to all instances contained in the MSC or a state referring to a subset of instances (non-global condition). The keyword ***condition*** together with the condition name has to be used for each instance to which it is attached. The keyword ***shared*** together with the instance list denotes the set of instances by which the condition is shared. A global condition is defined by means of the keyword ***shared all***.

The semantics of global conditions, representing global systems states, refer to all instances involved in the MSC. For each Message Sequence Chart: a) An initial global condition (global initial state), b) A final global condition (global final state), c) Intermediate global conditions (global intermediate states), may be specified using the keyword **shared all** in the textual representation. Graphically global conditions are global labels on processes axes as can be seen in Figure 10 as the Seizure, Idle and Talking.

4.3.2 Standard Message Flow Diagram

This example shows a simplified connection setup within a switching system. The example below shows the graphical representation (Figure 10), of the most basic MSC-constructs:

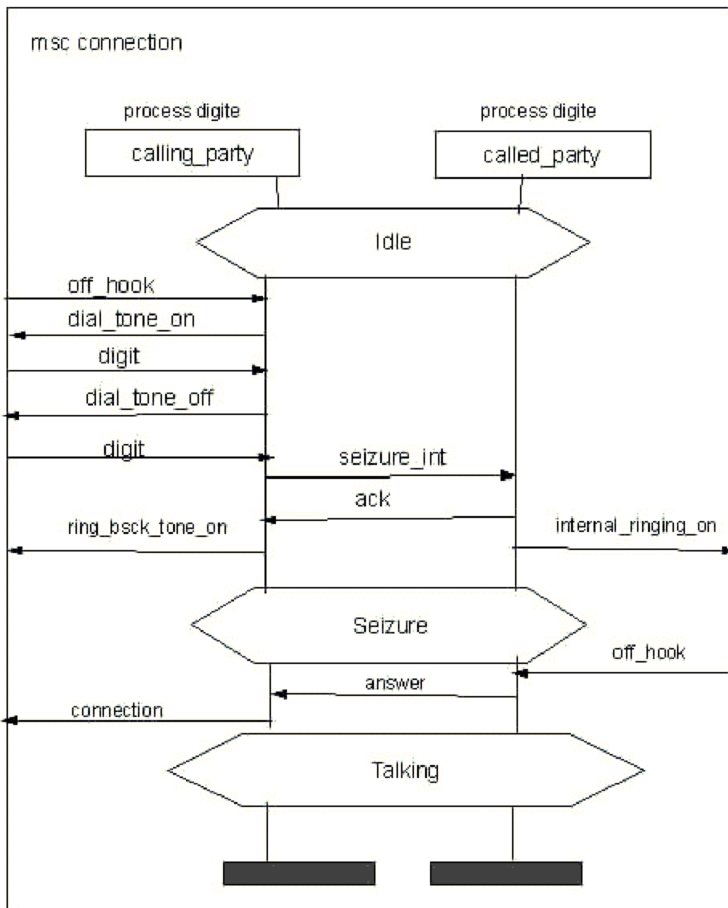


Figure 10 : Switching system from [ITU-T Z.120 pg 43-44]

4.3.3 MSC-Composition/MSC-Decomposition

Message Sequence Charts enable a system to be defined by looking different scenarios In the following example the composition of MSCs by means of global conditions is demonstrated. The final global condition 'CD_Playing' of MSC Play CD (Figure 11) is identical to the initial global condition of MSC Pause CD(Figure 12). This condition allows composition of two MSC charts into a larger more complete system description (Figure 13).

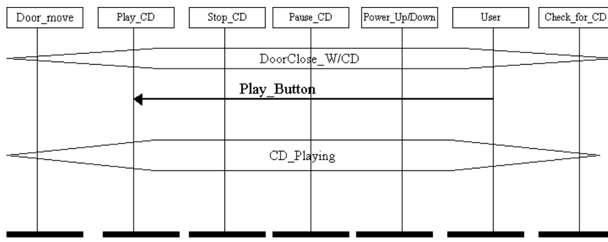


Figure 11 : MSC for Play CD

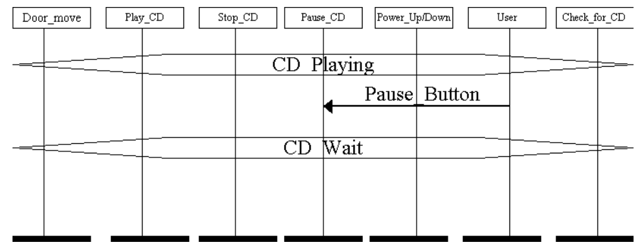


Figure 12 : MSC for Pause CD

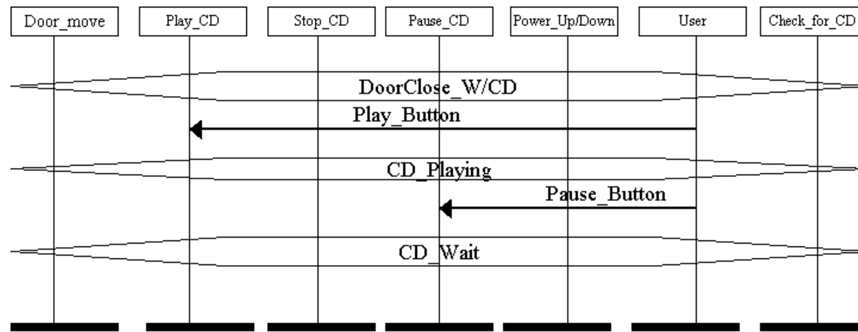


Figure 13 : Combined MSC for Play CD and Pause CD

As can be seen from the brief synopsis above MSCs are graphical devices used to represent the sending and receiving of messages between processes within a system. They may serve as:

a) An overview of a service as offered by several entities;
b) A Statement for requirements specification;
c) A basis for elaboration of SDAL specifications;
d) A basis for system simulation and validation;
e) A basis for selection and specification of test cases;
f) A specification of communication;
g) An interface specification;
h) A formalization of use cases within object-oriented design and analysis;

Even with all they have going for them they are of little use for mathematical reasoning. It is necessary to map these MSCs to an algebraic representation of process control and message flow. The algebraic representation we are going to use are Message Flow Graphs base on [LEUE 1].

4.4 Message Sequence Charts Vs. Message Flow Graphs

MSCs are graphical devices used to represent the sending and receiving of messages between processes within a system. They are used for individuals to gain a better understanding of a system's specifications, but they are of little use for mathematical reasoning. In order for us to be able to validate a message sequence chart against a state transition diagram we will have to have an mathematical expression of the MSC. We can do this by mapping the MSCs to an algebraic representation of process control and message flow called a Message Flow Graph. The Message Flow Graphs (MFGs) that we use in this thesis are from [LEUE 4] and will be covered in more detail in Chapter 5.

Chapter 5

DEVELOPING MESSAGE FLOW GRAPHS

The previous chapter went into detail about message sequence charts and their usefulness in understanding a system's specification. The very nature of a message sequence chart does not allow cross verification against another type of specification. In this chapter we will outline the development of a message flow graph that is a mathematical representations of a message sequence chart. Message flow graphs can be manipulated and transformed into other types of expressions and therefore are useful for cross verification.

5. Message Flow Graphs

In order for us to do any mathematical reasoning on a specification there needs to be an algebraic representation of the system. MSCs are graphical devices used to represent the sending and receiving of messages between processes within a system, but they are not mathematical objects. In order to meet this need of an algebraic notation a Message Flow Graph (MFG) was developed. Message Flow Graphs (Figure 14) are used as an alternate (mathematical) description of high-level message flow diagrams like MSCs. The Message Flow Graphs (MFGs) that we use in this thesis are from [LEUE 4] and are an algebraic representation of process control and message flow for communicating processes.

5.1 Description

A Message Flow Graph (MFG) is a graph representing concurrent processes exchanging messages. The graph has two types of nodes:

1. **send** nodes which represent a process sending a message
2. **receive** nodes which represent a process receiving a message

The graph has two types of edges:

1. ***next-event*** edges connect nodes to their successors within a process
2. ***signal*** edges connect nodes to nodes in other processes with which they communicate

The essential property of an MFG is that each node is connected by precisely one **signal** edge to a unique node in another process. This is not the case with **event** edges. If a node has more than one **event** edge leaving it branching has occurred and the program had to make a choice during code execution as to which path it is going to take. If a node is on the receiving end of more than one **event** edge this means that code execution from different nodes bring the system to the same code execution, in this manner conditional statements and loops can be represented (see 5.4.2 simple Message Flow Graphs vs. Conditional Message Flow Graphs). Two requirements set forth in [LEUE 4] to make an unambiguous formal interpretation are as follows:

Traces are Interleavings. The semantics are a precise determination of which execution traces a description allows. The interleaving model, in which a trace is an interleaving of all observable atomic events of the system which is consistent with the linear ordering of events within each process, from the point of view of a global observer. What this means in essence is that each process has its own ordering of events that are based upon where they occur on its axis, the further down the axis the later in time they occur (see 4.3 Beyond the Basics). The global observer can see all events of a process and all possible combination of events between all processes. A trace is a possible path through these combinations.

