INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Automated White-box Cluster Testing

Using Message Sequence Charts '96

by

Scott Cowan

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

December 1997

© Copyright

1997, Scott Cowan

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26977-9

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

**Automated White-box Cluster Testing Using
Message Sequence Charts '96**

submitted by

Scott Cowan

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

December 1997

Abstract

Many of the leaders in object-oriented technology have devoted much of their time to developing powerful programming languages and intuitive analysis and design methodologies. However, until recently, little time has been spent on object-oriented testing methodologies, let alone on automated object-oriented testing tools.

This thesis presents a new approach to system verification by illustrating a tool which automatically compares test runs against a user-supplied scenario model. This model can be expressed using either a) message sequence charts following the MSC'96 recommendation [MSC 96], or b) a more formal approach using regular expressions. Such a strategy provides a useful technique for white-box cluster testing, but does not constitute a testing methodology in itself.

In this dissertation, I will first give an overview of many of the object-oriented testing techniques being used today, followed by a detailed description of how message sequence charts can be used as a testing mechanism, and conclude with a number of examples demonstrating the usefulness of my proposed tool.

Acknowledgments

I would like to express my thanks to my supervisor Jean-Pierre Corriveau, and his constant attention to my work. I'd also like to thank Gerard Hurdle and Mike Valenta for their support at school everyday. Finally I'd like to thank Kathy Johnstone for keeping me sane in the past year and a half.

Table Of Contents

ABSTRACT	III
ACKNOWLEDGMENTS.....	IV
TABLE OF CONTENTS	V
LIST OF FIGURES.....	VIII
1. INTRODUCTION	1
1.1 MOTIVATION	1
1.2 OBJECTIVES	3
1.3 PROBLEM DESCRIPTION	3
1.4 APPROACH	7
1.5 CONTRIBUTIONS.....	8
1.6 OUTLINE.....	9
2. BACKGROUND.....	11
2.1 OBJECT-ORIENTED CONCEPTS	11
2.2 OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE)	14
2.2.1 Requirement Specification & Object-Oriented Analysis	16
2.2.2 Object-Oriented Design.....	17
2.2.3 Object-Oriented Implementation.....	18
2.2.4 Object-Oriented Testing	18
2.2.4.1 Black-Box Testing	21
2.2.4.2 White-Box Testing.....	22
2.2.4.3 Regression Testing.....	23
2.2.4.4 Testing Levels	24
2.2.4.5 A Simplified View of OO Testing	26
2.2.4.6 About DIVA and DIVeT	27
2.3 OBJECT-ORIENTED SCENARIO MODELING LANGUAGES	30
2.3.1 Object View: Object Message Diagrams.....	31
2.3.2 Behavioral View: Finite State Machines	32
2.3.3 Scenario View: Message Sequence Charts '96	33
2.3.3.1 Graphical Representation.....	35
2.3.3.2 Instance Oriented Textual Specification	37
2.3.3.3 Event Oriented Textual Specification	38
2.3.3.4 High-Level MSC.....	38
2.3.3.5 Optional Construct.....	40
2.3.3.6 Loop Construct	41
2.3.3.7 Alternative Construct	42
2.3.3.8 Reference Construct.....	42
2.3.3.9 Concurrent Construct.....	43
2.3.3.10 Parallel Construct.....	44
2.3.4 Path Expression Groups (PEGs).....	45
2.4 TGEN.....	47
2.4.1 Parse Trace	52
2.4.2 Parse Tree	53
2.4.3 Abstract Syntax Tree	54
2.4.4 MSC Grammar	58
2.5 MESA (A MSC EDITOR).....	62

2.6 TRACE TOOL	63
3. TESTING APPROACHES	65
3.1 OOSE INTEGRATION STRATEGY	67
3.2 AUTOMATING TESTING	68
3.3 ASTOOT	69
3.4 FREE APPROACH TO TESTING OO SOFTWARE	70
3.5 METHOD SEQUENCE SPECIFICATION	71
4. MSC TEST TOOL'S FEATURES AND LIMITATIONS	73
4.1 TYPICAL USAGE	73
4.2 USER AUTOMATICALLY BUILDS A MSC DOCUMENT USING A MSC EDITOR.....	74
4.3 USER INPUTS THE MSC DOCUMENT INTO DIVET.....	75
4.4 USER AUTOMATICALLY BUILDS A TRACE GRAMMAR THAT REPRESENTS THE MSC DOCUMENT.....	76
4.5 USER AUTOMATICALLY BUILDS A TRACE PARSER FOR THE TRACE GRAMMAR	78
4.6 USER EDITS A TEST FOR TESTING ONE USE CASE OF THE SYSTEM	78
4.6.1 User creates a new test.....	79
4.6.2 User edits the description.....	79
4.6.3 User adds the Trace Classes	80
4.6.4 User edits the test code.....	80
4.6.5 User automatically builds an execution trace of the test.....	81
4.7 USER SELECTS A TEST AND PARSES ITS EXECUTION TRACE	83
4.8 LIMITATIONS	86
4.8.1 Restructuring MSC'96.....	86
4.8.2 Naming Instances	87
4.8.3 Other Syntactic Limitations.....	88
4.8.4 Other Limitations	90
5. EXAMPLES.....	93
5.1 MESSAGE PASSING EXAMPLE.....	94
5.2 OPTIONAL CONSTRUCT	96
5.3 LOOP CONSTRUCT.....	98
5.4 ALTERNATE CONSTRUCT	100
5.5 CONCURRENT MESSAGES.....	103
5.6 MESSAGE SEQUENCE CHART REFERENCE.....	105
5.7 PARALLEL MESSAGES	109
6. CONCLUSIONS.....	113
6.1 FUTURE RESEARCH	114
6.2 CONCLUDING REMARKS.....	116
7. BIBLIOGRAPHY	117
APPENDIX A: LOADING AND RUNNING DIVET	120
APPENDIX B: MSC GRAMMAR FOR TGEN.....	121
APPENDIX C: BANKVER1.MSC	133
1 MSC DOCUMENT.....	133
2 TRACE GRAMMAR	135
3 TRACE CLASSES	136
APPENDIX D: BANKVER2.MSC	137
1 MSC DOCUMENT.....	137
2 TRACE GRAMMAR	138

3 TRACE CLASSES	138
APPENDIX E: BANKVER3.MSC.....	139
1 MSC DOCUMENT.....	139
2 TRACE GRAMMAR	140
3 TRACE CLASSES	140

List of Figures

FIGURE 1: EXAMPLE MESSAGE SEQUENCE CHART FOR 1 USE CASE OF AN ELEVATOR SYSTEM	4
FIGURE 2: TRACE OF ONE RUN OF AN ELEVATOR SYSTEM.....	5
FIGURE 3: A SPIRAL MODEL [JACOBSON 92]	15
FIGURE 4: SCENARIO FOR PHONE CALL.....	16
FIGURE 5: CATEGORIES AND ACTIVITIES FOR OBJECT-ORIENTED TESTING	26
FIGURE 6: DESIGN-IMPLEMENTATION VERIFICATION APPROACH (DIVA).....	29
FIGURE 7: AN OBJECT MESSAGE DIAGRAM	32
FIGURE 8: FINITE STATE MACHINE OF AN ELEVATOR	33
FIGURE 9: EVENT TRACE OF A PHONE CALL	34
FIGURE 10: ANATOMY OF A BASIC MSC	36
FIGURE 11: HIGH-LEVEL MSC EXAMPLE OF A TELEPHONE CONNECTION	39
FIGURE 12: PATH EXPRESSION OPERATORS [BRUEGGE83]	45
FIGURE 13: PEGS' PATH EXPRESSION OPERATORS [ADAMS 92].....	46
FIGURE 14: A PATH EXPRESSION GROUP.....	47
FIGURE 15: EXECUTION TRACE FORMAT FOR EACH MESSAGE.....	48
FIGURE 16: TGEN'S REGULAR EXPRESSION OPERATORS [GRAVER 91]	48
FIGURE 17: EXAMPLE TGEN GRAMMAR.....	50
FIGURE 18: TGEN INTERFACE	51
FIGURE 19: A SUCCESSFUL TGEN PARSE TRACE	52
FIGURE 20: AN UNSUCCESSFUL TGEN PARSE TRACE.....	52
FIGURE 21: TGEN'S TRACE OUTPUT	53
FIGURE 22: PARSE TREE FOR 'BABB'	54
FIGURE 23: TGEN'S PARSE TREE OUTPUT	54
FIGURE 24: STRING TO GRAPH TRANSLATOR	56
FIGURE 25: PARSE TREE OF EXAMPLE GRAPH	57
FIGURE 26: MESA (MESSAGE SEQUENCE CHART EDITOR, SIMULATOR AND ANALYZER..	62
FIGURE 27: EXAMPLE PART-OF HIERARCHY FOR A CAR	67
FIGURE 28: EQUIVALENT SEQUENCE TEST CASES [DOONG 94]	69
FIGURE 29: EDITING MSC DOCUMENT	75
FIGURE 30: MSC DOCUMENT MENU.....	76
FIGURE 31: EDITING TRACE GRAMMAR	77
FIGURE 32: TRACE GRAMMAR MENU	78
FIGURE 33: USE CASE MENU.....	79
FIGURE 34: TRACE CLASSES MENU	80
FIGURE 35: EDITING TEST CODE	80
FIGURE 36: TEST CODE MENU.....	81
FIGURE 37: EDITING TRACE.....	82
FIGURE 38: USE CASE MENU.....	83
FIGURE 39: OPEN SAVINGS ACCOUNT MSC.....	94
FIGURE 40: OPEN BANK MSC	97
FIGURE 41: CHECKING ACCOUNT CUSTOMERS MSC	99

FIGURE 42: ALTERNATE PAY MONTHLY INTEREST MSC.....	101
FIGURE 43: MULTIPLE REQUESTS FROM FIRST ACCOUNT MSC	103
FIGURE 44: REFERENCE PAY MONTHLY INTEREST MSC	105
FIGURE 45: CHECKING ACCOUNT PAY MONTHLY INTEREST MSC.....	106
FIGURE 46: SAVINGS ACCOUNT PAY MONTHLY INTEREST MSC.....	106
FIGURE 47: TERM DEPOSIT PAY MONTHLY INTEREST MSC.....	107
FIGURE 48: PARALLEL PAY MONTHLY INTEREST	110

1. Introduction

Many of the leaders in object-oriented (OO) technology have devoted much of their time to developing powerful programming languages and intuitive analysis and design methodologies used to improve the process of developing large scale OO systems. However, until recently, little time has been spent on object-oriented testing methodologies, let alone on automated object-oriented testing tools. This thesis presents a means of partially verifying that an implementation of a system corresponds to the design specifications. In other words, it aims to help in verifying that an object-oriented program actually works the way that it was designed. The approach is called DIVA (Design-Implementation Verification Approach), and the tool developed in conjunction with this thesis is called DIVEt (Design-Implementation Verification Tool). DIVEt automates¹ one specific facet of design-implementation verification, and its output is basically yes or no. Yes: an actual trace² of object interactions corresponds to one of the specified possible interactions; or no: the implementation and the design specifications do not coincide with respect to object interactions.

1.1 Motivation

The motivation for this thesis arose from a number of reasons. Most object-oriented methodologies barely address the issue of object integration, let alone object-oriented testing. Secondly, there are very few commercial, automated, object-oriented

¹ This approach automates test execution, however, the testing strategy and design must be done manually, so this approach should more accurately be called a semi-automated approach.

testing environments available today, and those that do exist mostly address either system black-box testing or white-box object testing. The verification of interactions between objects is typically downplayed if not ignored.

DIVA does not verify that a system outputs the desired data given some input, but instead it focuses on cluster testing [Corriveau 96]. More specifically, it verifies that the objects of a system interact in an expected way, such expectation being captured in the design.³ Let me elaborate. Three models are typically part of the design of a system; the scenario model⁴, the object model⁵ and the behavioral model⁶. The scenario model must be used for this form of cluster verification because

- 1) The object model typically doesn't contain the detail, nor the ordering of messages required.
- 2) The behavioral model (i.e. finite state machines) typically only describes the behavior of each class, not of the whole system, and therefore does not capture the expected interactions between instances of these classes.

My proposed approach to cluster testing is possible because of a recently developed scenario modeling language called Message Sequence Charts '96 (MSC'96) (See section 2.3.2 for a description of MSC'96). MSC'96 was chosen as the scenario modeling language because of three very important properties not found in other scenario

² A trace is a sequence of messages passed between objects captured at execution time.

³ It is important to understand that we do not have to distinguish between valid and invalid system behaviour here: the interaction specification (i.e. the scenario model) captures all expected system behaviour, be it for normal processing and/or exception handling.

⁴ The Scenario model shows details of the actual exchange of messages between objects [Corriveau 96].

⁵ The Object model shows which objects exist and what other objects form their collaborators [Corriveau 96].

⁶ The Behavioral model explicitly shows the internal workings of each object [Corriveau 96].

modeling languages such as event traces [Rumbaugh 91] or interaction diagrams [Jacobson 92].

- 1) MSCs can be represented graphically, but also in an equivalent textual form,
- 2) Both the graphical and the textual specifications of MSC'96 have been standardized, and
- 3) MSC'96 is a generative model: one MSC'96 diagram can capture a multitude of different sequences of object interactions.

1.2 Objectives

Although the use of MSC'96 does not inherently constitute a complete testing methodology, it does allow for a systematic and automatable approach to white-box cluster testing⁷, which is a form of integration testing for OO systems [Corriveau 96]. This thesis is therefore an experiment to see if the proposed approach to white-box cluster testing is possible. If an approach of this kind can be demonstrated to be effective, OO integration testing could be performed more frequently and more effectively, resulting in fewer late-integration disasters [Corriveau 96].

1.3 Problem Description

Given a set of message sequence charts describing the typical interactions within an OO system, the goal is to automatically verify that actual executions of the system (i.e.

⁷ White-box cluster testing is done to check that the actual exchange of messages (from the execution of the code) corresponds to an expected exchange of messages as captured in an interaction diagram [Corriveau 96]. To be more precise, this would be better categorized as grey-box testing.

actual runs of the code) coincide with the MSC specification. The difficulty of manually comparing traces of test runs to MSCs can be illustrated in Figure 1 and Figure 2.

Figure 1: Example message sequence chart for 1 use case of an elevator system

Figure 1 shows the objects involved in an elevator system, and the expected messages passed between objects during a typical run of the elevator system (refer to section 2.3.3 for a detailed description of message sequence charts '96).