Finite-State Semantics. Finite-state interpretations define the set of global system states, and alternatively an automaton, whose accepted language is identical with the set of system traces allowed by the specification. Only finitely many global system control states can be identified because of the explicit information available in a MFG. The limit on the number of these states is the size of the cartesian product of the state spaces of the individual processes. The reader whom is interested in more information on this can review [LEUE 4].

5.2 Simple Message Flow Graphs

Since the sending and receiving of messages are asynchronous the system state is derived by parallel composition of all the process states. Message sequence charts are representative of the passing of messages between processes and not the computations done within the processes. The MFG in this

example does not contain conditions (a notion introduced later), we therefore call it a simple MFG. We define the state of a process as either in a state of receiving a message or sending a message.

A **Process State** will have the following notation $(P\varkappa, \mu, \lambda)$:

1. Each process will have a name $P\varkappa$ where $\varkappa$ is a positive integer.
2. Each process will be in a state of sending (!) or receiving (?) represented by μ.
3. A message λ will be represented by a lower case alpha character.

For example (P1,!,u) represents process P1 in a state of waiting to send a message u. Process P2 in a state of ready to receive message x can be represented by (P2,?,x). The receiving of a message is dependent on a message of the same type previously sent but the origin of the received message is not known.

A **System State** will be represented by a composition of parallel process states. The initial system state for Figure 14 is: $S_0=\{(P1,!,a)(P2,?,a)(P3,?,b)(P4,?,c)\}$.

A **State Transition Function** is represented by $<S_i,p,S_j>$ where $S_{i,j}$ are system states and p is the process state completed.

Figure 14: Message Flow Graph

The representation of a MFG is a graph structure whose nodes are representations of message **send** and **receive** events. The two possible edges in a MFG are ***next event*** edges (***ne***) and ***signal*** (***sig***) edges, representing explicit relations on the nodes. The solid edge arrows represent the ***next-event*** relation, indicating the next node in the same process and dashed arrows correspond to the ***signal*** relation, indicating from which node and to which node a message is passed.

5.3 Message Flow Graph Definition

Let S,C and X denote arbitrary pairwise disjoint sets, the elements of which we call *sending* events, *receiving* events and *extra* nodes. Furthermore, let *ST* and *ET* denote arbitrary disjoints sets (also disjoint for S, C and X), whose elements we call *signal* and *event* types. We define a *Message Flow Graph* as a tuple

$$\Im = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$$

Where $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node labels and $(S \cup C, sig, stype, ST)$ is a digraph with edge labels satisfying the following conditions.

1. $sig \subseteq S \times C$ is a (necessarily bipartitie) bijective relation where $S = domain(sig)$ and $C = range(sig)$
2. The set $ET = (\{!, ?\} \times ST) \cup \{Top, Bottom\}$ contains the ***event types*** (we write $!t$ for $(!,t)$ and ?t for $(?,t)$).
3. If the type of a signal is t, then the corresponding send and receive events are of type $!t$ and $?t$ respectively: $(a,b) \in sig$ then $(\exists t \in ST)(stype((a,b)) = t \wedge etype(a) = !t \wedge etype(b) = ?t)$
4. Every component of the ne relation graph contains at most one start event: $(e, e' \notin range(ne) \wedge (e,e') \in ne^*) \rightarrow (e = e')$

This definition comes from [LEUE 4].

5.4 The Translation of Message Sequence Chart to Message Flow Graph

For a simple MSC in graphical form, the mapping is straight forward and the corresponding simple MFG structure could be regarded as just syntactic sugar. The MSC standard Z.120 defines notions like instances (the processes' control flows), message output and message input symbols.

- Instances are graphically represented by a vertical line, called instance axis. The intersection of a departing arrow to an instance axis is an output symbol, and the intersection of an incoming arrow and an instance axis an input symbol.

Given a MSC in graphical form we define a set of sending events S each element of which corresponds to a message output symbol and a set of consuming events C each element of which corresponds to a message input symbol. We call the arrow connecting a message input and a message output symbol a message symbol. For a simple MSC (sMSC), we identify it with its MFG by identifying elements of S and C with their graphical MSC representation if they correspond in the above sense.

5.4.1 Mapping Message Sequence Chart to a Simple Message Flow Graph

Let $ne \subseteq (S \cup C) \times (S \cup C)$ denote a ***next-event*** relation and let $sig \subseteq S \times C$ denote a signal relation such that $(x, y) \in ne$ ***iff*** y is a direct successor of x on some instance axis and $(v, q) \in sig$ ***iff*** v and q are connected by a **message** symbol.

5.4.2 simple Message Flow Graphs vs. Conditional Message Flow Graphs

Simple MFG

An MFG is ***simple*** (SMFG) if the following conditions are satisfied:

- There is no branching in the ***ne*** relation

- There are no cycles in the ***ne*** relation

- There is no self-sending

- All elements in some component are reachable from the start node

- For any signal type, there is a unique sender and a unique receiver process

Conditional MGF

In section 4.3.1 we described MSC conditions as stated in [ITU-T Z.120]. There is a syntactic composition operation defined in [LEUE 1] which allows MFGs to be 'joined' at these conditions.

This composition operation will enable more than one possible joining, which produces the effect of non-deterministic choice in MFGs. A non-terminating-loop-like behavior can be obtained by writing the same condition at the beginning of a loop and the end of a loop as can be seen in Figure 15.

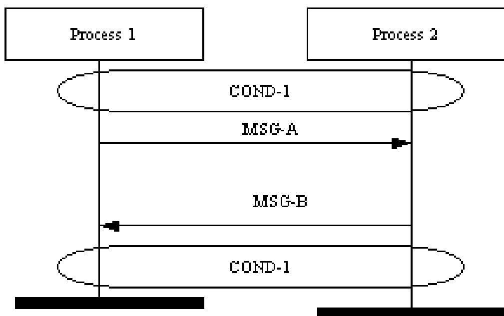
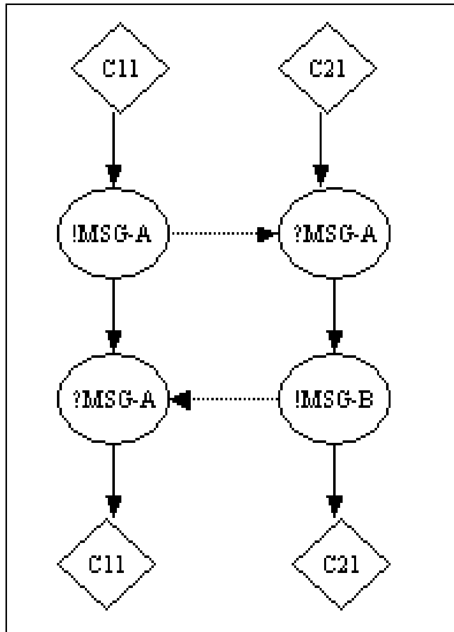


Figure 15 : Looping MSC

A loop in an MFG is simply a cycle in the next-event (solid-arrow) relation. This representation precludes the nodes as representing events, since in a trace of the MFG they may be transverse multiple times. MFG nodes should be thought of in the manner of statements of a programming language, which may be executed multiple times, each execution of which is a message-passing event.

Using the MSC specification in Figure 15, the corresponding MFGs with conditions, represented by the diamond-shaped nodes, is derived in Figure 16. The idea is that control arriving at the condition may continue in another MFG from a condition node with an identical label, as if the MFGs were 'joined' at these condition nodes. The 'unfolding' of the MFGs with conditions into a single structure (pbMFG), can be seen in figure Figure 17. The composition of cMFGs according to a composition graph yields a single graph called a pbMFG. Any paths through this pbMFG correspond to system traces.



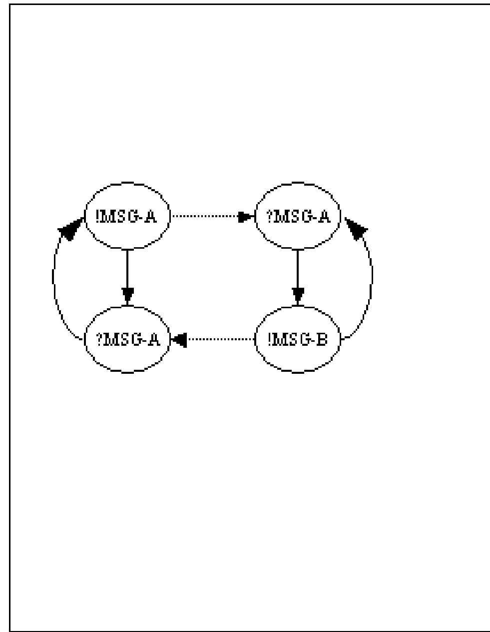


Figure 16 : cMFG

Figure 17 : pbMFG

5.5 MFG to a pbMFG

Composition is defined by two cMFGs only. We obtain a pbMFG such as can be found in Figure 17 from making all compositions possible from the cMFGs. The unfolding operation on a set of MFGs which composes a cMSC with all possible successors, as defined in [LEUE 1], intuitively done by taking the composition graph and 'pluggin in' each actual cMFG (without its initial and terminal condition nodes) in the appropriate place. The result of this operation is a pbMFG with branching and cycles, and provides us with a singe finite structure, a pbMFG, corresponding to the original set of cMFGs.