Figure 2: Trace of one run of an elevator system

TRACE-- a Floor--7'	RECVS-- #requestUp	ARGS-- '' #requestUp--MAKING A REQUEST FOR A FLOOR
TRACE-- ExternalRequest	RECVS-- #withDirection:floor:	ARGS-- #up ; a Floor--7' ; '' Creating ExternalRequest instance
TRACE-- ExternalRequest	RECVS-- #withFloor:	ARGS-- a Floor--7' ; '' Creating Request instance
TRACE-- an ElevatorControlManager	RECVS-- #processRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #processRequest:	ARGS-- an ExternalRequest(7)(#up) ; ''
#CentralizedDecisionStrategy		
TRACE-- an Elevator(1-A)	RECVS-- #addRequest:	ARGS-- an ExternalRequest(7)(#up);
TRACE-- an ExternalRequest(7)(#up)	RECVS-- #applyTo:	ARGS-- an Elevator(1-A) ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--4' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--4' ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--5' ; a Floor--4' ; CHOOSING-->>> '1-A'
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--5' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--5' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--5' ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--6' ; a Floor--5' ;
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--6' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--6' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--6' ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--7' ; a Floor--6' ;
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--7' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--7' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--7' ;
TRACE-- an Elevator(1-A)	RECVS-- #serviceFloor:direction:	ARGS-- a Floor--7' ; #up ;
TRACE-- a Floor--7'	RECVS-- #startService:with:	ARGS-- #up ; an Elevator(1-A) ;
TRACE-- ExternalRequest	RECVS-- #withDirection:floor:	ARGS-- #up ; a Floor--7' ; '' Creating ExternalRequest instance
TRACE-- ExternalRequest	RECVS-- #withFloor:	ARGS-- a Floor--7' ; '' Creating Request instance
TRACE-- an ElevatorControlManager	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ; ''
#CentralizedDecisionStrategy		
TRACE-- an Elevator(1-A)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- an Elevator(2-A)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- an Elevator(3-B)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- an Elevator(4-B)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- an Elevator(5-C)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- an Elevator(6-C)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- a ServiceElevator(7-S)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- a ServiceElevator(8-S)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest(7)(#up) ;
TRACE-- Request	RECVS-- #withFloor:	ARGS-- a Floor--7' ; '' Creating Request instance
TRACE-- a Door(#closed)	RECVS-- #open	ARGS--
TRACE-- an Elevator(1-A)	RECVS-- #doorStatusChanged:	ARGS-- #open ;
TRACE-- a DoorTimer(#not running')	RECVS-- #startTimer:	ARGS-- 5 ;
TRACE-- an Elevator(1-A)	RECVS-- #carRequest:	ARGS-- a Floor--4' ;
TRACE-- Request	RECVS-- #withFloor:	ARGS-- a Floor--4' ; '' Creating Request instance
TRACE-- an Elevator(1-A)	RECVS-- #addRequest:	ARGS-- a Request(4) ;
TRACE-- a Request(4')	RECVS-- #applyTo:	ARGS-- an Elevator(1-A) ;
TRACE-- an Elevator(1-A)	RECVS-- #doorTimeExpired	ARGS--
TRACE-- a Door(#open)	RECVS-- #close	ARGS--
TRACE-- an Elevator(1-A)	RECVS-- #doorStatusChanged:	ARGS-- #closed ;
TRACE-- a Floor--7'	RECVS-- #endService:	ARGS-- #up ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--6' ; a Floor--7' ;
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--6' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--6' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--6' ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--5' ; a Floor--6' ;
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--5' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--5' ;

TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--'5' ;
TRACE-- an Elevator(1-A)	RECVS-- #travelTo:from:	ARGS-- a Floor--'4' ; a Floor--'5' ;
TRACE-- an Elevator(1-A)	RECVS-- #arriveAtFloor:	ARGS-- a Floor--'4' ;
TRACE-- an ElevatorControlManager	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--'4' ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #elevator:arriveAtFloor:	ARGS-- an Elevator(1-A) ; a Floor--'4' ;
TRACE-- an Elevator(1-A)	RECVS-- #serviceFloor:direction:	ARGS-- a Floor--'4' ; #down ;
TRACE-- a Floor--'4'	RECVS-- #startService:with:	ARGS-- #down ; an Elevator(1-A) ;
TRACE-- ExternalRequest	RECVS-- #withDirection:floor:	ARGS-- #down ; a Floor--'4' ; ** Creating ExternalRequest
instance		
TRACE-- ExternalRequest	RECVS-- #withFloor:	ARGS-- a Floor--'4' ; ** Creating Request instance
TRACE-- an ElevatorControlManager	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- a CentralizedDecisionStrategy	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ; **
#CentralizedDecisionStrategy		
TRACE-- an Elevator(1-A)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- an Elevator(2-A)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- an Elevator(3-B)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- an Elevator(4-B)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- an Elevator(5-C)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- an Elevator(6-C)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- a ServiceElevator(7-S)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- a ServiceElevator(8-S)	RECVS-- #cancelRequest:	ARGS-- an ExternalRequest('4')(#down) ;
TRACE-- Request	RECVS-- #withFloor:	ARGS-- a Floor--'4' ; ** Creating Request instance
TRACE-- a Door(#closed)	RECVS-- #open	ARGS--
TRACE-- an Elevator(1-A)	RECVS-- #doorStatusChanged:	ARGS-- #open ;
TRACE-- a DoorTimer(#not running')	RECVS-- #startTimer:	ARGS-- 5 ;
TRACE-- an Elevator(1-A)	RECVS-- #doorTimeExpired	ARGS--
TRACE-- a Door(#open)	RECVS-- #close	ARGS--
TRACE-- an Elevator(1-A)	RECVS-- #doorStatusChanged:	ARGS-- #closed ;
TRACE-- a Floor--'4'	RECVS-- #endService:	ARGS-- #down ;

Figure 2 shows the actual messages passed between objects in an actual run of the elevator system. The first column shows the object receiving the message, the second column shows the message received, and the third column shows the actual arguments passed with the message. The manual verification that the above trace actually corresponds to the MSC in Figure 1 would be very time consuming. The manual process becomes even more time consuming and error-prone when a large OO project is being tested with numerous MSCs and execution traces.

My first goal is to define a process that supports such a verification. The first step of the process I propose (called DIVA), involves building a set of graphical MSCs. The second step consists of building a parser that can verify that a given sequence of messages does or does not conform to the set of MSCs. The third step involves building a set of tests and extracting an execution trace from a run of each test. The last step involves

comparing the actual sequence of messages to those implicitly defined in the parser. Details follow in the next section.

1.4 Approach

The approach I propose to automate white-box cluster testing is summarized in Figure 6. Let me briefly explain this diagram. Step 1 in DIVA involves building a scenario model (i.e. manually drawing all MSCs) from a use case model as defined by [Jacobson 92].⁸ Step 2 involves converting the scenario model into a trace parser. The first part of step 2 requires a tool that can convert graphical MSCs to an equivalent textual MSC specification like the one overviewed in [Ben-Abdallah 97]. The approach taken to accomplish steps 2b and 2c relies heavily on a public domain package called Tgen [Graver 91]. Tgen is a translator generator developed for the Smalltalk environment. After I worked out some problems in the MSC'96 grammar [MSC 96], I used Tgen to build a translator, called the MSC translator, which can convert a set of MSCs⁹ into an equivalent grammar called a trace grammar¹⁰. (For examples of trace grammars see Appendices C through E). The MSC translator is to be used each time the scenario model (i.e. the set of MSCs) is updated. Tgen is used again in step 2c to build a parser from the trace grammar, which is specific to the developed system. The resulting parser is called the trace parser and will correctly parse any execution trace that conforms to the trace grammar. Step 3a requires the user of my approach to build a test for every MSC in the

⁸ This seems infeasible unless one uses a generative model such as MSC'96. And drawing only 'typical' interactions is not acceptable for exhaustive white-box cluster testing.

⁹ A set of MSCs, in their equivalent textual specification, is called a MSC document as defined in [MSC 96].

scenario model. This test takes the form of a piece of code creating the relevant instances and initiating the interactions between them. An execution trace is then obtained in step 3b for each test and used as input into the trace parser in step 4. The trace parser produces a pass or fail verdict for each test provided. If all tests pass when parsed by the trace parser, then the current set of message sequences correspond to the scenario model. If any of the tests fail, then that test does not correspond to the scenario model, and either the test or the scenario model needs to be reviewed.

1.5 Contributions

There are four major contributions that this thesis provides. The first contribution of this thesis is a grammar representation of MSC'96. More specifically, I started by extending the formal representation of scenario modeling presented in [Adams 92], which follows the familiar framework of regular expressions. I then developed a set of transformations, described in Chapter 5, that can convert a MSC into a regular expression, and combine all of these regular expressions into one trace grammar. The second contribution is a MSC'96 translator. I have used my MSC grammar, Tgen and additional Smalltalk code that I wrote, to develop a translator that can be used to translate a textual MSC document into a trace grammar. (The translator I have built can be modified to translate an MSC document into any other representative Smalltalk object, not just specifically a grammar.) Thirdly, I have developed a trace tool for extracting the sequence of messages passed between objects for a given block of Smalltalk code. This

¹⁰ A trace grammar is a context free grammar that describes expected sequences of object interactions. See section 2.4.4 for a more detailed discussion.

tool builds an event trace from a test execution. Finally, I reworked an implementation-design verification process presented by [Kirani 94] by focusing on inter-object message verification using MSC'96.

1.6 Outline

In the first section of chapter 2, I will introduce some basic object-oriented concepts. This chapter is not intended to be a complete overview of the technology, however if you've never been exposed to it, it will give you a good basis for understanding this thesis. The next section then introduces DIVA by illustrating its role in Jacobson's Object-Oriented Software Engineering (OOSE) methodology [Jacobson 92]. It would be more appropriate to present its role in the Unified Modeling Language (UML) [Booch 97] and [Rumbaugh 97], and in the Objectory Software Development Process [Jacobson 97], however, none of these books have been released yet. Therefore, it will be presented in Jacobson's well established OOSE methodology. I will briefly touch on the iterative approach to developing software and the major steps involved in creating a good object-oriented system: requirements specification, analysis, design, implementation and unit testing, and integration. Section 2.3 presents some background information concerning different scenario modeling languages, focusing mainly on Message Sequence Charts (MSCs). The last section of chapter 2 will introduce a tool heavily used in this thesis called Tgen [Graver 91]. Tgen stands for Translator Generator, and is used to parse text and translate that text into a Smalltalk object. Since my entire testing method is based on comparing text, Tgen is central to my work.

Chapter 3 reviews Jacobson's use case driven testing approach, as well as one automated testing tool by [Doong 94], another by [Binder 96b], and a specification language [Kirani 94].

Chapter 4 is an in-depth look at DIVEt (Design Implementation Verification Tool). DIVEt is the tool developed in conjunction with this thesis, and is a useful aid when trying to understand my proposed automated testing approach. This chapter is presented as a list of steps taken in a typical usage of DIVEt. The order of steps listed in this chapter is not a necessary ordering, however it is the most intuitive and the order I recommend in using DIVEt. (DIVEt was implemented in VisualWorks¹¹ Smalltalk, so any prior experience with this language will facilitate the understanding of the proposed testing approach; however this is not a necessity.)

Chapter 5 gives a number of examples of how a system might be tested with DIVEt. In each example, there is a natural language description of what the test is supposed to do, a graphical representation of what the test should do in the form of a message sequence chart, a textual specification of the MSC, and finally a regular expression representation of the MSC. Each example is intended to shed some light on a different situation one might encounter when designing a system, and how that aspect is converted into a testable form.

Finally, I will conclude with a number of directions that this project suggests, some future research that could be done in this area, and some concluding remarks.

¹¹ VisualWorks is a registered trademark of ParcPlace-Digitalk

2. Background

The purpose of this chapter is to provide some background information required to understand the testing approach described in chapter 4. A brief overview of OO concepts will be followed by Jacobson's OOSE methodology. Next, I will present some testing techniques and definitions followed by a few design documentation "languages" (finite state machines, object model diagrams, message sequence charts and path expression groups). Next I will introduce Tgen - a translator generator tool for VisualWorks Smalltalk, MESA - a messages sequence chart editing tool, and lastly a trace tool used for extracting an execution trace from Smalltalk code.

2.1 Object-Oriented Concepts

A good object-oriented programming (OOP) background is very helpful in understanding this thesis, however if it is a new topic, this section will attempt to briefly describe what OOP is all about. The following is not a complete overview of object-oriented concepts, simply what is required to understand the thesis. More in-depth discussions can be found in [Lalonde 90], [Jacobson 92] and [Rumbaugh 91].

The world can be modeled in a software system with objects. Objects have properties such as height, color, number of legs, flavor, velocity, etc. Some objects can (and should) also perform certain actions. A knife can cut, a basketball player can slam-dunk, a traffic light can change color, a bank can open a new account. A system of objects constantly changes because some objects perform actions on other objects. For instance, a stock boy shelves canned peas, a factory worker assembles furniture, a janitor

cleans floors, a nurse bandages wounds, etc. Object-oriented programming takes this object modeling approach to developing software.

Object-oriented programming is obviously oriented around objects. An object consists of two things, its properties or state, and the actions it can perform or its behavior. An object's state is preserved with instance variables, while its behavior is invoked with methods. Methods are where the code resides. They are similar to functions in C programming in that they take parameters and return a result. However, methods are not simply executed, they are passed with parameters as messages to other objects. An object knows what instance variables describe it and what behavior it has by knowing what type of object it is. In object-oriented programming the type of an object is defined by what class it was created from. For instance, when a carpenter builds a chair he/she knows what a chair requires to be a chair. It must have enough legs to keep it from falling over, a flat surface to sit on, and a back high enough to lean back on. These are the properties that all chairs must have. Chairs also have the ability to be sat upon, stacked and moved around. This is the behavior of any chair. This information is known by the carpenter, but in object-oriented systems we develop classes that describe this information. So more accurately, classes describe certain types of like objects with instance variables and the methods. When an instance of a class is created it has its own storage for its instance variables and it can be sent (i.e. it reacts to) any of the messages defined in its class. So if a new instance of the class chair is created, the properties for that specific chair could be that it has four legs, a one foot diameter seat, and a one and a half foot back rest.

Object-oriented languages also have the valuable property that classes can inherit from other classes. In the examples provided in chapter 5, a bank system is used. A class exists called `BankAccount` which has instance variables *accountNumber*, *balance* and *owner*. Class `BankAccount` also has methods to get and set the values of these instance variables and retrieve a new account number. The classes `SavingsAccount`, `CheckingAccount` and `TermDeposit` inherit these properties and behavior from `BankAccount` by being subclasses. This means that these classes now get the properties and methods defined in `BankAccount` for free. The class structure of an object-oriented system is hierarchical in nature. All of the classes in a an object-oriented system usually inherit from one root class. In Smalltalk that root class is called `Object`. This way every object in the Smalltalk system inherits the properties of objects. This is an important means of code reuse that object-oriented languages boast about. A class structure is analogous to the hierarchical animal kingdom classification system traditionally used in the biology field.

One of the most difficult OO concepts to grasp, and one of the most dangerous is polymorphism. Jacobson defines it as the following,

“Polymorphism means that the sender of a stimulus does not need to know the receiving instance’s class. The receiving instance can belong to an arbitrary class.”
[Jacobson 92]

For example, if there exists a collection of objects, and a method is written to total the number of legs of all the objects in that collection, it need not know what kind of objects it is querying to get each individual count. Therefore, the method may pass the message `#numberOfLegs` to one object which might be an instance of the class `duck`. It may also

pass the same message to an instance of the class dog, person, chair, etc. As long as the instance understands the message, the sender need not know the receiving instance's class. Polymorphism is very powerful, however it can reduce the testability¹² of a system [see Corriveau 96].

Deciding on what classes should be created is not an easy task. A development team can spend months on this job alone for large systems. Understanding how these classes work together can be extremely complicated, which justifies why good design documentation of the system is required. Good analysis and design documentation can greatly reduce the time required to learn the system, either for new developers or a tester.

2.2 Object-Oriented Software Engineering (OOSE)

OOSE is a software development methodology created by Jacobson [Jacobson 92], which is also referred to as the use case driven approach (UCDA).. It involves more than just the development stage. It incorporates the entire software lifecycle where the user-defined system requirements act as the driving force behind the project development. It follows the spiral model as shown in Figure 3, which iterates through the steps of requirement specification, analysis, design, implementation & unit testing and integration testing.

¹² "testability is the relative ease and expense of revealing software faults" [Binder 94]

Figure 3: A spiral model [Jacobson 92]

The spiral model is meant to show that each step in building a large system need not be worked to completion, before moving on to the next step. But instead, each step should be revisited and reworked as knowledge of the system grows. Some previous software engineering approaches suggest that analysis should not begin until requirements specification is complete, and design should not commence until analysis is complete, etc. This technique is not as effective, because as knowledge of a system grows, faults can be found and improvements can be made if a team is allowed to reassess certain steps.

2.2.1 Requirement Specification & Object-Oriented Analysis

Use cases are an excellent means of expressing functional requirements [Jacobson 92]. They were developed by Jacobson, and are quite similar to the idea presented as scenarios by [Rumbaugh 91]. A scenario is a sequence of events executed during a particular run of the system. An example of a scenario for a telephone system is shown in Figure 4.

```
caller lifts receiver
dial tone begins
caller dials digit (5)
dial tone ends
caller dials digit (5)
caller dials digit (5)
caller dials digit (1)
caller dials digit (2)
caller dials digit (3)
caller dials digit (4)
called phone begins ringing
ringing tone appears in calling phone
called party answers
called phone stops ringing
ringing tone disappears in calling phone
phones are connected
called party hangs up
phones are disconnected
caller hangs up
```

Figure 4: Scenario for phone call

Jacobson developed the idea of a use case model. A use case model is used in the requirements specification of object-oriented systems, and consists of a number of use cases. A use case is a natural language description of one typical way an actor might use the system. A use case can be viewed as a

“...description for a set of scenarios, in the same sense that a class is a description for a set of objects. There should be a finite number of use cases for a system, just as there are a finite number of classes, whereas there are usually an infinite number of possible scenarios and objects.” [Rumbaugh 94].

For our purposes, an actor can be anything external to the system that may affect it. An actor could be a human, a machine or a task on another computer outside of the system. “More precisely, an actor is a role played by such an entity. One person can play several roles and thereby represent several actors such as a computer system operator, a database administrator, and an end user” [Rumbaugh 94]. Actors are not part of the system, they are used to “...represent everything that needs to exchange information with the system” [Jacobson 92]. They are therefore not described in detail. The customer deciding what the system should do is responsible for explaining how the users will input information, and how the system should respond. Use cases give a detailed description of how the system responds.

2.2.2 Object-Oriented Design

From this use case model, an approach called use case driven design is used to develop the system. It involves reworking the natural language description into terms of the domain objects¹³. It works well for designing object-oriented systems for a number of reasons. For one, use cases are an effective starting point for designing message sequence charts (MSCs) (For an explanation of message sequence charts refer to section 2.3.2). While use cases informally describe how the system will react to actor intervention, the

scenario model defines how the system will accomplish that reaction. Secondly, the system architecture is typically controlled by what the customers want to do with the system, thus the external perspective of use-cases can guide the creation of the internal viewpoint taken by a scenario model.

2.2.3 Object-Oriented Implementation

Once the design is documented in MSCs (or other scenario model notation), the next step is to implement the code. This step is dependent on the language chosen. Note that object-oriented approaches to software development do not require an object-oriented language. However, because DIVEt was implemented in VisualWorks Smalltalk, it can only be used to test systems that have been implemented in Smalltalk. But DIVA, my proposed method, can be applied to any object-oriented system.

2.2.4 Object-Oriented Testing

Finally the implementation is tested (For a detailed discussion on object-oriented testing refer to [Hajla 97]). Essentially there are three views on testing. The *optimistic view* is the view that an object-oriented approach to software development reduces the need for software testing. The *minimalist view* is that testing is necessary, but because of the object-oriented approach, it will be easier and less costly. Finally, the *expansive view* is that typical object-oriented features such as encapsulation, inheritance and polymorphism require new approaches to attain adequate testing. Also, since code is

¹³ Domain objects are the objects within the system being designed

designed to be reused, it should be “proven” (i.e. well-tested) so that it can be inherited without being retested.

But what do we mean by testing? Myers says that “...the most important considerations in software testing are issues of economics and human psychology.” [Myers 79] Human psychology is an issue in testing for a number of reasons.

Sometimes testing is viewed as “the process of showing that system produces the correct output.” But testing is generally defined as “...the process of executing a program with the intent of finding errors.” [Myers 79] This may just sound like another way of saying the same thing, but it is actually quite different. It is important that the tester not test a program to show that it works; rather, one should start with the assumption that it doesn’t work and that it contains errors.

Testing can actually be viewed as a *destructive* process [Myers 79], which may be why people find it difficult. But this is the attitude that should be taken. A *successful* test is one which produces an error. A test which produces no error, in hind sight, is a waste of effort, and should then be considered *unsuccessful* (This is the economic viewpoint).

Let’s look at the auto mechanic analogy. Suppose your car is not working as well as usual. If the mechanic performs a number of tests that are “successful” and no problems are found, and he/she says that the car is fine, then the customer is worse off for three reasons.

- 1) The car still has the problem.
- 2) The customer is out \$50.
- 3) The customer questions the mechanic’s abilities.

These results are actually quite unsuccessful. If the mechanic had said that the battery is not getting recharged, then the results would be successful, because the mechanic can go ahead and remedy the problem. It is for this reason that tests producing no error are considered unsuccessful, while tests producing errors are considered successful.

Because the process of testing is destructive, Myers suggests that it is best if testing is not be performed by the developer of the implementation under test [Myers 79]. The developer often subconsciously gears tests to prove that the system does work. Even if the developer does successfully demonstrate that the system does what it's supposed to do, the program may still do what it is not supposed to do. For example, let's take a simple program which requires one integer as input - a person's age, and outputs either "legal" - the person is legally allowed to obtain a driver's license, or "illegal" - the person is under age. If the developer shows that for every legal driving age entered into the program, the result "legal" is returned, the program is not proven correct because it may also say that under age drivers are of legal age. This is an error in the program.

Testing is an issue of economics because a good test suite¹⁴ is different for every system. That is, the ideal test suite would test every possible execution of the system under test with as few tests as possible, but in most cases this would require infinite resources and infinite time. Economics comes into play when time, resources and cost are all weighed and the best possible test plan is derived.

Testing activities are divided into two different goals. *Validation* testing is used to ensure that the system being built fulfills the requirements. *Verification* testing

¹⁴ A test suite is a set of test cases designed to "thoroughly" test a system.

determines that the system implemented has been implemented correctly. Jacobson sums it up with the following two definitions [Jacobson 92]:

- Validation: are we building the correct system?
- Verification: are we building the system correctly?

In an attempt to answer these two questions, there are two types of testing exercises. Black-box testing attempts to validate a system, while white-box testing attempts to verify it.

2.2.4.1 Black-Box Testing

Black-box testing involves testing an entity (a single object or a cluster of objects) with no knowledge of its implementation. The tester would then assume that the implementation is a black box into which a user can provide input, and from which the user gets output. Theoretically, the only way to thoroughly black-box test a system is to provide it with every possible combination of inputs, and verify that the corresponding outputs are correct. However, this approach is impossible, because even a trivial program which requires one single integer has an infinite number of inputs. “A fundamental consideration in program testing is one of economics” [Myers 79]. The next best approach to program testing is to maximize the yield on the testing investment (i.e. find the most errors with the fewest tests). One way to maximize the testing investment is to explore boundary conditions. This approach is called *boundary-value analysis*. For example, when testing our legal driving age program which requires only one integer, try values such as 1, 0, -1 to see how it handles young ages, or impossible entries as well as 150, an age that is very unlikely. But do not limit the tests to ones based on input

condition (input space), also consider the *result space*. For example, since this program is used to determine if the person whose age was entered is legally allowed to obtain a driver's license, we could add more economic tests. Let's say one must be 16 to obtain a driver's license in Ontario. Other input values to test could be 15, 16, and 17.

All of these tests have no knowledge of the implementation. If we do take the implementation into account, we can possibly derive more tests with a high probability of producing errors.

2.2.4.2 White-Box Testing

White-box testing is a testing strategy that allows the tester to take the implementation of the entity under test (EUT) into account. That is, the EUT is no longer a black box, but a white box with the implementation details known. A naive approach to white-box testing might be to make every line of code execute at least once. This condition is a necessary one, but in no way is it sufficient for a reasonable white-box test suite [Myers 79]. Another approach may be to test every path of execution in a method. This criteria is sufficient, but infeasible. For instance, if a trivial program consisted of 5 boolean conditions inside a 5 iteration loop. The number of paths of execution required to test would be $(2^5)^5$, which is more than 33 million and quite infeasible, even for a trivial program.

Even if every path of execution were tested, this in no way guarantees a fully tested system. The implementation could still be full of errors. There are three reasons for this. First, this testing approach doesn't guarantee that the program fulfills its requirements. For instance, if a program were to sort a set of number in ascending order,

but instead it sorted them in descending order, exhaustive path testing could not detect this error. Second, exhaustive path testing does not detect the absence of paths. Third, exhaustive path testing does not detect data-sensitive errors. For example, if a line of code read

IF ((A-B) < EPSILON) ...

both paths could be tested with no error revealed, when the line should actually compare EPSILON to the absolute value of A-B. The detection of this error is dependent on the values of A and B.

2.2.4.3 Regression Testing

DIVeT is a nice tool to support regression testing, because it of its automated nature. A regression test is done when a change has been made to the system. Roughly put, the purpose of regression testing is to verify that no new faults have been introduced by the change (see [Hajla 97] for a detailed discussion of regression testing). Since development typically follows the spiral model, regression tests can be done at each version (refer to Figure 3). They could also be done in much finer steps such as every major change within a version. Therefore, each test developed should be saved in a test suite and versioned with each version of the implementation. These tests can then be reapplied whenever regression testing is necessary. Since regression testing can be done quite often, it can become tiring and time consuming. This is why automating testing can be very effective. Some testing tools that automate testing are presented in chapter 3.

2.2.4.4 Testing Levels

The various methods of testing can be applied at different stages or levels of testing. The following is an outline of these different levels.

- **Unit Testing** involves testing a single unit of code, usually a class in OO development, but on large scale systems this could be a cluster of objects. When unit testing, in order to account for collaborating objects, sometimes it is necessary to incorporate other classes into the tests. Another way is to provide stubs¹⁵ that act as the interfacing classes. At this level, black-box and white-box testing are useful in finding faults.
- **Integration Testing** involves assembling tested units and verifying that they work together correctly. This can best be driven by the use cases developed in the analysis phase. “When we have certified the blocks and possibly also the service packages, integration testing is best done on a per use case basis. Hence, the use cases now form the major tool that drives the integration testing” [Jacobson 92]. Jacobson’s use case driven approach can then be applied at each step in the software lifecycle. “Integration testing is thus performed by testing each use case one at a time, both from an internal viewpoint [white-box testing] and an external viewpoint [black-box testing]” [Jacobson 92]. The internal viewpoint is based on the scenario model, while the external viewpoint is based on the requirements model.
- **System Testing** typically concerns assembling the entire system and testing the application from the user’s point of view. Some focuses for system testing include

- *operation testing* which involves testing the system under normal operation for a longer time.
- *overload or stress testing* to see how the system behaves when it is subjected to overload.
- *ergonomic testing* is important to see how usable the system is by non-computer professionals.
- *acceptance testing* is performed by the customer as a final check. This is a chance to see how the system works within its real environment.

A common approach to integration testing in OO is called *bottom-up* testing [Jorgenson 94]. It involves testing small units until they are shown to be reliable, and adding on to these units to cluster test them. More units and clusters are integrated together and shown to be reliable until finally the entire system gets tested. It is an alternative to *big bang* testing which involves assembling an entire system after completing unit tests, and hoping that everything works fine together. “Rather than testing each module in isolation, the next module to be tested is first combined with the set of modules that have already been tested.” [Myers 79]. This way, the entire system need not be available and running, and fixing problems is less difficult and less time consuming. When errors are discovered, they are easier to locate because less untested code needs to be debugged.

¹⁵ A stub is used in place of unfinished code. It provides an interface and returns values of the expected type, but has no specific behaviour to it.

2.2.4.5 A Simplified View of OO Testing

As we have seen above, testing can be divided into many levels of testing. These levels could even be divided further depending on the size of the project; unit testing could be split into method and class testing, and system testing could be divided into subsystem and system testing. Corriveau simplifies the view of these levels of testing by combining method and class testing into class testing, and combining cluster and subsystem integration into cluster testing [Corriveau 96]. This provides a nice categorization of testing techniques: testing techniques that can be applied to classes, and those that can be applied to clusters of classes, no matter what the granularity of the cluster. Figure 5 is a survey, extracted from [Corriveau 96], of white-box and black-box testing techniques as they fit into the categories of class and cluster testing.

	Class	Cluster
Black-box	<ul style="list-style-type: none"> • names of methods • tests on parameters (ranges, types, etc.) • tests on return values • tests for exception handling 	<ul style="list-style-type: none"> • test that the cluster as a black box produces the correct external behavior (as specified by the corresponding use-case)
White-box	<ul style="list-style-type: none"> • tests on instance variables (eg, initialization, updating, destruction, etc.) • code inspections on methods • path testing for each method • other method testing 	<ul style="list-style-type: none"> • check that the actual exchange of messages (from the execution of the code) corresponds to an expected exchange of messages as captured in an interaction diagram

Figure 5: Categories and Activities for Object-Oriented Testing

It is the activity under white-box cluster testing that this thesis is addressing, by illustrating the approach I developed called DIVA, with a tool I developed called DIVEt.

2.2.4.6 About DIVA and DIVEt

DIVEt facilitates integration testing. It provides a means of verifying the interactions between a cluster of objects against the system's scenario model. More accurately, the presented testing approach provides a means of verifying that design interaction diagrams (in this case message sequence charts) and the actual traces of messages generated by the implementation are in correspondence. DIVA is the method associated with the use of DIVEt. The steps in DIVA are shown in Figure 6 and described below. Each step is explained in more detail in chapter 4.

1. **Build Scenario Model** - User builds a scenario model (i.e. user builds a set of MSCs using a MSC Editor) from the requirements captured in the use case model.
2. **Build Trace parser**
 - a) User automatically converts them to their equivalent textual specification (a MSC document). [as explained in Ben-Abdallah 97].
 - b) User imports the MSC document into DIVEt and automatically converts it into a trace grammar. This trace grammar specifies a language of all the *expected execution traces* of the system. It is a grammar built out of regular expressions, where each regular expression represents one generic MSC (i.e. possibly a large set of possible interactions). Note that both steps 2a and 2b could be automated, however for demonstration purposes, they have be left as a manual step.
 - c) User automatically builds the trace parser from the trace grammar built in step 2c.

3. **Build Test Suite**

- a) Add a test in DIVEt for each scenario created in the scenario model. Each test must be written in Smalltalk code.
- b) For each test automatically build an execution trace.

4. **Parse Each Execution Trace** with the trace parser built in step 3 (Parsing is explained further in section 2.4). If all parses are successful, then the system was implemented as designed and the integration test passed. If the parse of an execution trace was unsuccessful, then the corresponding test failed and one of two things have gone wrong. Either the implementation has changed and the scenario model should be updated to reflect it, or the system implementation did not accurately follow the scenario model, and both should be reviewed.

This process should be repeated during the integration testing phase of iteration.

Figure 6: Design-Implementation Verification Approach (DIVA)

2.3 Object-Oriented Scenario Modeling Languages

An object-oriented program consists of a group of objects working together by invoking each others methods (i.e. sending messages) and keeping track of their current state. A well-designed system of objects documents how these messages will typically be invoked, and thereby describing how the system typically works. There have been a number of proposed ways to document a system's design.

Three views are required to give an overall description of an OO system. The following three definitions are extracted from [Corriveau 96].

- the object (or structure) view shows which objects exist and what other objects form their collaborators.
- the behavioral view is used to explicit the internal workings of each object.
- the communication (or scenario) view details the actual exchanges of messages between objects.

Each of these views can be expressed with different models. Object models typically give a static view of the objects in the system, scenario models illustrate the dynamic inter-object view of each object, while behavior models illustrate the dynamic internal behavior of each object in the system. For our purposes, we need a scenario model that:

- defines the objects involved in the system,
- illustrates expected message sequences between these objects,
- provides genericity; the ability to express more than one scenario,
- can be expressed graphically, and
- can be expressed textually.

The following sections describe three notations for such a scenario model.

2.3.1 Object View: Object Message Diagrams

Traditionally, the object view is illustrated by a static model which tries to capture the entire system as everything happens instantaneously. That is, the model provides no chronological order for the messages between objects. Objects in a system are represented by ellipses or boxes, and directed arcs indicate what messages may be passed from one object to another. Object models may sometimes contain other object models to allow for the description of subsystems. The lack of sequenced events in this model make it unusable for the purposes of white-box cluster testing as defined by [Corriveau 96].

Object message diagrams (OMDs), defined by the unified modeling language (UML) [Rumbaugh 97], describe both the object and scenario view, and do provide sequences of messages. Like MSC'96, OMDs can capture alternative paths of execution, parallel executions, and dynamic communication. But, from our viewpoint, their semantics are not as rich as MSC'96 (e.g. no concurrent regions) and most importantly for this thesis, they do not yet offer a corresponding standardized textual form.