5.6 MFGs to Global State Transition Diagrams

Sets of Message Sequence Charts and analyses of parallel code yield sets of cMFGs. Using unfolding, we may represent the set of cMFGs by a single pbMFG. In order to obtain a finite-state automaton from such a pbMFG, we have to define the global states, the start state and the state transition

function. This triple define the global state transition graph (GSTG). A requirement for this process is that there must be a finite number of global states [LEUE 1].

5.6.1 Obtaining the Global States, the Start State, and the Transition Relation

The global state of an MFG is determined by the local state of each of the processes, and by the "state" of each of the messages. Graphically, global states are certain sets of edges of the MFG, and the transition relation between states is obtained by deleting particular edges from the state and adding others. As noted before, the next event edges (***ne***) occurring in a state may be thought of as the set of positions where control lies in each process, and the signal edges (***sig***) may be thought of as ***signals sent by not yet received***. The start state $S_0=\{(j,m),(i,n),(k,p),(h,r)\}$ is simply the set of edges leading from ***TOP*** nodes in the graph. In Figure 18, additional syntax has been added to Figure 14 to help clarify the methodology of obtaining the global states and transition relations.

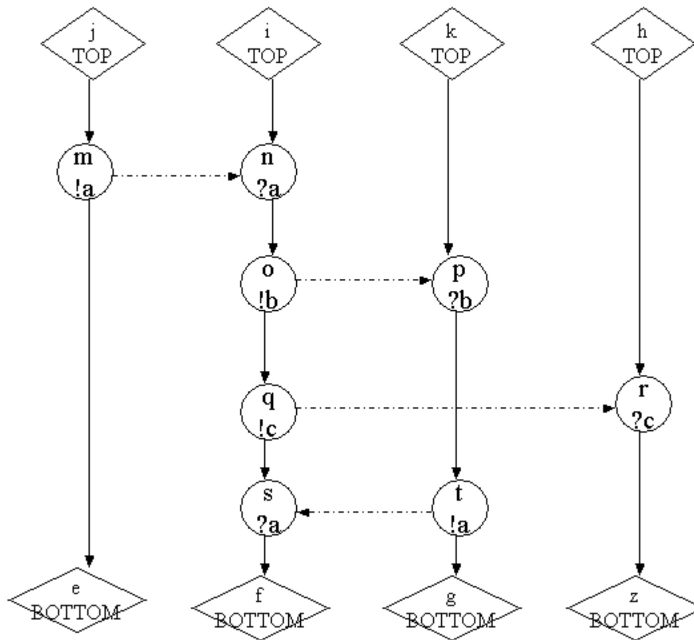


Figure 18 : MFG with node labels and messages

In this state the event of type ***!a*** at node ***m*** is enabled, because node ***m*** represents a send node. Node ***n*** is not enabled, because the send corresponding to it has not been taken. Since ***m*** is enabled, the

event corresponding to it may be taken, i.e. Executed, next to give a new state S_1. The triple < S_0,***m***, S_1> will thus be a member of the transition relation. The new state S_1 is obtained by omiting the ***ne***-edge ***(j,m)*** and adding the ***ne***-edge ***(m,e)*** to the state (to represent the change in location of the 'program counter' of the first process), and adding the ***sig***-edge <***m,n***> to represent the ***a*** signal sent but not received. In new state S_1 ={(***m,e***),(***i,n***),(***k,p***),(***h,r***),<***m,n***>} the node ***n*** is now enabled. Node ***n***, is a receive node and as such it requires not only that its 'program counter' be at the right position (i.e. a ***ne***-edge (***i,n***) with ***n*** as second coordinate is in the state), but also the signal has be sent (i.e. a ***sig***-edge <***m,n***> with ***n*** as second coordinate is also in the state). When the action corresponding to node ***n*** is taken, the edges <***m,n***> and (***i,n***) are removed from the state S_1, and (***n,o***) is added to represent the advancement of the program counter. The resulting state is S_2 ={(***m,e***),(***n,o***),(***k,p***),(***h,r***) }, with transition relation <S_1,***n***, S_2 >. Node ***o*** is enabled in S_1. To see the complete graph and transition states see Appendix A.

5.6.2 Enabling and State Transitions for Branching MFGs

Unlike Figure 18 many systems will involve branching MFGs (Figure 19). To accommodate branching one has to do a little more than in the above example.

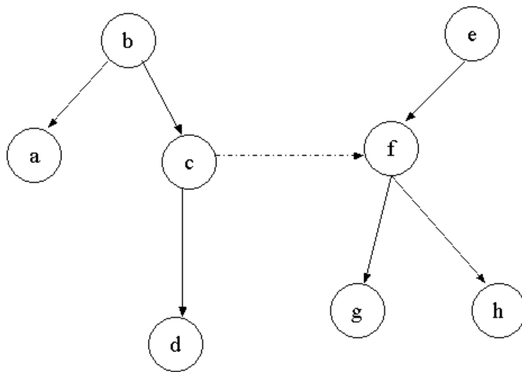


Figure 19 : Part of an MFG with Branching

Assume that the graph is a small part of a larger pbMFG, and that ***a*** is a send event. All arrows in the chart belong to the ***ne*** relation except for the pair <c,f> which belongs to the ***sig*** relation. Consider the global system state G={(b,a),(b,c)(e,f)}. Send events a & c are enabled. If event c occurs, then the

system will advance the 'program counter' by omitting the edge (b,c) from G and adding all outgoing ne edges of node b, in this case (c,d), to the successor state G'. In addition you must also remove the possibility of choosing the enabled action represented by a, which represents a choice alternative to the occurrence of the c event. Hence the edge (b,a) is removed from G as we transit via c to G'. You also have to represent the sending event x leaves a message in transit, and thus the sig edge <c,f> is added to G to from the successor state G'.

Chapter 6

CROSS VERIFICATION OF SPECIFICATIONS

The proceeding chapters have described in detail state transition diagrams and message sequence charts. To be able to do cross verification of these specifications there must be some relationship that connects them. In this chapter we explore the need for this connecting relationship and what correlations exist between the two specifications. An algorithm for cross verification is presented for the development of evolving systems. The algorithm is developed directly from the work of Stephan Leue [LEUE 1, LEUE 2, LEUE 3, LEUE4] but instead of creating the automaton we take the abstraction of the GSTG and search for it as a sub-graph of the original State Transition Diagram.

6. Specifying Evolving Systems

The success of a piece of software is not based on its complexity or simplicity. Speed and accuracy of algorithms within a program do not necessarily determine its success. What truly determines the success of software is its ability to meet the needs of a customer. The first step in producing software is to gather its requirements. Once these requirements are gathered they are put together in the form of a specification. Usually these specifications start as natural language descriptions of a system. However, there are several methods of writing out the specifications of software in formal language to lessen ambiguous, inconsistent and incomplete specifications. Formal description techniques (FDT), like State Transition Diagrams (STD), Specification and Description Language (SDL), Structured Analysis and Design Technique (SADT), Object-Oriented Analysis and Design (OOAD) and Message Sequence Charts (MSC) are used daily in the software industry. There are tools available that support development and formal verification of many of these FDTs . Increasing the power of a verification system can be done by taking one verified view, mapping it to the other view and running verification on it. An example of this can be found in the telecommunications industry, where tools have been developed that will take MSC and convert it to a SDL and vice-versa.

This approach produces a new view of the system to be used in the development process. It tends to lend itself to the view that systems are stagnant and do not evolve. In this thesis we are concerned with evolving systems that are either under development or being re-engineered with new functions being added to the system. This thesis concentrates on a development process where a complete system has a verified State Transition Diagram and a function is to be added to the system and is written as a Message Sequence Chart. In this type of development, consistency between the formal description techniques must be maintained by validating the system specifications against each other.

6.1 Consistency in Software Specifications

In the beginning of computer programming, most software could be developed by a single individual. With the development of bigger more powerful computers, software has become larger and more complex. Software development now generally consists of many teams working on individual modules that are brought together to form the complete software package. With Object Oriented Analysis and Design teams can work on objects that have a myriad of types of messaging going on within, but they only need to pass on to the other teams the types of messages the object itself can receive and send. If this type of information is not consistent across the specification then the approach will fail. It is obvious to see that if Object A can only receive messages of type {a,b,c} but Object B is trying to send messages of {x,y,z} to it then the development process is flawed and the program will not work. The specification and development of objects could be excellent but if TEAM 1's objects and TEAM 2's objects do not have the some common grammar there would be no means of coalescing or verifying them with each other.

6.1.2 Specification Breakdown

When discussing about Message Sequence Charts it is natural to take a look at Object Oriented Analysis and Design. OOAD deals with the messaging between objects and is often represented by some type of Message Sequence Chart. In OOAD objects are in essence "Black Boxes" that have the interfaces to send and receive messages, but hide the inner workings of the object. When designing the system you are not interested in "how" the object does its work (or the messaging going on inside the object), where the only the messages it can receive and send. Objects themselves can be made up of other objects that communicate with each other. Object A could be made up of Object A1 and Object A2 that communicate with each other.

6.1.3 Message Sequence Charts vs. State Transition Diagrams

Message Sequence Charts depict quite well the communication between processes. They do not show with what is happening within the processes but only the outside communication. In like manner, State Transition Diagrams do not show what is occurring within a state but indicates how the system behaves in response to external events. In order to be able to validate one against the other a transformation must occur so that they are depicting the same type of behavior. In [LEUE 4] the process of transforming a MSC into a global state transition graph is described in fine detail.

6.1.4 Abstraction of Global State Transition Graphs

The derivation method described in [LEUE 4] can produce a very complex GSTG with a conservative upper bound on the number of states being the size of the cartesian product of the state spaces of the individual processes. It will be a very special case when there is a one-to-one correspondence between the GSTG and the STD. A mechanism is needed to be able to relate the GSTG to a STD. In general what is needed is a less complex version of the GSTG, in a sense a summary. What is needed is an abstraction of GSTG that has just as much computational power as the GSTG but is less complex. An abstraction of GSTG A is an GSTG A' such that the states of A' are a subset of the states of A and alphabet of A ' is based on sequences of letters, (i.e. Words) from A. What we propose is a correlation between MSC and STD where the MSC global conditions are a subset of the states of the STD.

Example of an Abstraction:

The Global State Transition Graph for the Message Flow Graph specifications in Figure 20 is shown in Figure 21. We can form an abstraction, Figure 22, of the of the GSTG by only showing information of state sets {S0,S6},{S2},{S11} which are dependent upon conditions C1={C11,C12},C2={C21,C22} and C3={C31,C32}.

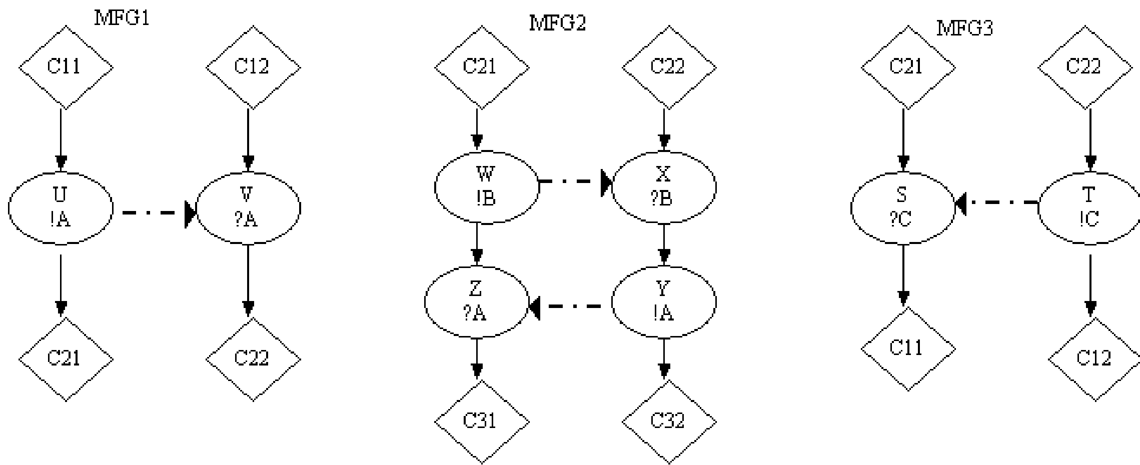


Figure 20 : Set of Message Flow Graphs

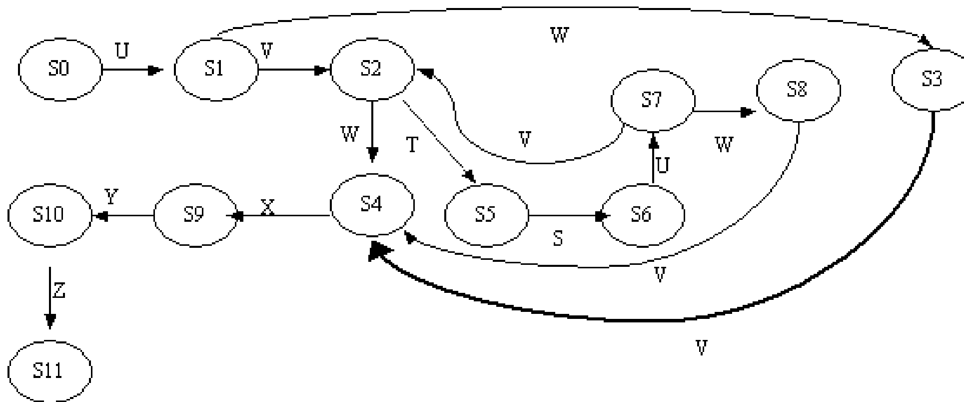


Figure 21 : Global State Transition Graph

Definition of Abstraction:

(A',M,f) is an ***abstraction*** of A iff

- *M* is a mapping of the state set of *A* into the set of states of *A'*. (i.e. *M* picks out a subset of the states of *A* which correspond with a particular state of *A'*)

- *f* maps the alphabet of *A* into words representing the alphabet of *A'*. (i.e. sequence of transitions among states in *A* are translated into a transition of *A'*)

- ***m(n)*** is a path from some state in ***I***(s) to some state in ***I(s')*** in *A'* iff there is a transition from ***s*** to ***s'*** on ***n*** in *A.*

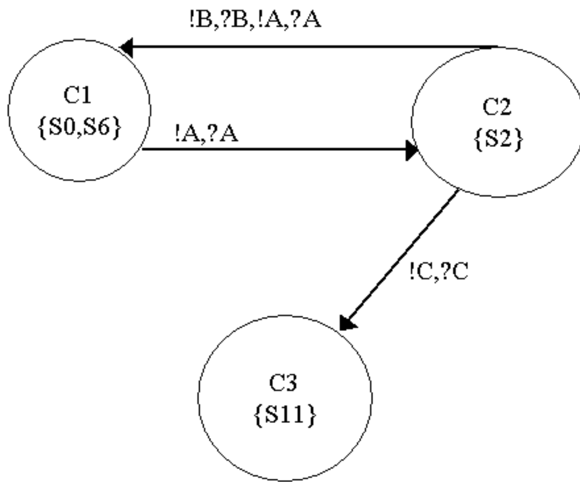


Figure 22 : Abstraction of Global State Transition Graph Figure 21

In order for specifications to be verified against each other they must have some common connections. We theorize is that if a correlation is drawn between the global conditions of the MSC and the states of the STD then the abstraction ***G'*** of the derived GSTG could be verified against the STD. The most obvious correlation is to have the Global Conditions in the MSC be a subset of the States in the State Transition Diagram.

6.2 The Methodology

With evolving specifications or even software projects that have several types of specifications there is a need to validate the specifications against each other. We have been concentrating on message sequence charts and state transition diagrams because MSCs are widely used within the telecommunications industry and many formal language descriptions can be transposed into STDs.

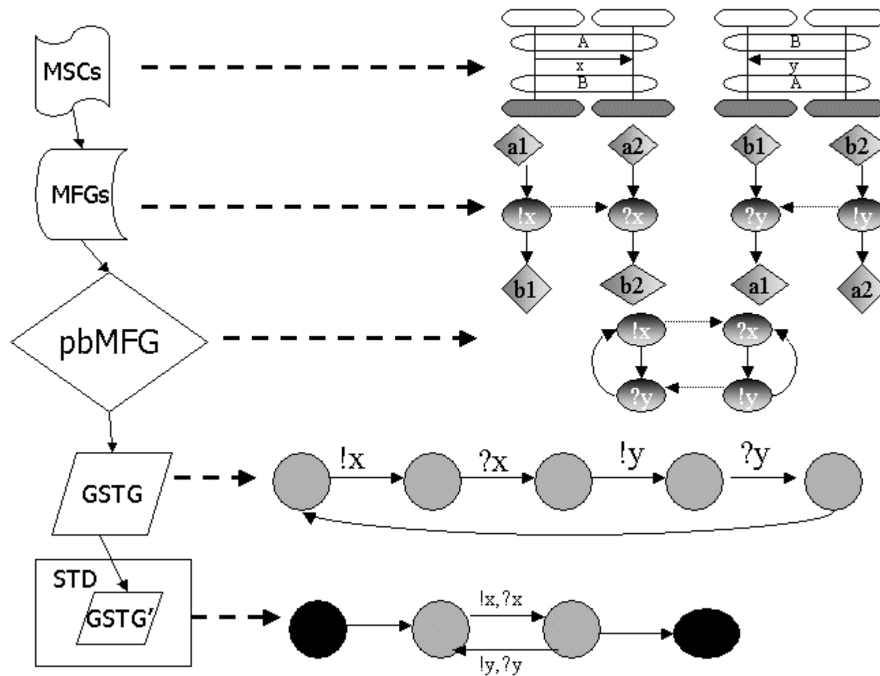


Figure 23 : Algorithm Pictured

We assume that a State Transition Diagram (S) exists for a system and that it has been verified. A new function is to be added to the system and has been specified with Message Sequence Charts. The MSCs are transformed into Conditional Message Flow Graphs. We then unfold these cMFGs to create a pbMFG. The pbMFG is the basis for the needed Global State Transition Graph. Taking into account the Global Conditions as seen in the MSCs and related to the Global State Transition Graph an abstraction of the GSTG *C'* is derived by collapsing paths through the GSTG. Verification that the function will fit into the original STD is accomplished by searching T for a subgraph C.