An example OMD can be seen in Figure 7.

Figure 7: An object message diagram

2.3.2 Behavioral View: Finite State Machines

Finite state machines (FSMs) are generally used to illustrate the behavioral view of each object as it changes states. This is often done in a graphical model such as the one shown in Figure 8, but can also be expressed in a regular expression form. Finite state machines have received a lot of attention as a model for testing. For example, Binder's FREE approach to object-oriented testing [Binder 96b], described in section 3.5, is based on finite state machines. These models can be used in an automated testing approach, but since they typically only provide intra-object message sequences for each object, they are not sufficient to test the interactions between objects within a system.

And it is commonly accepted that attempting to perform integration testing by synthesizing the FSMs of individual objects into a FSM for the whole system is

problematic both in terms of scalability and precision (as not all possible interactions are necessarily expected ones).

Figure 8: Finite state machine of an elevator

2.3.3 Scenario View: Message Sequence Charts '96

The scenario (or communication) model is used to illustrate interactions between objects. We will use MSC'96 for white-box cluster testing, because this scenario notation satisfies all five properties we require.

Alternatives to MSCs exist, but typically lack genericity. For example, Rumbaugh introduced the notion of an event trace [Rumbaugh 91]. An event trace simply gives a graphical view of all the messages passed between objects for a given scenario. Figure 9 describes the event trace of a phone call scenario (shown in Figure 4) when a caller telephones a callee [Rumbaugh 91]. The scenario and event trace are respectively very close to Jacobson's use case and interaction diagram.

Figure 9: Event trace of a phone call

Jacobson associates each interaction diagram to a use case. Use cases are meant to describe typical uses of the system not single uses (refer to section 2.2 for a more detailed discussion). Conversely, Jacobson's interaction diagrams only capture a specific set of interactions as is the case with MSC'92 [MSC 92]. For example, every time a user dials a number, the dials message is sent to the Phone Line object. These repeated messages can take up considerable space in an event trace, plus they allow an event trace to only describe one particular path of execution. That is, they only allow the description of the dialing of a specific telephone number (e.g. 555-1234). Conversely MSC'96 can describe a class of event traces. For example, the repetition of the dial messages can be eliminated by using a looping construct. Not only does this construct reduce the size of the message sequence chart and the effort put into creating it, but it allows the MSC to describe many event traces, not just one. Other things lacking from Jacobson's interaction diagrams and MSC'92 are operators to describe optional messages, unordered concurrent messages, alternate paths of execution, and references to other interaction diagrams. These are some of the constructs that MSC'96 offers.

2.3.3.1 Graphical Representation

Message sequence charts '96 is a telephony standard that was proposed by the International Telecommunications Union (ITU), formerly called the Consultative Committee on International Telegraphy and Telephony (CCITT). It is complementary to the ITU specification and design language (SDL) [MSC 96]. The fact that a means of scenario modeling is standardized allows industry to begin building tools that support it, and even allows different tools to possibly cooperate.

Figure 10: Anatomy of a basic MSC

Figure 10 above points out the basic components of a MSC. The outer box around the entire MSC represents the boundary of the system. Therefore, to send messages to the environment (i.e. any external objects whether they be system objects external to the MSC or physical entities external to the system), must be passed via the boundary. This example was extracted directly from Recommendation Z.120 [MSC 96].

An important aspect of MSC'96 is that a graphical MSC may also be represented textually. This means that a MSC can be stored in a text file which takes up very little

storage space compared to its graphical counterpart. An example tool that supports this property is described in section 2.5. There are two different textual specifications of every MSC, one is called the instance oriented textual specification, the other is the event oriented textual specification. They both have all the information required to build the graphical MSC; they simply provide it in different formats. The instance oriented textual specification presents all of the message events from the perspective of each instance, where the instances are listed in any order. The event oriented textual specification presents all of the message events in chronological order. The event oriented textual representation is the representation used by DIVEt, because it is the one that more closely reflects the chronological order of an execution trace. The following is an example of each type of representation of the MSC in Figure 10. A detailed discussion of these formats is beyond the scope of this thesis, but can be found in [MSC 96]

2.3.3.2 Instance Oriented Textual Specification

```

msc connection; inst calling_party: process digite,
called_party: process digite;
    instance calling_party: process digite;
        condition Idle shared all;
        in off_hook from env;
        out dial_tone_on to env;
        in digit from env;
        out dial_tone_off to env;
        in digit from env;
        out seizure_int to called_party;
        in ack from called_party;
        out ring_back_tone_on to env;
        condition Seizure shared all;
        in answer from called_party;
        out connection to env;
        condition Talking shared all;
    endinstance;
    instance called_party: process digite;
        condition Idle shared all;
        in seizure_int from calling_party;

```

```

        out ack to calling_party;
        out internal_ringing_on to env;
        condition Seizure shared all;
        in off_hook from env;
        out answer to calling_party;
        condition Talking shared all;
    endinstance;
endmsc;

```

2.3.3.3 Event Oriented Textual Specification

```

msc connection; inst calling_party: process digite,
called_party: process digite;
calling_party: instance process digite;
called_party: instance process digite;
all: condition Idle;
calling_party: in off_hook from env;
               out dial_tone_on to env;
               in digit from env;
               out dial_tone_off to env;
               in digit from env;
               out seizure_int to called_party;
called_party: in seizure_int from calling_party;
               out ack to calling_party;
calling_party: in ack from called_party;
               out ring_back_tone_on to env;
called_party: out internal_ringing_on to env;
all: condition Seizure;
called_party: in off_hook from env;
               out answer to calling_party;
calling_party: in answer from called_party;
               out connection to env;
all: condition Talking;
calling_party: endinstance;
called_party: endinstance;
endmsc;

```

The important enhancements of MSC'96 not present in Jacobson's interaction diagrams or MSC'92 are discussed below.

2.3.3.4 High-Level MSC

High-level message sequence charts (HMSCs) are used to graphically describe the overall flow of the system. There are no extra constructs available to HMSCs that are not

available to basic MSCs, so they are not a crucial part of MSC'96. What they are used for is enhancing overall system understanding by improving readability.

Figure 11: High-level MSC example of a telephone connection

The example HMSC shown in Figure 11 gives a high-level description of a telephone connection. The top triangle is the start symbol, while the bottom one is the end symbol. The small circle is a connection point used where two paths converge or diverge. The system in the above example begins at the start symbol, and if the connection is disconnected, the flow of execution continues to the connection point. At this point the system attempts to make a connection, or it fails in doing so. If a failure was encountered then the system returns to check the disconnected condition and the process is repeated, otherwise the system verifies that it is connected and comes to the end symbol.

As you will see in the following sections, there are constructs available for basic MSCs that could just as easily be used to illustrate the same information. A loop construct and an alternate path construct are all that's required to describe this HMSC with a basic MSC.

High-level MSCs give a functional description of a system with no explanation of objects or their interactions. This type of notation is similar to data-flow model notation. For a more detailed discussion on data-flow models refer to [Sommerville 95]. From my viewpoint, this feature of MSC'96 does not offer enough detail for white-box cluster testing purposes. It is for this reason, that HMSCs are not dealt with by DIVEt.

2.3.3.5 Optional Construct

In the next paragraphs, an example of each construct not available in interaction diagrams or MSC'92 is given in both the graphical and the event oriented textual formats. For more details of the standard, please refer to Recommendation Z.120 [MSC 96].

MSC'96 has the ability to indicate that a message or a number of messages are optional. That is, the typical usage of the system may or may not consist of the enclosed interactions. This is an important improvement over interaction diagrams.

Graphical **Textual**

```

msc OptionalExample
inst objectX, objectY;
all: optbegin;
    objectX:
        out messageA to objectY;
    objectY:
        in messageA from objectX;
        out messageB to objectX;
    objectX:
        in messageB from objectY;
optend;

```

2.3.3.6 Loop Construct

Message sequence charts can contain constructs that indicate looping as well. This construct allows for the indication of a minimum and maximum number of repetitions with the maximum able to be infinity. This can reduce the time and effort put into creating MSCs as well as reduce the size of them by eliminating repetition of messages.

Graphical **Textual**

```

msc LoopExample
inst objectX, objectY;
all: loopbegin;
    objectX:
        out messageA to objectY;
    objectY:
        in messageA from objectX;
        out messageB to objectX;
    objectX:
        in messageB from objectY;
loopend;

```

2.3.3.7 Alternative Construct

The alternative construct allows the designer to illustrate more than one potential path of execution, where each path of execution is separated by a horizontal dotted line. So, in the following example either messageA is passed otherwise messageB is.

Graphical

Textual

```

msc AlternativeExample
inst objectX, objectY;
all: altbegin;
    objectX:
        out messageA to objectY;
    objectY:
        in messageA from objectX;
alt;
    objectY:
        out messageB to objectX;
    objectX:
        in messageB from objectY;
altend;

```

2.3.3.8 Reference Construct

Reference constructs can reduce the amount of work required to create MSCs by allowing the entire MSC to be broken into subsystems, or more specific MSCs. These subsystems can then be referenced from any other MSC. They also make large MSCs more readable. In the following example, messageC is send by objectX and received by objectY. After that, the reader is referred to the MSC labeled AlternativeExample (in section 2.3.3.7) for the remaining messages invoked.

Graphical **Textual**

```

msc ReferenceExample
inst objectX, objectY;
objectX:
    out messageC to objectY;
objectY:
    in messageC from objectX;
all: reference AlternativeExample;
  
```

2.3.3.9 Concurrent Construct

There are two constructs designed to deal with concurrency. The first construct is called a concurrent region. A concurrent region illustrates concurrency from the perspective of the instance. The second is the parallel construct. A parallel construct illustrates concurrency from the perspective of a process. For example in the following example, objectX may send messageA first, then messageB or vice versa. The same is true for objectY; it may receive messageA first, then messageB or vice versa. All of the events occurring within a concurrent region of an instance, can occur in any order. Another way of viewing a concurrent region is as a “don’t care” construct, because the order of messages sent or received within a concurrent region doesn’t matter. That is, each instance doesn’t care in what order the events occur.

Graphical **Textual**

```

msc ConcurrentExample
inst objectX, objectY;
objectX:
    concurrent;
        out messageA to objectY;
        out messageB to objectY;
    endconcurrent;
objectY:
    concurrent;
        in messageA from objectX;
        in messageB from objectX;
    endconcurrent;
altend;
  
```

2.3.3.10 Parallel Construct

The parallel construct allows for the illustration of more than one thread of execution. In the following example, the **AlternativeExample** MSC executes in parallel with the **LoopExample** MSC. Therefore, nothing is known about the order of message events between the two.

Graphical **Textual**

```

msc ParallelExample
inst objectX, objectY;
all: parbegin;
    all: reference AlternativeExample;
par;
    all: reference LoopExample;
parend;
  
```


2.3.4 Path Expression Groups (PEGs)

Path Expressions were initially introduced by [Campbell 73] as a means of synchronizing concurrent processes. Later, Bruegge and Hibbard [Bruegge 83] used path expressions to define what sequence of messages are acceptable by an object. In other words, a path expression can be used to define a set of possible sequences of messages that an object may receive. The notation used is that of regular expressions similar to the well known Backus Naur Form (BNF) [Hopcroft 79]. The operations used by Bruegge and Hibbard are described in Figure 12.

Operation	Definition
A;B	A is performed then B is performed
A B	Either A or B is performed
A*	A is performed zero or more times
()	Expressions enclosed in parenthesis are treated as a single entity

Figure 12: Path expression operators [Bruegge83]

For example, the following path expression could be used to describe the order of operations performed on a bank account.

PATH open (deposit | withdraw | payInterest)* close END

This means that a bank account must be opened first. After that any number of deposits, withdrawals or interest payments may be made on that account before it is closed, after which it accepts no more messages.

Adams [Adams 92] uses a regular expression notation for describing object-oriented systems similar to that of Bruegge and Hibbard with a couple of minor changes.

Three extra operators are added; the '+', '&' and the '[]' operators. These operators can be seen in Figure 13.

Operation	Definition
A;B	A is performed then B is performed
A B	Either A or B is performed
A+	A is performed one or more times
A*	A is performed zero or more times
A&B	A is performed then B is performed or visa versa
[A]	A is performed zero times or once
()	Expressions enclosed in parenthesis are treated as a single entity

Figure 13: PEGs' path expression operators [Adams 92]

Unlike path expressions, which define the dynamic behavior of individual objects, Path Expression Groups (PEGs) are used to describe the expected interactions between collaborating objects. PEGs therefore capture the important aspects of the object interactions. This happens to be basically the same information that MSC'96 express. The two advantages that MSC'96 has over PEGs are:

1. Messages sequence charts can be expressed both textually and graphically, PEGs can only be expressed textually.
2. Message sequence charts also have the constructs to express concurrency, PEGs do not. Adams only focused on "traditional sequential object-oriented systems" [Adams 92].

```

bank totalDeposits;
bank accounts;
(checkingAccount balance;
| savingsAccount balance;
| termDeposit balance)*;

```

Figure 14: A path expression group

As an example, the PEG in Figure 14 tells us how the typical bank calculates the total amount of deposits by its customers. First something from outside of the bank system sends the `totalDeposits` message to a bank. The bank responds by requesting all of its accounts from itself. The bank then gets the balance of either a checking account, savings account or term deposit from zero or more accounts. This PEG therefore describes one typical usage of the bank system, where the group of objects involved are instances of the classes `Bank`, `CheckingAccount`, `SavingsAccount` and `TermDeposit`. A complete set of PEGs can therefore describe all typical uses of that group of objects. They also form a grammar that defines a language; the language of acceptable sequences of messages that the system typically executes.

2.4 Tgen

MSCs are graphical and textual, while PEGs are strictly textual. Since MSCs can be converted to an equivalent textual specification [Ben-Abdallah 97], textual MSCs can then be translated into a notation quite similar to PEGs. This textual notation is required for the purposes of building a trace parser for a specific set of MSCs. A problem with PEGs in its current form is that it does not capture the sender of any messages. It only captures the methods invoked by specific objects. I propose a textual format that is

similar to PEGs, but in which the sender of a message is also captured. The following example shows the previous PEG in my new format.

```
#totalDeposits, env, bank;
#accounts, env, bank;
(#balance, bank, checkingAccount;
| #balance, bank, savingsAccount;
| #balance, bank, termDeposit)*;
```

This new format eliminates any ambiguity about which object sent which message. Each statement now has the following format of Figure 15, and the regular expression operators are shown in Figure 16.

```
#messageName , sendingObject , receivingObject ;
```

Figure 15: Execution trace format for each message

Operation	Definition
A B	A is performed then B is performed
A B	Either A or B is performed
A+	A is performed one or more times
A*	A is performed zero or more times
[A]	A is performed zero times or once
()	Expressions enclosed in parenthesis are treated as a single entity

Figure 16: Tgen's regular expression operators [Graver 91]

Notice that there is no '&' operator like there is in Adam's notation. Ideally my new notation would incorporate this operator, but my proposed notation must be usable by Tgen which doesn't support such an operator. Describing concurrent systems without the '&' operator is a little more difficult, however a solution is described in section 5.5.

Tgen is the tool we will use to build a translator that will translate MSCs to our new regular expression format, thereby automating the creation of regular expressions to describe our system. Tgen is a parser generator built for the VisualWorks Smalltalk programming environment. The tool is used to create a parser given a context free grammar (CFG). The grammar specified must be presented in a format similar to Backus-Naur Form (BNF). The parser produced by Tgen can then be extended to be a translator. That is, by extending this parser (i.e. writing additional Smalltalk code) it can translate text into an object (e.g. a parse tree). For an in-depth discussion of Tgen refer to [Graver 91].

A Tgen grammar specification must be a CFG, which consists of a sequence of grammar rules similar to those used by LEX/YACC [Lesk 75]. A context-free grammar is informally defined by the following four components. For a more complete and formal explanation of grammars, scanners and parsers refer to [Fischer 91].

- 1) A finite set of terminals; this is the token set produced by the scanner ¹⁶.
- 2) A finite set of different, intermediate symbols, called the non-terminals.
- 3) A non-terminal that starts all derivations called a start symbol or goal symbol.
- 4) A finite set of productions (sometimes called rewriting rules) of the form $A \rightarrow X_1 \dots X_n$, where A is a non-terminal, and $X_1 \dots X_n$ is a sequence of terminals and non-terminals.

For example, a grammar required to create all strings consisting of only the characters 'a' and 'b' is as follows.

1)	Start	:	ABString ;
2)	ABString	:	
3)			'a' ABString
4)			'b' ABString ;

Figure 17: Example Tgen grammar

Line 1 is the starting rule that must be applied stating that every sentence in this language is an ABString. An ABString is one of three things, corresponding to lines 2 through 4;

- 1) an null string,
- 2) an ABString with an 'a' at the beginning, or
- 3) an ABString with a 'b' at the beginning.

So for example, 'a', 'babb' and "" are all sentences in this language. The string 'babb' is constructed by applying the rules numbered: 1, 4, 3, 4, 4, 2. The last rule applied is number 2, because every string ends with a null string. The null string character in Tgen is displayed as '\$' "16r0004". Tgen has the capability of accepting a context-free grammar and building a stand-alone program that can parse any input string to see that it is part of this described language. Figure 18 shows Tgen's interface with the grammar and input string described above. After "accepting" the input string, the top-right text editor tells the user how to view the parse results.

¹⁶ The scanner reads a source program and produces a stream of tokens. The parser then processes the tokens until it recognizes a syntactic structure that requires semantic processing. It then makes a direct call to a semantic routine or method in this case.

Figure 18: Tgen interface

Rules can also include what are called token class specifications. “A token class is defined by giving its token class terminal followed by a colon (:), a regular expression ... an optional scanner directive, and terminated by a semicolon (;) [Graver 91]. For example the token class <name> is a string of one or more sequential lowercase or uppercase characters, while the token class <space> is a string of one or more sequential space, tab, or carriage-returns.

```
<name>      : [a-zA-Z]+ ;
<space>     : [\s\t\r]+ ;
```

The token classes are input into Tgen in the top-left text editor of the interface. Token classes together with the grammar are all that is required to build a parser. Token classes are defined in DIVEt when building a parser from the trace grammar, however, they are hidden from the user.

The output of the parser can be a parse trace of the rules applied, a parse tree, or an abstract syntax tree (i.e. any structured object), as illustrated in the next sections.

2.4.1 Parse Trace

The following is a parse trace of the string 'babb' using Tgen. The output gets displayed in the top-right text editor.

```
LL Parser trace of:  babb

apply production Start      : ABString
apply production ABString   : 'b' ABString
cancel 'b' from input
apply production ABString   : 'a' ABString
cancel 'a' from input
apply production ABString   : 'b' ABString
cancel 'b' from input
apply production ABString   : 'b' ABString
cancel 'b' from input
apply production ABString   :

The parse of the test input was successful.
```

Figure 19: A successful Tgen parse trace

The parse trace of the string 'abc' which is not in this language looks like the following.

```
LL Parser trace of:  abc

apply production Start      : ABString
apply production ABString   : 'a' ABString
cancel 'a' from input
apply production ABString   : 'b' ABString
cancel 'b' from input
```

Figure 20: An unsuccessful Tgen parse trace

The input text box that initially had the 'abc' contains the following error inserted after the character 'c' as in Figure 21.

Figure 21: Tgen's trace output

Basically, it is saying that the terminal 'c' is not part of this language, only the characters 'a', 'b', and the null string '\$' "16r0004" are acceptable input. DIVEt hooks into the Tgen code, and provides its own interface to build parsers. DIVEt reuses the parse trace output, by displaying it in a separate window, to tell the user if a test passed or not. A test passes if the trace parser can successfully parse its execution trace, otherwise the test fails.

2.4.2 Parse Tree

A parse tree is simply the easiest and most intuitive structured object that can be created by a Tgen parser. A parse tree is a tree rooted at the start symbol, that represents each grammar rule applied during the parse, where each tree node consists of a name and an arbitrary number of ordered child nodes. The parse tree generated by parsing 'babb'

can be seen in Figure 22: Parse tree for 'babb', but is displayed textually in Tgen as shown in Figure 23.

Figure 22: Parse tree for 'babb'

Figure 23: Tgen's parse tree output

2.4.3 Abstract Syntax Tree

Parse trees are an intuitive means of dealing with the parsed information and they are a standard output for parsers, however alone they are not very useful because they

have little meaning in that representation. This is why Tgen allows the creation of any structured representation of the input string by extending the automatically generated parser into a translator.

To extend a parser into a translator, scanner directives must be specified with the token classes and the grammar.

```
<name>      : [a-zA-Z]+ ;
<space>     : [\s\t\r]+ {ignoreDelimiter} ;
```

In the above example, *ignoreDelimiter* is a scanner directive. It is a specification that when a <space> is encountered in the input text, that the scanner ignore it. The scanner directive *ignoreDelimiter* is actually a message that will be passed to the scanner whenever a token from that token class is recognized. If a class name is specified as a scanner directive instead of a method name, then the directive is to return an instance of that class instead.

Therefore in order to build a translator, a scanner directive must be specified for each rule in a grammar. And for each directive specified, a Smalltalk instance method must be implemented for the scanner. The method must have as many parameters as the rule has non-terminals. Every time the rule is applied when parsing, the Smalltalk method is invoked with the returned values of the non-terminals passed as parameters. This way the input string can be converted into a Smalltalk object.

Figure 24: String to graph translator

Figure 24 shows the Tgen specification for a string to graph translator. The grammar tells us that to make a graph the input string must be in the format of a CommandList. A CommandList is made up CreateNode and CreateEdge commands. The parse tree of the input in Figure 24 is shown in Figure 25.

Figure 25: Parse tree of example graph

Working backwards across the input the parser recognizes the `<name>` 'Y', preceded by the `<name>` 'X', preceded by the terminal 'edge'. The rule then applies the `CreateEdge` rule. It then sends the `#edgeFrom: 'X' to: 'Y'` message to the scanner, which creates an instance of the class `Edge`. The parser proceeds to match the second `CommandList` rule with a `CommandList` followed by a `CreateEdge`. The `CommandList` is again searched backwards to recognize the `<name>` 'Y' preceded by the terminal 'node'. The `CreateNode` rule is applied, and the `#createVertexLabeled: 'Y'` message is sent to the scanner. The scanner then builds a new instance of the class `Node`. The first `CommandList` rule is applied with a `CommandList` followed by a `CreateNode`. Still

working backwards another instance of class node is created and returned. The parser matches the first CommandList rule again with a CommandList and a CreateNode. The CommandList this time is the null string and is matched with the third CommandList rule. This results in the #createGraph message being sent to the scanner, which creates a new instance of the class Graph. At this point leaf tokens in the parse tree are resolved, and the parser begins to work backwards up the tree. The last application of the CommandList rule is resolved with the new graph and the 'X' node. This invokes the #toGraph:addVertex: message, which simply adds the 'X' node to the new graph. Continuing back up the tree, CommandList rule is resolved with the graph (containing the 'X' node) and the 'Y' node. The #toGraph:addVertex: method is invoked again, and the 'Y' node is added to the graph. Finally the 'XY' edge is added to the graph and the graph is returned.

This is exactly the method used to take a textual MSC and translate it into a trace grammar. In practice, debugging a translator specification is as challenging as debugging a computer program. Especially since Tgen only provides primitive debugging tools, and the MSC'96 grammar was found to have problems. But once everything was made to work, the translator built by Tgen, for translating MSCs in textual format to a grammar, was easily extracted from the class library and used independently of Tgen.

2.4.4 MSC Grammar

The following is a brief look at some of the important rules and what they do to translate a MSC grammar into the corresponding trace grammar. Each of the constructs

are described in full detail in Chapter 5. This section may be skipped if the reader does not require details of the MSC translator.

```
MSCDocument : DocumentHead MessageSequenceChartList
              {mscDocumentDocumentHead:mscList:} ;
MessageSequenceChartList :
    {OrderedCollection}
    | MessageSequenceChart MessageSequenceChartList
      {mscListMsc:mscList:} ;
```

The above rules say that a MSCDocument consists of a document head (which contains header information about the entire document), and a MSC list. A MSC list is a MSC followed by another MSC list. When the parser reaches the end of the MSC list, a null string is found and a new instance of OrderedCollection is returned. When a MSC is recognized, the #mscListMsc:mscList: method adds that MSC to the MSC list.

```
MessageSequenceChart : 'msc' MSCHead MSCBody 'endmsc' End
                      {mscMscHead:mscBody:end:} ;
```

This rule tells us that a MSC begins with 'msc' and ends with 'endmsc', and it consists of a head and a body. The head is a list of instances, while the body is a list of statements. The following rules work to build a statement list in the same manner as the MSC document builds a MSC list.

```
MSCBody : MSCStatementList
          {mscBodyMscStatementList:} ;
MSCStatementList :
    {OrderedCollection}
    | MSCStatement MSCStatementList
      {mscStatementListMscStatement:mscStatementList:} ;

MSCStatement : Name ':' InstanceEventList
              {mscStatementName:instanceEventList:}
```

Similarly, a MSC statement consists of the name of an instance responsible for sending or receiving messages followed by an instance event list. Again an instance event list consists of a list of instance events which could directly or indirectly be any of the following:

```

MessageOutput : 'out' MsgIdentification 'to' InputAddress
                {messageOutputMsgIdentification:inputAddress:} ;
MessageInput  : 'in' MsgIdentification 'from' OutputAddress
                {messageInputMsgIdentification:outputAddress:} ;
InlineExpr   : LoopExpr {answerArgument:}
                | OptExpr {answerArgument:}
                | AltExpr {answerArgument:}
                | ParExpr {parExpr:}
                | ExcExpr {doNothingWith:} ;
LoopExpr     : 'loopbegin' End MSCBody 'loopend'
                {loopExprEnd:mscBody:}
OptExpr      : 'optbegin' End MSCBody 'optend'
                {optExprEnd:mscBody:}
AltExpr      : 'altbegin' End MSCBody AltExprList 'altend'
                {altExprEnd:mscBody:altExprList:}
ParExpr      : 'parbegin' End MSCBody ParExprList 'parend'
                {parExprEnd:mscBody:parExprList:}
MSCReference : 'reference' MSCRefExpr ReferenceGateInterface
                {mscReferenceMSCRefExpr:referenceGateInterface:}

```

A message output begins with 'out', followed by a message identification (a message name), the terminal 'to', and an input address (a receiver). A message input begins with 'in', followed by a message identification, the terminal 'from' and an output address (a sender). The difficulty at this point is that the instance sending and receiving these messages is not known at this level. A solution to this is to put in a token of my own, and when the entire instance event list is assembled in the MSC statement rule, the token is replaced with the value returned by the Name non-terminal.

The four expression rules are also types of instance events. These expressions begin with a 'begin' token, end with an 'end' token and have another msc body within them. There are therefore cycles within this grammar that allow MSC statement lists to be defined within other MSC statement lists.

The loop expression returns a '(' symbol followed by a MSC body, followed by a ')*' if the MSC body can appear 0 or more times, or a '+)' if it must appear 1 or more times.

The optional expression returns a '(' symbol followed by a MSC body, followed by a ')?' symbol indicating that the MSC body may appear, but doesn't have to.

The alternate expression returns a '(' symbol followed by a MSC body, followed by any number of additional MSC bodies preceded by the 'l' symbol and finally ending with a ')' symbol. This means that one of the MSC bodies will appear because they are all separated by the 'l' ("or" operator).

The parallel operator returns a '(' symbol followed by every permutation of the MSC bodies surrounded in parenthesis and each permutation separated by 'l' operator. A final ')' symbol is appended to the end.

The reference construct returns the MSC reference expression which becomes a non-terminal in the trace grammar.

Each of these parts of a message sequence chart get broken down into their very basic components and systematically reassembled into one long string which results in a trace grammar.

The token classes, MSC grammar, and the specified methods used in the MSC translator can be studied in Appendix B. The actual source code, used by the MSC

translator, is a number of instance methods of the class MSCBuilder found on the disk labeled "DIVEt". For instructions on how to load and run the code for DIVEt, see Appendix A.

2.5 MESA (A MSC Editor)

Figure 26: MESA (Message sequence chart Editor, Simulator and Analyzer)

H. Ben-Abdallah and Stefan Leue have been developing a tool called MESA (Message sequence chart Editor, Simulator and Analyzer). It has been designed to

support a number of functionalities including the following, which have been extracted from [Ben-Abdallah 97]:

- Analysis of MSC specifications for “*things that can go wrong*”.
- *Synthesis* of code skeletons in full-fledged specification notations.
- Means to *simulate* the execution of MSC specifications.
- A GUI-based *editor* to manipulate MSC specifications.

It is the last function that is important for our purposes. MESA provides the MSC editor required to graphically build MSCs, and from that, automatically build the corresponding textual specification. It is for this reason, that I deemed it unnecessary to re-implement a tool of this sort.

2.6 Trace Tool

In order for DIVEt to be able to compare execution traces to the scenario model, there must be a way to extract an execution trace from a typical run of the system. I have developed a tool which does just this; it is called the trace tool.

The trace tool was implemented in VisualWorks Smalltalk, and has a very simple interface. The user simply gives a block of Smalltalk code to the trace tool, and asks for an execution trace of it. The trace tool then handles executing the code and keeping track of all messages that have been passed between objects. There is one parameter that allows the user to decide if she would like to trace intra-object messages or not, and another parameter which accepts a collection of trace classes. Trace classes are the classes of interest for the given block of code. When a class is added to the collection of trace classes, the trace tool knows to capture messages to and from the class, all its

subclasses, and all instances of these classes. A trace output from this tool is simply a string of messages separated by carriage returns, where each message follows the format shown in Figure 15. The example below shows an execution trace of aBank opening a savings account for Kathy Johnstone. The first line is the line of executed Smalltalk code, the remaining 5 lines are the execution trace.

```
aBank openAccount: SavingsAccount for: 'Kathy Johnstone'.

#openAccount:for:,env,bank1;
#new,bank1,SavingsAccount;
#initialize,SavingsAccount,savingsAccount2;
#newAccountNumber,savingsAccount2,BankAccount;
#owner:,bank1,savingsAccount2;
```

Looking at this execution trace, the first line tells us that the #openAccount:for: message was passed from the environment to bank1. The message #new is then passed from bank1 to the class SavingsAccount, which creates a new savings account. That savings account is then initialized by class SavingsAccount. A #newAccountNumber is then retrieved by savingsAccount2 from the class BankAccount. And finally bank1 assigns savingsAccount2 an #owner:. This is the same example used in section 5.1.

The trace tool is used in the test editor every time the user builds a trace from the test code, as described in section 4.6.5.

3. Testing Approaches

When building a software system, its testability is an important aspect to consider throughout the entire process. The easier a system is to test, the more effective a test strategy will be. Binder defines testability as:

“...a result of two factors: *controllability* and *observability*. A controllable implementation can be set to a desired state, have a desired set of inputs applied and be invoked in this state. An observable implementation does not obscure the occurrence of a fault or hinder determination of correctness.” [Binder 96]

It is therefore in the designer’s best interest to increase the testability of the design. A number of factors can render a design less testable, however, “inter-component dependency is seen as the single greatest obstacle to testability” [Binder 96]; where inter-component dependency is when one object “uses” another.

A common approach to testing begins with class or unit testing, where a unit can be a method or more often a class. The next step in testing is to confirm that these units work well together. This is called integration. Jorgenson et. al. list a number of approaches to integration testing which have been adapted to OO integration testing. [Jorgenson 94]

- the gradual replacement of stubs and drivers by separately tested units
- pairwise integration, in which each unit is integrated with its direct collaborators

- bottom-up integration guided by the part-of hierarchy¹⁷
- top-down integration guided by the functional decomposition tree
- “big bang” integration where all units are thrown together

The development and testing of units can be ordered by dependency. Murphy et. al. approached integration as follows

“The least dependent cluster was developed first and was tested as described earlier [unit tested]. Once tested, this least dependent cluster was used to supply a support environment for the testing of the next cluster.”
[Murphy 94]

Essentially Murphy et. al. use the bottom-up approach to integration described above, where a part-of hierarchy is a useful tool for illustrating some of the dependencies in a system.¹⁸ The least dependent units are leaves, and the most dependent units are at the top of the tree. An example of a part-of hierarchy can be seen in Figure 27.