6.3 The Example

As an example of this method let us return to the description of a CD Player as outlined in ***3.3.1 State Transition Tables***. In Figure 24 the events have been changed to messages so that they can be related to the message sequence charts that describe the new function is to be added to the system.

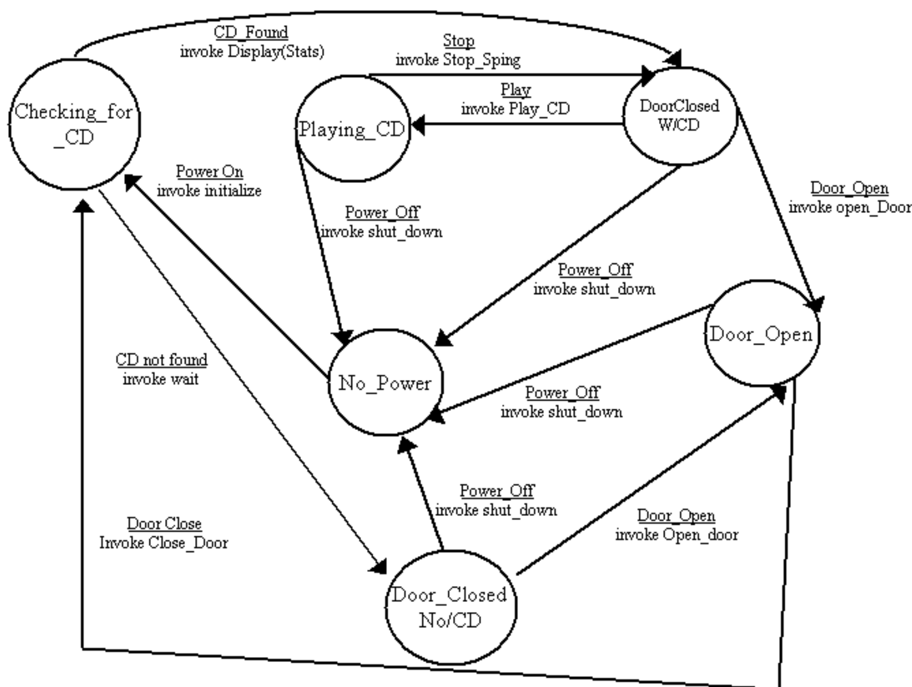


Figure 24 : CD State Transition Diagram with Messages

As a natural language description when the Power is turned the following will happen:

A power on automatic play function is to be added to the CD player. When the CD player receives power it will close the CD player door if it is open and if a CD is present display its stats on the LCD screen and play the CD. If no CD is present then the CD player will display "No CD" on the LCD screen.

6.4 The Corresponding Functional MSC

To make the above natural language description clearer and more precise we will use Message Sequence Charts to show communication between processes. The simple MSCs and their derived cMFGs that correspond to the above function specification can be seen in Appendix B. Only the processes that have some type of messaging occurring are included. It is obvious that the entire set of processes could be added to the graph but this would clutter the graph and make it less readable. There are two different types of conditions that a process can go through either a global system state

(global condition) referring to all instances contained in the MSC or a state referring to a subset of instances (non-global condition). Since we are only concerned with the global conditions and their relation to the states of the STD they are the only ones depicted on the MSCs.

6.5 pbMFG to Global State Transition Graph

Unfolding the cMFG we get the pbMFG as depicted in Appendix B Figure 36. The next step is to create the state chart depicted by this pbMFG. This process was explained in ***5.5 MFG to a pbMFG*** and in the example there are branching alternatives that must be taken into account when resolving the states. We will concentrate on the occurrence of the (No_Power,F). In Appendix B there is a table of Global System States that can be used in reference to the following dialog. When the program code represented by F occurs then a choice is made within F as to which program path it takes (i.e. what message it sends/receives and to/from whom). We are not concerned with how F makes it decision only that a branching occurs. To indicate that the enabled event (No_Power,F) is taken in S_{14} we must eliminate <D,F> and (No_Power,F) as well as add all the out going edges (F,G) and (F,J) giving us S_5. The state transition function therefore looks like < S_{14},F, S_5>. If in S_5 the enabled event (F,G) is taken we must do more than only add the events (G,DoorClose_NoCD) and <G,H>, we must eliminate the possibility of taking the other enabled events. The edges (A,I), (F,J) and (No_Power,L) must be eliminated as we move from S_5 to S_7, < S_5, H,S_7>. Once the transition function is defined and the global states are defined we can now create the GSTG as depicted in ***Figure 25 : Abstraction Z' of CD player***GSTG**.**

6.6 Abstraction of the GSTG

The structure of the abstraction depends on global states represented by the global conditions (No_Power, Checking_For_CD,Door_Close_W_CD, DoorClose_NoCD, Playing_CD) of the MFG. We are able to collapse pathways through sets of sates and create state sets $\{S_0\}$, $\{S_5\}$, $\{S_6, S_{10}, S_{11}, S_9, S_{12}\}$, $\{S_8\}$, $\{S_{13}\}$ respective to the global conditions.

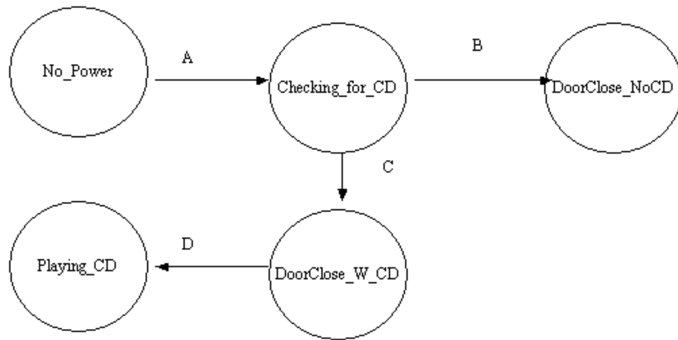


Figure 25 : Abstraction *Z'* of CD player GSTG

Alphabet Abstraction of CD player GSTG based on words from the alphabet of GSTG.

A=(!Power_Button,?Power_Button,!Close_Door,?Close_Door,!Hello,?Hello)

B=(!Display(No_CD),?Display(No_CD))

C=(!Display(Stats),!Play_CD,?Play_CD^?Display(Stats))

D=(?Play_CD^?Display(Stats))

In so much that we defined the alphabet for *Z'* to be {A,B,C,D} we could have just as easily have defined it to be {Power_On, CD_Not_Found, CD_Found, Play}. This type of user definition of the alphabet allows us to make a correspondence between *Z'* and the STD. Verification of the system can now take place by using any practical algorithm (i.e. breadth first search) to search the state transition diagram for a sub-graph that corresponds to the abstraction graph. If the sub-graph is found then the verification is complete, otherwise verification has failed.

7. Concluding Remarks

Software systems have become larger and more complex with time. Software engineering methods are constantly being improved to handle the large and evolving specifications for these types of systems. There are many different types of specifications that represent varying system attributes. Telecommunications has become a large part of the software field and often uses Message Sequence Charts as a specification, along with Specification Description Language. We proposed a method of verifying MSCs against an arbitrary STD (which a SDL can be transformed into). This method had much of its basis on the works of Peter Ladkin and Stefan Leue [LEUE 3,LEUE 4]. This method has several drawbacks and areas where more research needs to be done:

- Global State Transition Graph can quickly become complex (with an upper bound on the number of states being the cartesian product of the process states).
- Branching in Message Flow Graphs can cause the pbMFG to become complex
- The defining of states for the abstract GSTG which depend on corresponding sets of states from the GSTG depends on the individual and is not automatic

The algorithm for defining the message flow graphs and global state transition graph are straight forward and the complexity of them probably can be overcome with brut force (i.e. make a program that will handle the details.)

The last drawback of defining the states for the abstract GSTG which are dependent upon corresponding sets of states from the GSTG at this time must have user interaction. There does not seem to be concrete logic as to how to develop the abstract set of states. Logic can define some of the states easily. For example when the states only have one path way from C.s to C.s' (see Figure 26 : Singlar path collapsed), this will easily map to just one set state {{s},{s'}} which can be done recursively. Where more research is needed is in the case of cycles. If there are cycles within the GSTG then the user must determine what states make up the set (see Figure 27 : Cycle collapsed). This might be able to be made automatic but more information would have to be saved within the GSTG, i.e. where the global conditions lie within relation to the states (see Figure 28 : GSTG with Condition notation).

In the end user interaction is need. The user must determine which states are end-states, and the mapping of the alphabet from the abstraction to the messages in the STD. It is noted that this mapping is not a logical necessity but rather a cognitive one. It makes it easier for the reader to understand how the specifications are related and verified against each other. It is our opinion that some user interaction is will always be preferred in order that the cognitive understanding of the cross verification is solidified.