¹⁷ A part-of hierarchy is a graphical view showing which objects are part of other objects.

¹⁸ A bottom-up approach to software testing is a common approach, however, a top-down approach can be used simultaneously to eliminate the possibility of late integration disasters.

Figure 27: Example part-of hierarchy for a car

This diagram shows that a car is dependent on its body, frame and engine. The body is dependent on the panels and the windows. The engine is dependent on the block and the rad, etc. When testing this car, the car cannot be tested until the engine works. The engine cannot be tested until the block and the rad both work, etc. This is what leads many testers to a bottom-up approach to testing. This is exactly the approach suggested by Jacobson [Jacobson 92].¹⁹

3.1 OOSE Integration Strategy

Jacobson's approach to testing is just as described in the previous paragraph. "The purpose of integration testing is to test whether different units that have been developed are working together properly." [Jacobson 92].

¹⁹ Whether the bottom-up or top-down approach is used, the proposed white-box cluster testing approach doesn't presuppose any level of abstraction. That is, this testing approach can be used to verify the interactions between objects or higher level components such as cluster or subsystems.

The next step is to determine how to build a test suite. Again Jacobson's use case driven approach relies heavily on use cases. "The use cases constitute an excellent tool for the integration test since they explicitly interconnect several classes and blocks. The basis of the specification emanates from the interaction diagrams." [Jacobson 92] Integration by use cases has also been suggested by [Firesmith 92]. Jacobson also notes that "we must be able to do regression testing without any great effort. ... The goal should therefore be to automate as much as possible of the testing" [Jacobson 92]. This is one of the goals of DIVEt. The more automated the testing strategy, the faster the testing can be done, and the more frequently tests can be run. This leads to better regression testing.

3.2 Automating Testing

Automated testing is an ideal way to maximize yield of testing. Tools that support such a feature provide the ability to reapply tests frequently and with little effort, when changes are constantly made to a system. Fowler says that "Test code, once written, should be kept forever. Set up your test code so that you can run every test with a simple command line or GUI button push. The code should respond either with "OK" or with a list of failures" [Fowler 97]. However, different tools support different features. An automated tool could excel in black-box testing and/or white-box testing. It could automate class testing and/or support integration testing. Whatever the type of testing applied, "automated testing leads to higher reliability by allowing more testing" [Binder 96]. One such tool developed by [Doong 94] presents a means of black-box testing.

3.3 ASTOOT

ASTOOT (A Set of Tools for Object-Oriented Testing) is a research system developed for testing programs in the Eiffel environment [Doong 94]. It automates class testing, which is needed because manual inspection of the state of complex objects can be slow and difficult. ASTOOT tries to automatically determine if a sequence of messages applied to an object leaves that object in a correct state. Methods can be applied in many different sequence, but whatever the order, the object under consideration must always be left in a correct state. The ASTOOT approach adapts ADT (Abstract Data Type²⁰) testing strategies to object-oriented software. The ASTOOT system automatically generates paired sequences of method operations which should yield the same observable results. Both sequences are executed, and the two resulting objects are compared. If the objects are not equivalent, then the test has successfully revealed a fault. For example, Figure 28 shows how tests 1 and 2 run sequences of methods on stacks A and B, which should yield the same result in all four sequences; 5 is on top.

Test	Test Sequence
1 Stack A	create, push(5), push(6), pop
Stack B	create, push(5)
2 Stack A	create, push(5), push(6), pop, top
Stack B	create, push(5), top

Figure 28: Equivalent sequence test cases [Doong 94]

In order for this approach to work each class must have an equivalence method developed, one that compares two instances of that class and determines if they are

²⁰ An abstract data type is a software module that encapsulates data and operations. They differ from objects in that they do not support inheritance, polymorphism among others.

equivalent objects. What drives this tool is a standalone driver that must initialize the object state, run a test sequence, then report the resulting state. ASTOOT demonstrates the ability to automate black-box testing. My thesis also provides a means of automated testing, but contrary to ASTOOT, it provides a means of automated white-box testing similar to the approach described in section 3.5.

3.4 FREE Approach to Testing OO Software

Robert Binder presents a tool for automated software testing in [Binder 96b]. The FREE (Flattened REGular Expression) approach provides unit, integration and system testing capabilities. It does so by flattening²¹ the class or cluster under test and modeling it as a finite state machine. Since finite state machines can be represented as regular expressions, an approach similar to the one used in DIVA is applied.

Binder's approach supports two types of testing; representation-based responsibility testing and implementation-based testing. The representation-based testing extracts message sequences from the class' specification, while implementation-based testing extracts message sequences from the implementation.

Binder claims that this approach is scaleable to cluster testing by assembling each class' finite state machine into a "system-wide state model". The testing techniques applicable to the class level are then also applicable to cluster, subsystem and system levels. Although this approach seems feasible in theory, as previously mentioned, it is probably not practical, because in large OO systems the number of states of the system is unmanageable, and because not all possible interactions are necessarily expected ones.

3.5 Method Sequence Specification

The closest testing approach to my proposal is an approach presented by [Kirani 94]. Kirani's approach uses regular expressions for representing the method sequence specification, similar to the method used in DIVA. The main difference being, these regular expressions are used to describe acceptable sequences of messages that a single object may encounter in a similar fashion described by [Adams 92]. "The method sequence specification of a class specifies the correct sequence usage for all the objects of the class." [Kirani 94]

Kirani et. al. say consistency and completeness are important factors in a good specification. "A specification is complete when all the required components are present and completely developed. A specification is consistent if its components do not conflict with each other." [Kirani 94] Kirani et. al. foresee three possible uses of method sequence specifications.

1. Ensuring conformance of the implementation of a class with the corresponding method sequence specification.
2. Constructing a runtime verification system for a class that ensures the correct method invocation sequence for each object of the class.
3. Generating state-based test cases for testing the low-level design and implementation of the class.

However, it is the first usage that is of most interest here, because it is the closest application to the one presented in this thesis. The major differences being that

²¹ Flattening is a means of viewing inherited features

- DIVA does not verify method sequences with respect to a single object, but rather method sequences between multiple collaborating objects,
- DIVA provides a graphical means of building the specification, which is a much more intuitive approach.

4. MSC Test Tool's Features And Limitations

This chapter presents a sort of user manual for DIVEt in the form of one detailed use case for the entire system. Initially the use case is summed up in point form, then each step is described in detail, illustrating what the user does, and how the system reacts.

4.1 Typical Usage

One brief use case illustrating the typical usage of DIVEt might read like the following:

- User automatically builds a MSC document using a MSC editor
- User inputs the MSC document into DIVEt
- User automatically builds a trace grammar that represents the MSC document
- User automatically builds a trace parser for the trace grammar
- User edits a test for testing one use case of the system
 - User creates a new test
 - User edits the description
 - User adds the Trace Classes
 - User edits the test code
 - User automatically builds an execution trace of the test
- User selects a test and parses its execution trace
- The trace parser either successfully or unsuccessfully parses the execution trace

I review each of these steps in the following sections.

4.2 User automatically builds a MSC document using a MSC editor

MESA as presented in section 2.5 provides the functionality required to graphically build MSCs and from them automatically build the MSC document required as input for DIVEt.

4.4 User automatically builds a trace grammar that represents the MSC document

Figure 30: MSC document menu

When the user right-clicks in the MSC document text editor, the menu shown in Figure 30 is displayed. The usual editing functions are available as well as the option to build a trace grammar from this MSC document. When a trace grammar is automatically built using this menu, the trace grammar text editor is automatically refreshed with this new grammar. Figure 31 shows the result of building a trace grammar.

Figure 31: Editing trace grammar

The trace grammar created by DIVEt is really a number of regular expressions. One regular expression represents one MSC, albeit restricted as explained later. The first regular expression is simply a starting point to allow any of the regular expressions to be applied. The reason this is called a grammar is because it is the specification for a language; the language of which any execution trace of our system must be a sentence.

At this point, I present another option for the user. The scenario modeling language of choice might not be MSCs, but instead regular expressions used in the same

manner as PEGs. If this is the case, the set of regular expressions can be assembled into a trace grammar and directly pasted into the trace grammar text editor (Chapter 5 gives more details about how a trace grammar is made).

4.5 User automatically builds a trace parser for the trace grammar

Figure 32: trace grammar menu

At this point, the user can right-click in the trace grammar text editor and show the menu in Figure 32. The user now has the option to edit the trace grammar, using any of the usual editing functions. However, this is unlikely, because there should be no necessary changes. This part of the tool could actually be hidden from the user, but it is useful as a debugging tool. Now the user should right-click in the trace grammar text editor and choose the option to build a trace parser for this language of MSCs.

4.6 User edits a test for testing one use case of the system

The user can create or edit a test by performing some or all of the following steps.

4.6.1 User creates a new test

Figure 33: Use case menu

By right-clicking within the test list box, the user can see the menu shown in Figure 33. By choosing the option to create a new test, the user is prompted for a test name. I have found that the title for the MSC under test makes a good name for the test. (This name may also be the same name given to a use case title in the requirements model). A test editor window opens, as in Figure 35, with the template for a new test. The test editor window consists of an name label at the top which identifies the test being edited. Below the name label on the left is the description text editor. To the right of that is the trace classes list box. At the bottom left is the test code text editor. And at the bottom right is the trace text editor. Each test consists of a name, description, list of trace classes, test code and execution trace. Each is defined in the following sections.

4.6.2 User edits the description

In the description text editor, the tool provides the space for a natural language description of what the test is testing. I have found that this is a good place to put the use case under test from the analysis phase.

4.6.3 User adds the Trace Classes

Figure 34: Trace classes menu

The user is now editing a new test that will test one use case of the system against the trace grammar. By right-clicking in the trace classes list box, the user has the option to add a trace class. A trace class is a class of objects that the user is interested in. That is, any class that is listed here will be used as the basis for tracing the execution of the test. Any message sent to any of these classes or subclasses, or any instances of these classes or subclasses will be recorded in the execution trace.

4.6.4 User edits the test code

Figure 35: Editing test code

The user may now actually write the test code in the test code text editor, which will test a specific MSC. Since this tool was developed in Smalltalk, the system being tested must be a Smalltalk system and the tests are therefore also written in Smalltalk code. Temporary variables can be declared as in any block of Smalltalk code. DIVEt also provides a means to set up a test before the trace begins. This is done by the messages `#traceOn` and `#traceOff`. These messages are sent to self, which is the test, and are used to start and stop recording object interactions, respectively. For example, in Figure 35, the temporary variable *aBank* is declared by putting it between `|`'s. The class *BankAccount* is initialized. The variable *aBank* is assigned a new instance of class *Bank* and given the name 'Piggy Bank'. And the `#traceOn` message is passed to self, indicating that from this point on, the test is to record object interactions in an execution trace. More specifically, only record the object interactions invoked by sending the message *#openAccount: SavingsAccount for: 'Kathy Johnstone'*.

4.6.5 User automatically builds an execution trace of the test

find...
replace...
undo
copy
cut
paste
do it
build execution trace
cancel

Figure 36: Test code menu

The user can debug the test, by right-clicking within the test code text editor, the option is available to “do it”. This is a common term in Smalltalk programming to execute the *selected* piece of code. If any syntax errors exist in the test code, errors will be indicated by the Smalltalk compiler, and the user will then have the option to fix them. After the test is bug free, a trace can be automatically built from its execution by selecting “build execution trace” from the menu.

Figure 37: Editing trace

The execution trace is displayed in the execution trace text editor, and is available for editing. One might want to edit an execution trace to perhaps change the names of certain objects so they conform to the names of the corresponding objects in the trace grammar. Notice that by default the instances are named by their class name followed by

a number. The numbers are simply assigned as the instances are introduced, beginning at one.

4.7 User selects a test and parses its execution trace

Figure 38: Use case menu

After repeating the process of creating and editing a test for each MSC in the scenario model, the user must return to the main DIVEt window. From this window, a test may be selected from the test list box. By right-clicking within the test list box the menu from Figure 38 is displayed and the selected test can be compared against the trace grammar. What is actually happening is the trace of the selected test is being parsed by the trace parser built from the trace grammar.

At this point if the parse is successful, then the execution trace follows one or more of the rules in the trace grammar. This is the desired result. If the parse is unsuccessful, then one of two things have happened.

1. The implementation has changed and the scenario model has not been updated to reflect it, or
2. The system implementation did not accurately follow the scenario model, and both should be reviewed.

Below is an example of a successful parse for the Open Savings Account example described in section 5.1. The first 6 lines are the text that is being parsed, while the remainder of the output is a list of the rules applied to achieve a proper parse.

```
LL Parser trace of:
#openAccount:for:,env,bank1;
#new,bank1,SavingsAccount;
#initialize,SavingsAccount,savingsAccount2;
#newAccountNumber,savingsAccount2,BankAccount;
#owner:,bank1,savingsAccount2;

apply production Start      : OpenSavingsAccount
apply production OpenSavingsAccount      :
'#openAccount:for:,env,bank' <number> ';' '#new,bank'
<number> ',SavingsAccount;'
'#initialize,SavingsAccount,savingsAccount' <number> ';'
'#newAccountNumber,savingsAccount' <number> ',BankAccount;'
'#owner:,bank' <number> ',savingsAccount' <number> ';'
OpenSavingsAccount00
cancel '#openAccount:for:,env,bank' from input
cancel '<number>' from input
cancel ';' from input
cancel '#new,bank' from input
cancel '<number>' from input
cancel ',SavingsAccount;' from input
cancel '#initialize,SavingsAccount,savingsAccount' from
input
cancel '<number>' from input
cancel ';' from input
cancel '#newAccountNumber,savingsAccount' from input
cancel '<number>' from input
cancel ',BankAccount;' from input
cancel '#owner:,bank' from input
cancel '<number>' from input
cancel ',savingsAccount' from input
cancel '<number>' from input
cancel ';' from input
apply production OpenSavingsAccount00      :
OpenSavingsAccount0
apply production OpenSavingsAccount0      :
```

The parse of the test input was successful.

Because of DIVEt relies on Tgen for its parsing capabilities, I have reused one of the output formats that it provides. A parse trace is not a very user-friendly output to

have to interpret, however it is the only output Tgen provides that indicates what the problem is with the parse if it has failed. If the bottom line says that the parse was successful then the execution trace of the selected test follows one or more of the rules in the trace grammar and there's really no need to look at the trace.

To demonstrate what an unsuccessful parse would look like, suppose the code was changed such that a bank no longer understands the message #owner:, but understands the message #customer: instead, which does exactly the same thing. The following text is the text being parsed.

```
#openAccount:for:,env,bank1;
#new,bank1,SavingsAccount;
#initialize,SavingsAccount,savingsAccount2;
#newAccountNumber,savingsAccount2,BankAccount;
#customer:,bank1,savingsAccount2;
```

The following is the same text as above with an error message inserted into it. The parser halted after it exhausted all the rules in the grammar and indicated an error in the execution trace like the following:

```
#openAccount:for:,env,bank1;
#new,bank1,SavingsAccount;
#initialize,SavingsAccount,savingsAccount2;
#newAccountNumber,savingsAccount2,BankAccount;
#customerSCANNER ERROR: illegal character encountered:
'$: "16r003A"' ->:,bank1,savingsAccount2;
```

The error being conveyed is that the scanner accepted the characters c-u-s-t-o-m-e-r, but it didn't understand the character ':'. This is because there is a message that a bank understands called #customers, which returns all the customers of that bank, however when trying to match this message with #customer:, the scanner turned up an error at the colon.