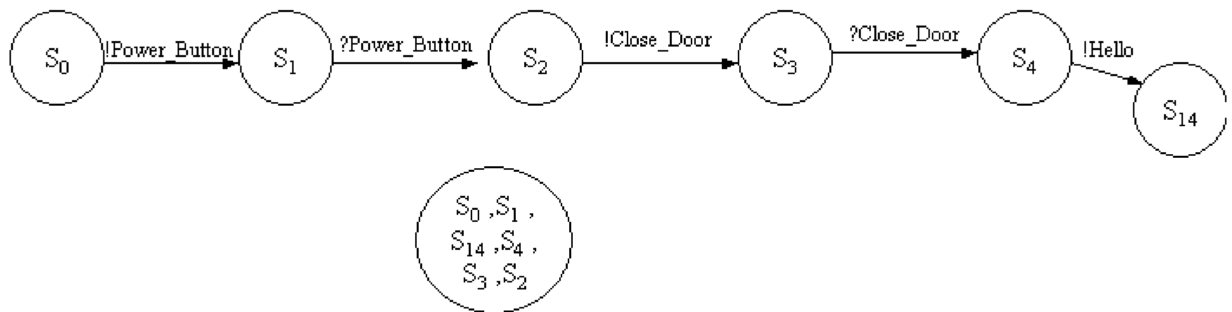


Figure 26 : Singlar path collapsed

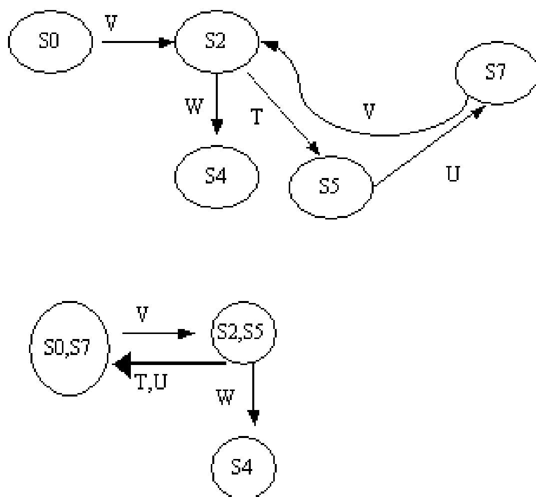


Figure 27 : Cycle collapsed

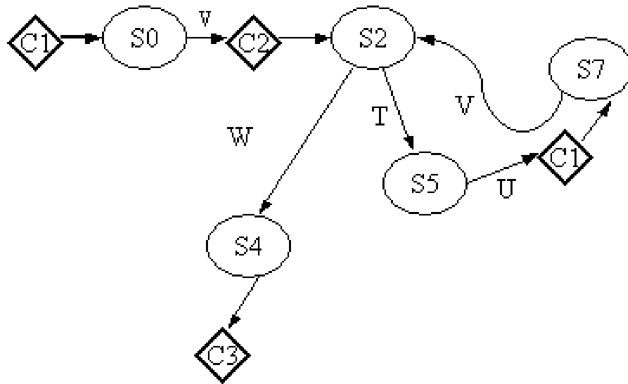


Figure 28 : GSTG with Condition notation

As with must research it only points you into directions were more research is needed. Some open questions that our research on cross verification pointed to are the following:

- How much information is retained about conditions within the GSTG?
- How much more information is need to be able to logically choose the set of states for cycles?
- How does the type of communication (synchronous vs asynchronous) effect the GSTG and the abstraction?

Appendix A

Appendix A shows the derived Gobal System States and transition relations based on MFG in Figure 18. The Global State Transition graph is then derived from Table 2.

GOBAL STATES	By Node Label	By event Label
$S_0=\{(j,m),(i,n),(k,p),(h,r)\}$	$<S_0,m,S_1>$	$<S_0,!a,S_1>$
$S_1=\{(m,e),<m,n>,(i,n),(k,p),(h,r)\}$	$<S_1,n,S_2>$	$<S_1,?a,S_2>$
$S_2=\{(m,e),(n,o),(k,p),(h,r)\}$	$<S_2,o,S_3>$	$<S_2,!b,S_3>$
$S_3=\{(m,e),(o,q),<o,p>(k,p),(h,r)\}$	$<S_3,p,S_4><S_3,q,S_5>$	$<S_3,?b,S_4><S_3,!c,S_5>$
$S_4=\{(m,e),(o,q),(p,t),(h,r)\}$	$<S_4,t,S_6><S_4,q,S_7>$	$<S_4,!a,S_6><S_4,!c,S_7>$
$S_5=\{(m,e),(q,s),<q,r>,<o,p>,(k,p),(h,r)\}$	$<S_5,p,S_7><S_5,r,S_9>$	$<S_5,?b,S_7><S_5,?c,S_9>$
$S_6=\{(m,e),(o,q),(t,g),<t,s>,(h,r)\}$	$<S_6,q,S_8>$	$<S_6,!c,S_8>$
$S_7=\{(m,e),(q,s),<q,r>,(p,t),(h,r)\}$	$<S_7,t,S_8><S_7,r,S_{10}>$	$<S_7,!a,S_8><S_7,?c,S_{10}>$
$S_8=\{(m,e),(q,s),<q,r>,(t,g),<t,s>,(h,r)\}$	$<S_8,r,S_{11}><S_8,s,S_{12}>$	$<S_8,?c,S_{11}><S_8,?a,S_{12}>$
$S_9=\{(m,e),(q,s),<o,p>,(k,p),(r,z)\}$	$<S_9,p,S_{10}>$	$<S_9,?b,S_{10}>$
$S_{10}=\{(m,e),(q,s),(p,t),(r,z)\}$	$<S_{10},t,S_{11}>$	$<S_{10},!a,S_{11}>$
$S_{11}=\{(m,e),(q,s),(t,g),<t,s>,(r,z)\}$	$<S_{11},s,S_{13}>$	$<S_{11},?a,S_{13}>$
$S_{12}=\{(m,e),(s,f),<q,r>,(t,g),(h,r)\}$	$<S_{12},r,S_{13}>$	$<S_{12},?c,S_{13}>$
$S_{13}=\{(m,e),(s,f),(t,g),(r,z)\}$		

Table 2

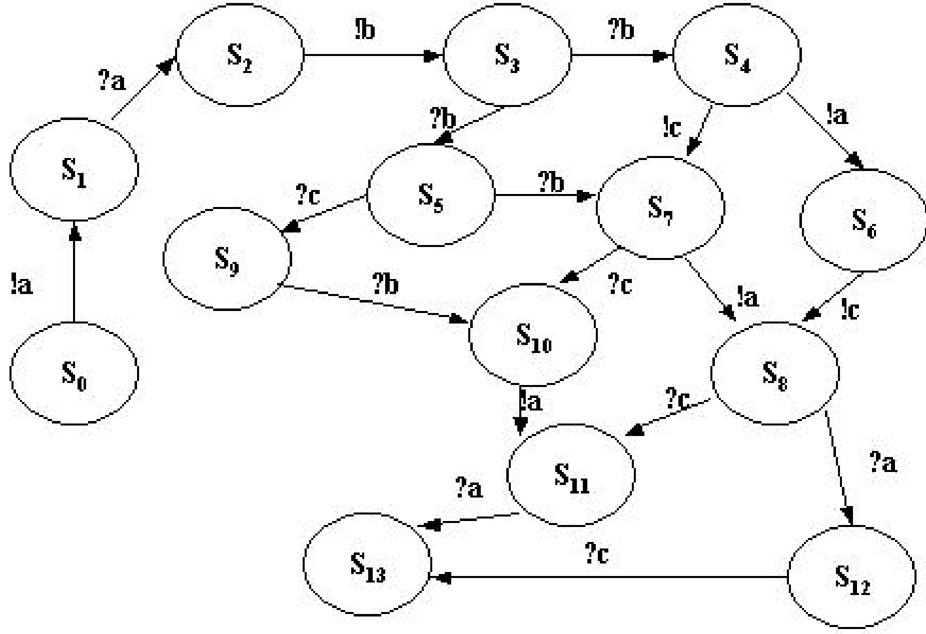


Figure 29 : Global State Graph

Appendix B

Appendix B shows the derived MSCs and cMFGs that correspond to the function specified in Section 6.4.

Derived MSCs

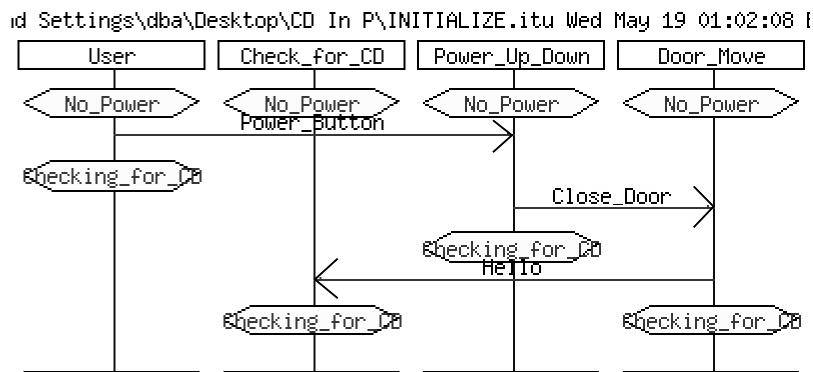


Figure 30 : MSC Initialize

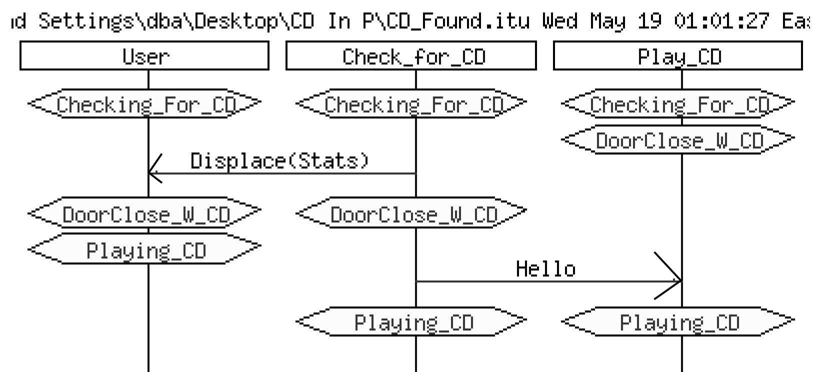


Figure 31 : MSC for CD Found

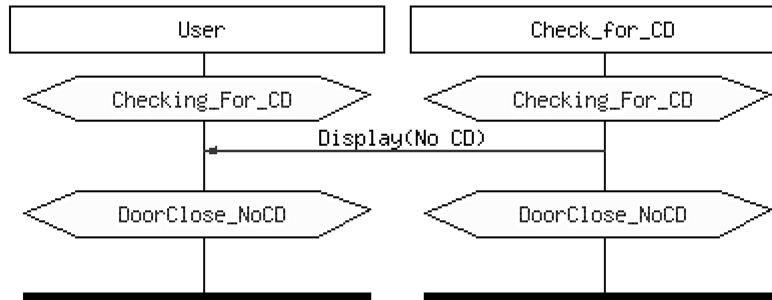


Figure 32 : MSC for No CD found

Derived cMFGs

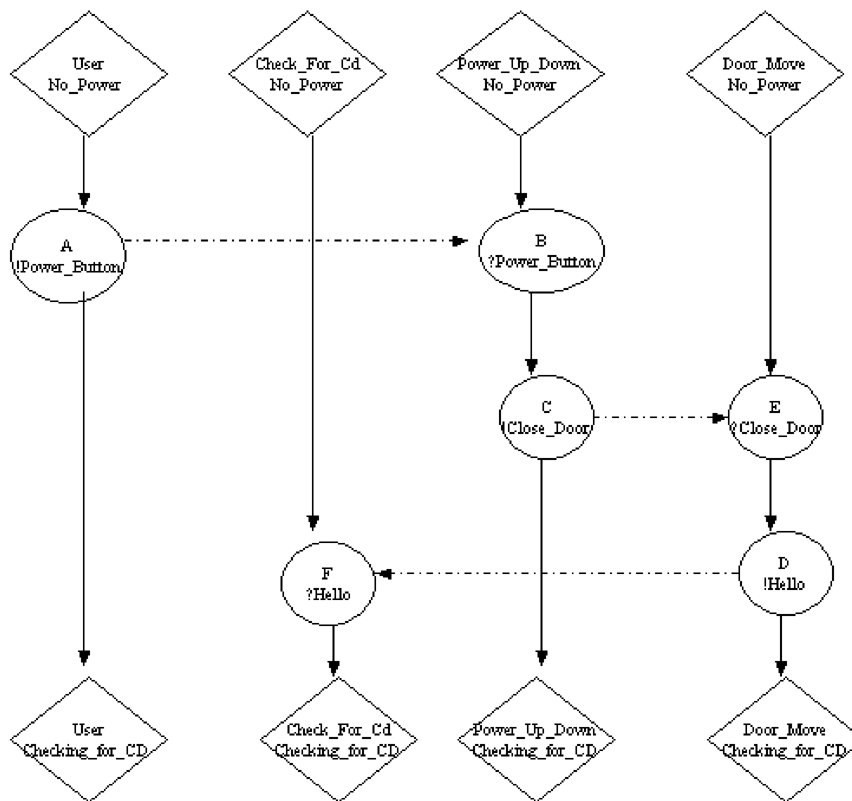


Figure 33 : MFG derived from Figure 30 : MSC Initialize

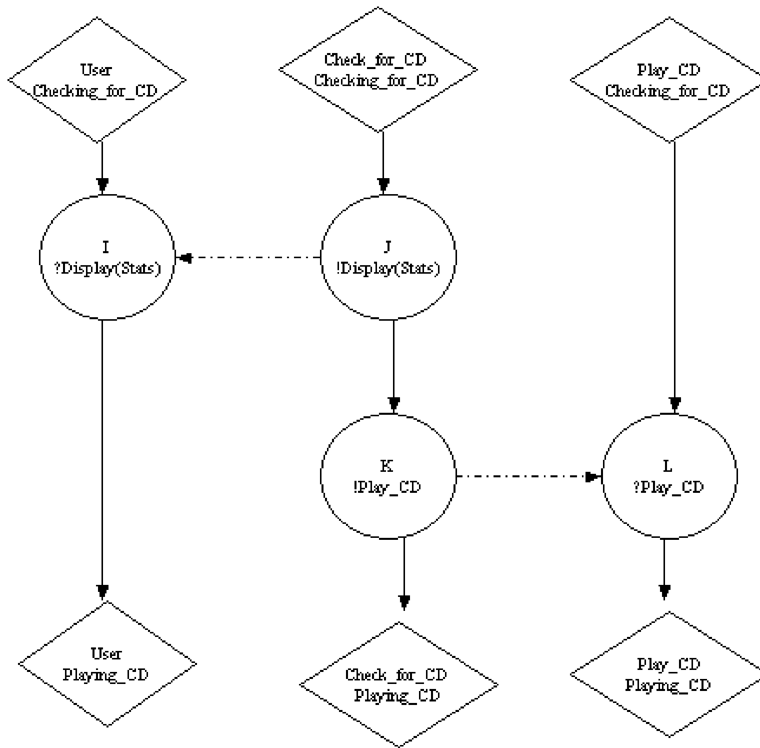


Figure 34 : MFG derived from Figure 31 : MSC for CD Found

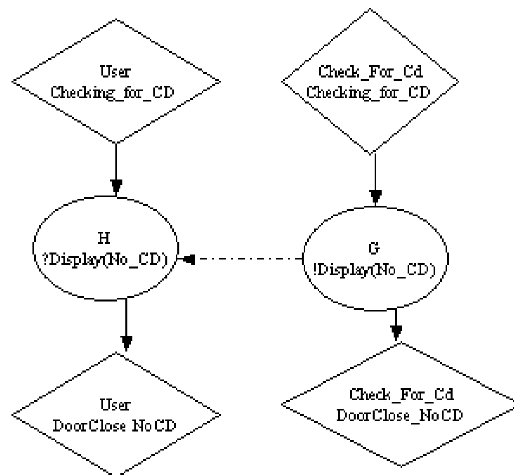


Figure 35 : MFG derived from Figure 32 : MSC for No CD found

pbMFG

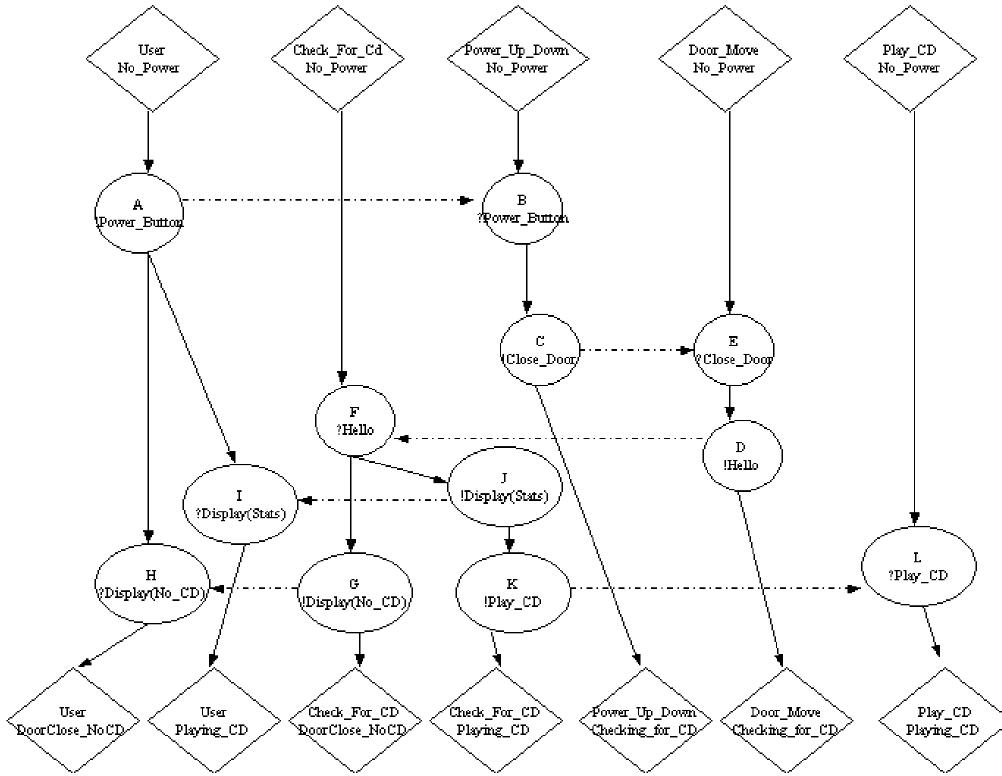


Figure 36 : pbMFG

Global System States

S_0={(No_Power,A),(No_Power,F),(No_Power,B),(No_Power,E),(No_Power,L)}

S_1={(A,H),(A,I),<A,B>,(No_Power,F),(Power_Up,B),(No_Power,E),(No_Power,L)}

S_2={(A,I),(A,H),(No_Power,F),(B,C),(No_Power,E),(No_Power,L)}

S_3={(A,I),(A,H),(No_Power,F),(C,Checking_for_CD),<C,E>,(No_Power,E),(No_Power,L)}