4.8 Limitations

DIVeT provides a simple, automated means of white-box testing clusters of classes in an object-oriented system. It can handle the most common constructs of MSC'96 and provide thorough testing results of any system, however, DIVeT has its limitations. This section attempts to address these limitations with a complete explanation for each one.

4.8.1 Restructuring MSC'96

First, MSC'96 is a telephony documentation standard that has been informally adapted to object-oriented scenario modeling, therefore, not all MSC'96 constructs are applicable to object-oriented software development. For instance, shared statements, such as a shared condition, do not apply because object-oriented languages typically do not allow one message to be passed to more than one object. That is, more than one object cannot share the same condition statement as they do in MSCs. For example in Figure 10 there is a MSC with a condition at the top requiring that both the `calling_party` and the `called_party` be idle. This is not a difficult condition to satisfy, however it is not possible to query both instances simultaneously.

Another construct not necessary in OO systems is the gate construct, because it does not map to anything in the OO world.

Some other features of MSC'96 are not considered because they are language specific. For example, timer statements, exceptions and conditions are all constructs defined by the language of implementation. For example, there are many different methods in Smalltalk that are considered conditions. The method `#isEmpty`, for instance,

can be passed to many kinds of objects like collections to see if they are empty. Another message that can be passed to any object is `#isNil`. A parser could not possibly realize that the `#isNil` message is a condition, unless every condition message is known by the trace parser. This is not possible since the set of these messages is unlimited (because it is user defined) and always changing. Therefore, it is better to leave this sort of construct out of the MSCs DIVEt accepts. The same reasoning applies for the timer statements and exceptions. Since each of these types of statements are very different for each programming language, leaving such constructs out allows DIVA to be applicable to any OO programming language.

4.8.2 Naming Instances

Often there is the case that a MSC requires two instances of the same class be kept distinct. DIVEt handles this in an execution trace by giving each instance the name of its class, beginning with a lower case letter, with the instance number appended to it. These numbers are used to uniquely identify each instance in the event that there are two instances of the same class. This poses two problems with naming instances in DIVEt.

- 1) **Problem:** Does each instance number in the MSCs have to match the number given to an instance in the execution trace?

Solution: DIVEt takes care of this problem by taking into account the fact that it is possible that an instance name in the execution trace has a number appended to it. For example, if the next message expected by the MSC is a `#newAccountNumber` message sent from any instance of `SavingsAccount` to the class `BankAccount`, then the trace parser will accept any message of this form:

`'#newAccountNumber,savingsAccount' <number> ',BankAccount;'`²²

Therefore, if the message `#newAccountNumber,savingsAccount1,BankAccount;` is encountered, it will be parsed properly. This means that any number can follow the instance name. Instance numbers are never appended to classes, because classes in Smalltalk are always unique.

2) **Problem:** What if the name of an instance in the MSC is completely different from the name generated in the execution trace.

Solution: If for instance, the name of an instance of class `Person` is “janitor” in the MSC, but the execution trace generates the name `person3`, then the use case editor provides a find and replace function that could be used to replace all occurrences of `person3` to `janitor`. Currently, DIVEt does require user intervention at this step to ensure that a correct parse is done.

4.8.3 Other Syntactic Limitations

The MSC grammar presented in [MSC 96] presented a few problems. For example, within the grammar, the tokens ‘loop’ and ‘begin’ appear with a space between them. This introduces an ambiguity²³ with other rules that have just the token ‘loop’ in them. For the purposes of this thesis, the space was left out between these tokens as well as any occurrence of ‘loop’ and ‘end’ and the new tokens ‘loopbegin’ and ‘loopend’ were introduced. The same was done for the tokens ‘opt’, ‘alt’ and ‘par’. The exact changes can be found in Appendix B.

²² <number> is defined as any sequence of digits between 0 and 9.

A second ambiguity was found in the MSC grammar. This ambiguity was a result of having no distinction between `<name>s`²⁴. The token `<name>` is used in many rules of the MSC grammar. It is used to match “almost” any string of characters. It is used to match the name of a MSC document, a MSC, an instance, a reference, etc. However, some of the rules in the MSC grammar are so similar that they produce ambiguities. The only way I could remove these ambiguities was by differentiating between certain names. That is, I created a new type of name (called `<ambiguous_name>` for lack of a better name) which begins with an underscore character (i.e. `'_'`). I also ensured that regular `<name>s` cannot begin with an underscore character. This way, it is obvious which type of name is being looked at, and the proper rule can be applied. This constitutes one of several minor changes I had to make to MSC'96 presented in [MSC 96]. The specific changes can be found in Appendix B.

The MSC grammar also didn't account for the possibility of having a `':'` in a message name, which any Smalltalk method needing parameters requires. To be consistent with MSC'96, my tool automatically replaces every colon in every message with a period, which is an acceptable character in the MSC grammar. For example the message `#at:put:` would be converted to `#at.put`.

These are all of the technical or syntactical limitations with DIVEt. The following paragraphs present some limitations of a more functional nature. That is, the limitations that aren't specific to my implementation of DIVEt, but are specific to the general approach used in DIVA.

²³ An ambiguity in a grammar means that a parser for this grammar cannot deterministically decide which rule of the grammar to apply next.

4.8.4 Other Limitations

This first point could be called a limitation, however, I call it a design decision. The tool used to automatically build an execution trace from a block of Smalltalk code was designed to be able to capture intra-object messages and inter-object messages²⁵. However, it is my decision that tracing every intra-object message would make for a very clumsy testing tool. By capturing all intra-object messages in the execution trace, the user wrongly gets the impression that the scenario model must include all intra-object messages, otherwise the trace parser will fail when one is encountered. Robert Binder said, “It is very important to hide implementations with this kind of work [scenario modeling]. Don’t document every interaction; instead, focus on the key ones” [Binder 96]. Based on this quote, it would be better to leave it up to the designer to decide whether or not to include intra-object messages in the scenario model, instead of insisting on their presence. It is for this reason, that when an intra-object message is encountered in a MSC, that message is automatically made optional. This way, the message is included in the trace grammar, but since the trace tool doesn’t capture these messages, they will never be matched by the parser.

A second limitation with using regular expressions to represent MSCs is that the loop construct cannot be mapped exactly to a regular expression construct. For example, suppose the design of some baseball program has a loop to iterate through the nine innings, indicating that each game must have at least nine innings, perhaps more if the

²⁴ <name> is a token class defined in Appendix B

²⁵ Messages sent between different objects.

score is tied. The only way to map this loop to a regular expression used in a rule of the trace grammar, is to use the '*' operator as in the following:

```
(#top,game,inning; #bottom,game,inning;)*
```

This accepts any game with 9 innings, even 10 or more if the game has extra innings, however it also erroneously accepts a game with 4 innings as an acceptable baseball game. This would result in an erroneous agreement with the scenario model, when in actual fact it should not. Currently Tgen does not support more sophisticated formal tools that could be used to address this problem.

Another limitation introduced because of the use of regular expressions, is the difficulty of mapping parallel constructs of MSC'96 to a regular expression. The only way to represent parallel regions with a regular expression is to provide all possible combinations of execution sequences and allow an execution to take any of these paths.

The problem is somewhat simplified because the language of implementation is Smalltalk, which does not support true concurrency. Messages within different processes are typically not interleaved in VisualWorks. Processes typically run to completion, which means that parallel regions can be dealt with as sequential blocks of messages, the only difficulty is dealing with the order that the parallel regions are executed.

For instance, if the parallel regions A, B and C can be executed in any order, then the regular expression representing this is shown below and allows all sequences of these three regions separated by the 'I' operator.

```
(A B C | A C B | B A C | B C A | C A B | C B A)
```

This is the only correct solution, using only the operators that Tgen provides. However this makes a grammar long and unreadable.

Ideally we would want Tgen to support a '&' operator as [Adams 92] describes in Figure 13. If this were the case, the optimal solution could be obtained. The following regular expression would be the optimal solution and could be correctly represented and efficiently parsed.

(A & B & C)

However since this operator is not available in Tgen, the first solution is used as seen in section 5.7. The same approach is used for concurrent regions in section 5.5, where A, B and C are messages sent to an object and the order is irrelevant. The implementation of the '&' operator in Tgen is one of the most important improvements I see for DIVEt to be applicable to large scale problems.

Each of these limitations has been dealt with in DIVEt, and are all potential areas for future research as described in section 6.1.

5. Examples

In this chapter each construct of MSC'96 that is dealt with by DIVEt is described using an example. A bank system is used throughout, and following Jacobson's suggestions on testing, each example involves testing one use case. The use cases chosen to describe the typical uses of this bank are not necessarily well designed use cases, because the goal is to cover all the functionality of MSC'96 handled by DIVEt, not to design a good bank system. The use cases are therefore incomplete. The constructs of MSC'96 are presented in order of complexity, where the simplest and most common examples are first, and the more complex and least common are last. The first five use cases can be found on the accompanying disk in the file *BankVer1.msc*. The sixth and seventh examples are different implementations of the same use case found in example four. They can also be found on the accompanying disk in the files *BankVer2.msc* and *BankVer3.msc* respectively.

For presentations purposes each MSC document has been divided into separate MSCs. The three MSC documents used in the following examples, are found in the three *BankVerX.msc* files, can be seen in their entirety in Appendices C through E.

In the following examples, only two trace classes were specified; Bank and BankAccount. Since trace classes automatically define all subclasses as trace classes, the classes of concern are Bank, BankAccount, CheckingAccount, SavingsAccount and TermDeposit.

5.1 Message Passing Example

This example demonstrates the capability of DIVEt to convert a simple MSC to the rule of the trace grammar. It is important to see how each message is converted into part of the regular expression listed below.

Graphical Message Sequence Chart

Figure 39: Open savings account MSC

Textual Message Sequence Chart

```

msc OpenSavingsAccount;
inst bank, BankAccount, savingsAccount, SavingsAccount;
bank:
  in openAccount.for. from env;
  out new to SavingsAccount;
SavingsAccount:
  in new from bank;
  out initialize to savingsAccount;

```

```

savingsAccount:
    in initialize from SavingsAccount;
    out newAccountNumber to BankAccount;
BankAccount:
    in newAccountNumber from savingsAccount;
bank:
    out owner. to savingsAccount;
savingsAccount:
    in owner. from bank;
bank:
    out addAccount. to bank;
    in addAccount. from bank;
endmsc;

```

Notice that each message is really recorded twice in the textual MSC; once under the sender object and once under the receiver object. However, the doubles are eliminated by the MSC translator when translated to a rule of the trace grammar.

Also notice in the fourth line of the textual message sequence chart that the message #openAccount:for: is written openAccount.for. with periods instead of semi-colons. As mentioned in section 4.8, this is because the MSC'96 standard doesn't accept colons within message names. This is dealt with by converting (i.e. replacing all colons with periods for) each method name in the execution trace before it is parsed.

Regular Expression

```

OpenSavingsAccount :
(' #openAccount:for:,env,bank' <number> ';'
'#new,bank' <number> ',SavingsAccount;'
'#initialize,SavingsAccount,savingsAccount' <number> ';'
'#newAccountNumber,savingsAccount' <number> ',BankAccount;'
'#owner:,bank' <number> ',savingsAccount' <number> ';'
('#addAccount:,bank' <number> ',bank' <number> ';' )? );

```

The regular expression above, is actually a rule in the trace grammar. As described in section 2.4, a rule consists of a non-terminal on the left of the colon on the

first line, and a sequence of terminals and non-terminals on the right of the same colon. The left part of this rule is the non-terminal `OpenSavingsAccount`. It is used as a means of referring to the right side. The right side is the sequence of terminals and non-terminals arranged in a regular expression. The terminals are the strings enclosed in single quotes, while the non-terminals are the words enclosed by the characters `<` and `>`.

Notice that the non-terminal `<number>` appears many times in this regular expression. This is a result of insisting that instances be uniquely named in the execution trace. By including the `<number>` token after instances of `bank` and `savingsAccount`, it allows the trace parser the ability to parse a trace with any bank or savings account, no matter what instance number was appended to the end of it. Refer to section 4.8 for more details.

The graphical MSC, textual MSC and regular expression in this example are all describing basically the same information. They all express that when opening a savings account, a new account is created, initialized, given an account number, and assigned an owner. The only part of the regular expression that is not consistent with the MSC, is that the `#addAccount:` message need not be invoked. This is because intra-object messages are not captured in the execution trace as explained in section 4.8.

5.2 Optional Construct

The optional construct provides a means of indicating that a specific path of execution may or may not be invoked. This type of construct can be exactly represented using the `'?`' operator described in Figure 16. All of the messages within this construct are put inside braces to treat them as a single optional unit. In this case only the message

#name: sent from the environment to bank is optional. This is different from the optional message shown in the regular expression of section 5.1

Graphical Message Sequence Chart

Figure 40: Open bank MSC

Textual Message Sequence Chart

```

msc OpenBank;
inst bank, Bank;
Bank:
    in new from env;
    out initialize to bank;
bank:
    in initialize from Bank;
all: optbegin;
    bank:
        in name. from env;
    optend;
endmsc;
  
```

Regular Expression

```
OpenBank :
('#new,env,Bank;' '#initialize,Bank,bank' <number> ';'
('#name:,env,bank' <number> ';' )? );
```

This time, the graphical MSC, textual MSC and the regular expression all represent exactly the same set of acceptable message sequences. It describes that in order to open a bank, first a new bank must be created and initialized, then it can be given a name, however giving a bank a name is not necessary.

5.3 Loop Construct

The loop construct provides a means of repeated sequences of messages. Again the entire sequence of messages is treated as a single unit by enclosing it in braces. Then the '*' or '+' symbol, as described in Figure 16, follows to indicate that the enclosed sequence of messages may be invoked zero or more times, or one or more time, respectively. MSC'96 allows a loop to indicate the minimum and maximum number of iterations, but regular expressions do not accommodate this feature. Therefore, only the first parameter is taken into account. If the minimum iterations parameter is zero, then the '*' operator is used, otherwise the '+' operator is used.

To simplify this example, let's assume all bank accounts are checking accounts.

Graphical Message Sequence Chart

Figure 41: Checking account customers MSC

Textual Message Sequence Chart

```

msc CheckingAccountCustomers;
inst bank, checkingAccount;
bank:
    in customers from env;
    out accounts to bank;
    in accounts from bank;
all: loopbegin;
    bank:
        out owner to checkingAccount;
    checkingAccount:
        in owner from bank;
    loopend;
endmsc;
  
```

Notice the “loopbegin” and the “loopend”: tokens on the 7th and 10th lines respectively, of the textual MSC have no space after “loop”. As discussed in section 4.8, eliminating this space removes an ambiguity in the grammar presented in [MSC 96].

Regular Expression

```
CheckingAccountCustomers :
('#customers,env,bank' <number> ';'
('#accounts,bank' <number> ',bank' <number> ';' )?
('#owner,env,checkingAccount' <number> ';' )* );
```

This example shows the object interactions involved in retrieving all customers with accounts from a bank. First, a list of the accounts is retrieved from the bank, followed by a loop that iterates through these accounts and retrieves the owner from each one. The following example doesn't assume that all accounts are checking accounts, this requires a new construct that deals with alternate paths of execution.

5.4 Alternate Construct

Alternate paths of execution can be indicated by enclosing them in an alternate construct. This construct also maps exactly to a regular expression. Simply separating each alternate sequence by a 'l' operator, and enclosing the entire expression in parentheses provides the same meaning in a regular expression.


```

all: altbegin;
    checkingAccount:
        in payMonthlyInterest from env;
    alt;
    savingsAccount:
        in payMonthlyInterest from env;
        out interestRate to BankAccount;
    BankAccount:
        in interestRate from savingsAccount;
    alt;
    termDeposit:
        in payMonthlyInterest from env;
        out interestRate to BankAccount;
    BankAccount:
        in interestRate from termDeposit;
    altend;
loopend;
endmsc;

```

Regular Expression

```

PayMonthlyInterest :
( '#payMonthlyInterest, env, bank1; '
( '#accounts, bank1, bank1; ' ) ?
( ( '#payMonthlyInterest, env, checkingAccount' <number> ';' )
| ( '#payMonthlyInterest, env, savingsAccount' <number> ';'
'#interestRate, savingsAccount' <number> ', BankAccount;' )
| ( '#payMonthlyInterest, env, termDeposit' <number> ';'
'#interestRate, termDeposit' <number> ', BankAccount;' ) ) * );

```

This example is a little more complex than any of the previous ones. Basically it shows us that when a bank pays its monthly interest to its accounts, first it retrieves its accounts, then it loops through each account and sends each the #payMonthlyInterest message. Depending on what type of bank account it is, it will respond differently. A checking account doesn't pay interest, so it will do nothing. A savings account will retrieve the interest rate from the class BankAccount, and pay that percentage in interest. A term deposit will also retrieve the interest rate and base its calculation on that.