S_4={(A,I),(A,H),(No_Power,F), (C,Checking_for_CD), (E,D),(No_Power,L)}

S_5={(A,I),(A,H),(F,J),(F,G),(C,Checking_for_CD),(D,Checking_for_CD),(No_Power,L)}

S_6={(A,I),(J,K),<J,I>,(C,Checking_for_CD),(D,Checking_for_CD),(No_Power,L)}

S_7={(A,H),<G,H>,(G,DoorClose_NoCD),(C,Checking_for_CD),(D,Checking_for_CD)}

S_8={(H,DoorClose_NoCD),(G,DoorClose_NoCD),(C,Checking_for_CD),(D,Checking_for_CD)}

S_9={(A,I),(K,Playing_CD),<K,L>,<J,I>,(C,Checking_for_CD),(D,Checking_for_CD),(No_Power,L)}

S_{10}={(I,Playing_CD),(J,K),(C,Checking_for_CD),(D,Checking_for_CD),(No_Power,L)}

S_{11}={(I,Playing_CD),(K,Playing_CD),<K,L>,(C,Checking_for_CD),(D,Checking_for_CD),(No_Power,L)}

S_{12}={(A,I),(K,Playing_CD),<J,I>,(C,Checking_for_CD),(D,Checking_for_CD),(L,Playing_CD)}

S_{13}={(I,Playing_CD),(K,Playing_CD),(C,Checking_for_CD),(D,Checking_for_CD),(L,Playing_CD)}

S_{14}={(A,I),(A,H),(No_Power,F), (C,Checking_for_CD), <D,F>,(D,Checking_for_CD),(No_Power,L)}

State Transitions

<S_0,!Power_Button, S_1>,<S_1,?Power_Button, S_2>,<S_2,!Close_Door, S_3>,<S_3,?Close_Door, S_4>

<S_4,!Hello, S_{14}>,<S_5,!Display(Stats), S_6>,<S_5,!Display(No_CD), S_7>,<S_7,?Display(No_CD), S_8>

<S_6,!Play_CD, S_9>,<S_6,?Display(Stats), S_{10}>,<S_9,?Play_CD, S_{12}>,<S_9,?Display(Stats), S_{11}>

<S_{10},!Play_CD, S_{11}>,<S_{11},?Play_CD, S_{13}>,<S_{12},?Display(Stats), S_{13}>,<S_{14},?Hello, S_5>

Global State Transition Graph

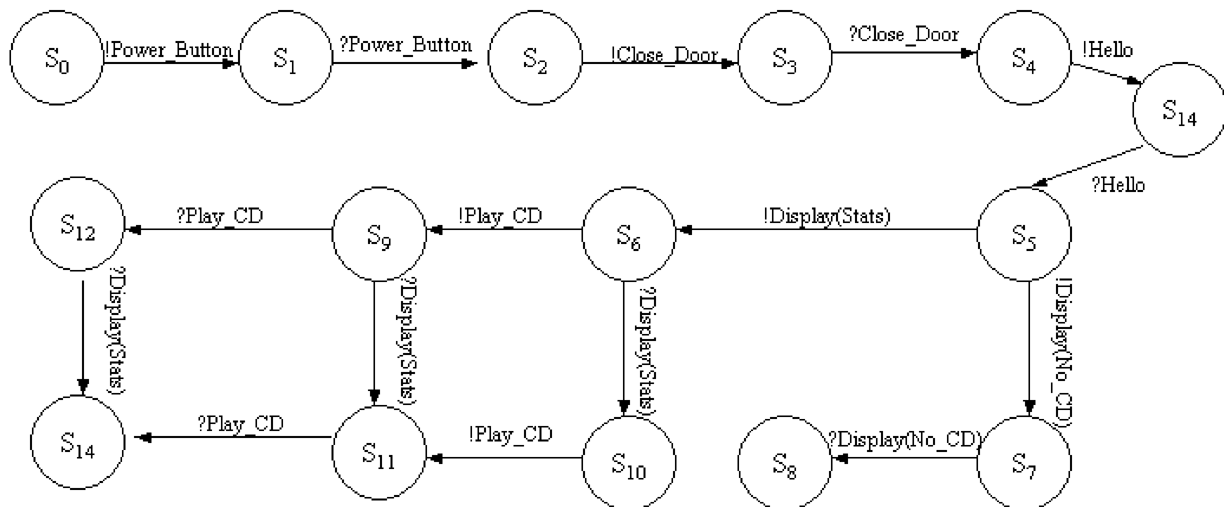


Figure 37 : CD player GSTG

BIBLIOGRAPHY

[ABDALLA99] M.M. Abdalla, F. Khendek and G. Butler, "New Results on Deriving SDL Specifications from MSCs", Proceedings of SDL Forum '99, Elseview Science B. V., R. Dssouli, G.V. Bochmann and Y. Lahav (eds.), Montreal, Canada, June 21-25, 1999

[ABDALLA96] H. Ben-Abdallah, S. Leue, "Syntactic Analysis of Message Sequence Chart Specifications", Electrical and Computer Engineering, University of Waterloo, Technical Report 96-12, copyright Hanene Ben-Abdallah and Stefan Leue, November 1996

[ANDERSSON] M. Andersson, J. Bergstrand, "Formalizing User Cases with Message Sequence Charts", Master Thesis, Department of Communication Systems, Lund Institute of Technology, May 1995

[DAAE] L. Daae, J. Castelein, K.H. Lam, http://panoramix.univ-paris1.fr/CRINFO/dmrg/MEE98/misop033/#12, viewed: April 9, 2001

[DAVIS] A.M. Davis, "Software Requirements Analysis and Specification", Prentice-Hall International Editions, 1990

[GAO] U.S. Government Accounting Office, "Contracting for Computer Software Development – Serious Problems Require Management Attention to Avoid Wasting Additional Millions", Report FGMSD-80-4 November 1979.

[GAO2]U.S. Government Accounting Office, "New Denver Airport: Impact of the Delayed Baggage System", RCED-95-35BR October 1994

[ITU-T Z. 100] ITU-T Specification and Description Language (SDL). Recommendation Z. 100, 1992 Geneva

[ITU-T Z. 106] ITU-T Common Interchange Format for SDL. Recommendation Z.106, 2000, Geneva

[ITU-T Z.120] ITU-T Message Sequence Chart (MSC). Recommendation Z.120, 1996, Geneva

[LEBLANC] Simulation, Verification, and Validation of Models, Verilog White Paper, 1998

[LEUE 1] S. Leue and P.B. Ladkin. Implementing and Verifying MSC Specifications Using Promela/XSpin. In: J.-C. Grégoire, G. Holzmann and D. Peled (eds.), Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System. DIMACS Series Volume 32, American Mathematical Society, Providence, R.I., 1997

[LEUE 2] S. Leue, "Methods and Semantics for Telecommunications Systems Engineering", PHD Thesis, Universitat Bern, January 19, 1995

[LEUE 3] S. Leue, P. Ladkin, "What do Message Sequence Charts Mean?", Proceedings of the Sixth International Conference on Formal Description Techniques, North-Holland, 1994

[LEUE 4] S. Leue, P. Ladkin, "Interpreting Message Flow Graphs", Formal Aspects of Computing, 37(9), January 1995

[MILLER] H. W. Miller, "Reengineering Legacy Software Systems", Digital Press, 1998

[PRESSMAN] R. S. Pressman, "Software Engineering -A Practitioner's Approach", McGraw-Hill 1997.

[ROBERT] G. Robert, F. Khendek, P. Grogono, "Deriving an SDL Specification with a Given Architecture from a Set of MSCs", SDL '97: TIME FOR TESTING –SDL, MSC and Trends, Elsevier Science B.V., 1997

[VAQUEZ] F. Vazquex, "Using Object Oriented Structured Development to Implement a Hybrid System", Software Engineering Notes, Vol 18, No. 4, pp 44-53