5.5 Concurrent Messages

The concurrent construct is not as easy to map to a regular expression as the first constructs. Any message events within a concurrent construct may occur in any order. The only way to represent this with a regular expression is to provide all possible combinations of execution sequences and allow an execution to take any of these paths.

Graphical Message Sequence Chart

Figure 43: Multiple requests from first account MSC

Textual Message Sequence Chart

```

msc MultipleRequestsFromFirstAccount;
inst bank, savingsAccount;
bank:
  in multipleRequestsFromFirstAccount from env;
  out owner to savingsAccount;
  out balance to savingsAccount;
  out accountNumber to savingsAccount;
savingsAccount:
  concurrent;
    in owner from bank;
    in balance from bank;
    in accountNumber from bank;
  
```

```

        endconcurrent;
    endmsc;

```

Regular Expression

```

MultipleRequestsFromFirstAccount :
('#multipleRequestsFromFirstAccount,env,bank' <number> ';'
('#owner,bank' <number> ',savingsAccount' <number> ';'
'#balance,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number> ';')
| ('#owner,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number> ';'
'#balance,bank' <number> ',savingsAccount' <number> ';'
| ('#balance,bank' <number> ',savingsAccount' <number> ';'
'#owner,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number> ';')
| ('#balance,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number> ';'
'#owner,bank' <number> ',savingsAccount' <number> ';')
| ('#accountNumber,bank' <number> ',savingsAccount' <number> ';'
'#owner,bank' <number> ',savingsAccount' <number> ';'
'#balance,bank' <number> ',savingsAccount' <number> ';')
| ('#accountNumber,bank' <number> ',savingsAccount' <number> ';'
'#balance,bank' <number> ',savingsAccount' <number> ';'
'#owner,bank' <number> ',savingsAccount' <number> ';') );

```

This example demonstrates the situation of some external object generating multiple requests at a bank. The bank passes these requests onto an account, where the account doesn't care in what order they arrive. The rule above shows that when the message `#multipleRequestsFromFirstAccount` is invoked, that it must be followed by the messages `#owner`, `#balance` and `#accountNumber` sent from the bank instance to the `savingsAccount` instance, and that they may occur in any order. Notice that the three messages are listed above in the six possible different orders, as described in section 4.8.

5.6 Message Sequence Chart Reference

A reference in a MSC provides a means to include one MSC within another, by referring to it by a name. A reference maps exactly to the notion of a non-terminal in a regular expression. Wherever MSC A references MSC B, MSC A simply includes B in the regular expression as a non-terminal. In the following example, each of the accounts has its own MSC to pay monthly interest. Therefore each of these MSCs are converted to rules and the non-terminals are used by the MSC titled PayMonthlyInterest.

Graphical Message Sequence Chart

Figure 44: Reference pay monthly interest MSC

Figure 45: Checking account pay monthly Interest MSC

Figure 46: Savings account pay monthly interest MSC

Figure 47: Term deposit pay monthly interest MSC

Textual Message Sequence Chart

```

msc PayMonthlyInterest;
inst bank1, BankAccount, checkingAccount, savingsAccount,
termDeposit;
bank1:
    in payMonthlyInterest from env;
    out accounts to bank1;
    in accounts from bank1;
all: loopbegin;
    all: altbegin;
        all: reference
CheckingAccountPayMonthlyInterest;
        alt;
            all: reference
SavingsAccountPayMonthlyInterest;
            alt;
                all: reference TermDepositPayMonthlyInterest;
                altend;
            loopend;
    altend;
loopend;
endmsc;

msc CheckingAccountPayMonthlyInterest;
inst checkingAccount;
checkingAccount:
    in payMonthlyInterest from env;
endmsc;

msc SavingsAccountPayMonthlyInterest;
inst savingsAccount, BankAccount;
savingsAccount:
    in payMonthlyInterest from env;
  
```

```

        out interestRate to BankAccount;
BankAccount:
    in interestRate from savingsAccount;
endmsc;

msc TermDepositPayMonthlyInterest;
inst termDeposit, BankAccount;
termDeposit:
    in payMonthlyInterest from env;
    out interestRate to BankAccount;
BankAccount:
    in interestRate from termDeposit;
endmsc;

```

Regular Expression

```

PayMonthlyInterest : ('#payMonthlyInterest,env,bank1;'
('#accounts,bank1,bank1;')?
((CheckingAccountPayMonthlyInterest)
 | (SavingsAccountPayMonthlyInterest)
 | (TermDepositPayMonthlyInterest))* );

CheckingAccountPayMonthlyInterest :
('#payMonthlyInterest,env,checkingAccount' <number> ';' );

SavingsAccountPayMonthlyInterest :
('#payMonthlyInterest,env,savingsAccount' <number> ';'
'#interestRate,savingsAccount' <number> ',BankAccount;' );

TermDepositPayMonthlyInterest :
('#payMonthlyInterest,env,termDeposit' <number> ';'
'#interestRate,termDeposit' <number> ',BankAccount;' );

```

This set of four MSCs is designed to make the PayMonthlyInterest MSC smaller, and a little more readable than the one shown in section 5.4. In the bottom three rules above, each one begins with a non-terminal: CheckingAccountPayMonthlyInterest, SavingsAccountPayMonthlyInterest, and TermDepositPayMonthlyInterest, respectively. The first rule gets the accounts, loops through them and refers to the proper non-terminal based on what type of account it is.

5.7 Parallel Messages

Smalltalk does not actually support true concurrent processes, because parallel regions do not actually run in parallel. Recall that a process in Smalltalk either finishes to give up control, or voluntarily gives up control to another process. Therefore, we need not concern ourselves with parallel sections interleaving their messages. What is of concern, is the order of execution of each section. This is the very same problem posed by the concurrent region construct, except concurrency is at a level of processes not messages. Again, if Tgen supported the '&' operator as described in section 4.8, a parallel region could be mapped well to a regular expression, but instead we must use the same approach of allowing for each possible ordering of the parallel regions.

Graphical Message Sequence Chart

Figure 48: Parallel pay monthly interest

Textual Message Sequence Chart

```

msc PayMonthlyInterest;
inst bank1, BankAccount, checkingAccount, savingsAccount,
termDeposit;
bank1:
    in payMonthlyInterest from env;
    out accounts to bank1;
    in accounts from bank1;
all: parbegin;
    all: reference CheckingAccountPayMonthlyInterest;
par;
    all: reference SavingsAccountPayMonthlyInterest;
par;
    all: reference TermDepositPayMonthlyInterest;
parend;
endmsc;

```

The referenced MSCs are the same as the previous example.

Regular Expression

```

PayMonthlyInterest : ('#payMonthlyInterest,env,bank1;'
('#accounts,bank1,bank1;')?
((CheckingAccountPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
| (CheckingAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
| (SavingsAccountPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
| (SavingsAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
| (TermDepositPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
| (TermDepositPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)) );

CheckingAccountPayMonthlyInterest :
('#payMonthlyInterest,env,checkingAccount' <number> ';' );

SavingsAccountPayMonthlyInterest :
('#payMonthlyInterest,env,savingsAccount' <number> ';'
'#interestRate,savingsAccount' <number> ',BankAccount;' );

TermDepositPayMonthlyInterest :
('#payMonthlyInterest,env,termDeposit' <number> ';'
'#interestRate,termDeposit' <number> ',BankAccount;' );

```

This example is not an intuitive one, in fact, it is not appropriate for a bank system, but it will do to get the point across. This example retrieves the accounts from the bank, and in groups of three (one savings account, one checking account and one term deposit), it pays the monthly interest on them in parallel. As you can see, this example

has no value within a banking system, but note that, as in the concurrent region example, the first rule above allows the three monthly interest payments to occur in any order.

6. Conclusions

This chapter summarizes DIVEt's approach to software testing (DIVA) and its contribution to the object-oriented software testing community.

A means of white-box cluster testing has been proposed; testing interaction traces against interaction specifications. DIVEt provides a means of verifying the implementation against the scenario model to ensure that what was implemented corresponds to what was designed. This approach is used to verify that the objects within the cluster under test collaborate in the manner that they were intended. That is, the objects send the messages expected as illustrated by the MSCs. The proposed approach facilitates integration. This type of testing falls into a category called white-box cluster testing because many objects are being integrated together, and the tester has knowledge of the internal structure of the program.

DIVEt was designed with many useful features. It was designed to fit well into Jacobson's OOSE methodology [Jacobson 92], in that DIVEt is heavily use case driven. It also automates some of the testing process just as ASTOOT did [Doong 94], which relieves testers of much manual work. Test automation provides more reliable results and speeds up the process to allow more effective testing to be accomplished. DIVEt offers a different approach to cluster testing than Binder's FREE approach. Since the synthesis of individual object's finite state machines may not scale up well for large OO projects, DIVA offers a nice alternative. The approach taken uses a technique similar to the approach described by [Kirani 94] which uses method sequence specification; the big

difference being that Kirani's approach doesn't deal with inter-object interactions, but rather intra-object messages.

6.1 Future Research

DIVeT provides a clean method of white-box cluster verification, however it has limitations that would keep it from becoming an industrial strength testing tool. For example, the process of creating a trace parser from a set of MSCs could be fully automated to save time. Ideally, textual MSCs would be automatically translated into an equivalent regular expression format, which can be used to automatically build a trace parser. The reason this is not currently done is that the intermediate steps provide substantial information when testing the tool. If these steps were fully automated, this would hide the trace grammar from the tester's view, which would relieve her of requiring any knowledge of the trace grammar at all.

To make this tool more practical, there must be a way to create and view message sequence charts graphically. Ideally, the tool would be integrated with DIVeT and would allow the user to build MSCs, convert them to instance and event oriented textual MSC documents, and perhaps even display MSCs given either of the textual formats. MESA deals with these exact problems and provides a nice user-friendly interface for manipulating MSCs. High-level message sequence charts have not been dealt with by DIVeT, however since no new constructs are introduced at this level, it was not deemed necessary. Few additions to this tool would be required to be able to parse them as well.

One of the biggest fallbacks of DIVeT is that it is too strict. "If the interaction diagram and the implementation are in exact correspondence (this is unlikely), event trace

coverage is equivalent to thread coverage.” [Binder 96b] There is currently no means of expressing a level of importance of messages. That is, which messages must be invoked and which messages could be skipped without returning a failed parse. In other words, the trace parser insists on a strict message ordering, and perhaps its approach could be relaxed a little more. If instead of the MSC translator returning a yes or no answer, a little more intelligence could be built in to allow it to skip over some less important and incorrectly parsed messages and provide the user with a means of listing these skipped messages. This approach would require a more sophisticated tool than Tgen, and would allow a failed test case to indicate how badly it failed.

Perhaps the best time spent on improving DIVEt would be improving Tgen so it could handle the concurrent constructs defined in MSC’96. Adding the ‘&’ operator as defined in Figure 12 would provide better solutions to concurrent regions and parallel constructs. Creating a new operator to efficiently parse a sequence of messages in any order would greatly improve the performance of the trace parser. It could do so by reducing the size of the trace grammar and therefore the parse time.

A not so stringent parser could allow for a design pattern [Gamma 95] search tool. If a collection of design patterns were expressed with message sequence charts, a more intelligent parser could find these common design patterns either in an execution trace or the scenario model. Once design patterns are recognized, they can facilitate the illustration of a system.

Regular expressions also present a limitation as described in section 4.8.4. Regular expressions and finite state machines express the same information, and neither of them can express the state of an object. Extended finite state machines, however, can

express an object's state with variables. Perhaps a new notation could accommodate variables in regular expressions as well to better describe certain systems.

Finally, it would be useful to relieve the user of insisting that the instance names of the objects in an execution trace match those illustrated in the MSCs.

6.2 Concluding Remarks

This thesis contributes a MSC'96 parser created from Tgen. This parser has been extended to build a MSC translator which translates any MSC document into a nearly equivalent grammar. This translator can be modified to suit another projects needs such that it outputs some other representation of a MSC document. It also contributes a grammar representation of MSC'96, which is a formal representation of a scenario model that follows the more familiar framework of regular expressions. I have developed a tool for extracting the sequence of messages passed between objects for any supplied block of Smalltalk code. And finally, I developed an implementation-design verification process presented by [Kirani 94] by focusing on inter-object communication instead of intra-object communication, and applying it to cluster testing.

I have successfully applied an automated approach to integration testing, however, in its current state this tool is merely a proof-of-concept, and is not yet of industrial strength.

7. Bibliography

- [Adams 92] Adams, G., *Describing Groups of Interacting Objects Using Path Expressions*, Master's Thesis, University of Ottawa, Canada, December 1992.
- [Ben-Abdallah 97] Ben-Abdallah, H., and Leue, S., *MESA: Support for Scenario-Based Design of Concurrent Systems*, Technical Report 97-12, Dept. of Electrical and Computer Engineering, University of Waterloo, October 1997.
- [Binder 94] Binder, R., *Design for Testability with Object-Oriented Systems*, Communications of the ACM, Vol. 37, No. 9, pp. 87-101.
- [Binder 96] Binder, R., *Testing Object-Oriented Software: a Survey*, Software Testing, Verification and Reliability. Vol. 6, 1996, pp. 125-252.
- [Binder 96b] Binder, R., *The FREE Approach to Object-Oriented Testing: An Overview*, <http://www.rbsc.com/pages/FREE.html>.
- [Benvenuti 97] Benvenuti, Nicola, *MSC'96 Editor*, Master's Thesis, Carleton University, Ottawa, Canada, 1997.
- [Booch 97] Booch, G., Rumbaugh, J., Jacobson, I., *Unified Modeling Language User Guide*, Addison-Wesley, Available fall 1997.
- [Campbell 73] Campbell, R. and Habermann, A., *The Specification of Process Synchronization by Path Expression*, Lecture Notes in Computer Science, Vol.16, Springer Verlag, 1973, pp. 89-102.
- [Corriveau 96] Corriveau, J-P., *Traceability and Process for Large OO Projects*, Tools USA '96 Conference, Santa Barbara, U.S.A., August 1996.
- [Doong 94] Doong, R.-K. and Frankl, P. G., *The ASTOOT approach to testing object-oriented programs*, ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 4, pp 101-130.
- [Firesmith 92] Firesmith, D. G., *Testing object-oriented software*, Technical Report, Ossian, Ind. U.S.A., Advanced Technology Specialists.
- [Fischer 91] Fischer, C. N. and LeBlanc, R. J. Jr., *Crafting a Compiler with C*, Benjamin/Cummings, 1991.
- [Fowler 97] Fowler, M. with Scott, K., *UML Distilled*, Addison-Wesley, 1997.

- [Gamma 95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns*, Addison Wesley, 1995.
- [Graver 91] Graver, J. O., *T-gen User's Guide*, Technical Report SERC-TR-50-F, Software Engineering Research Center, April, 1991.
- [Hajla 97] Hajla, Halim Ben, *Traceability in Object-Oriented Quality Engineering*, Master's Thesis, University of Ottawa, Canada, October 1997.
- [Hopcroft 79] Hopcroft, J., *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.
- [Jacobson 92] Jacobson, I., et al., *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, 1992.
- [Jacobson 97] Jacobson, I., Booch, G., Rumbaugh, J., *The Objectory Software Development Process*, Addison-Wesley, Available winter 1997.
- [Jorgensen 94] Jorgensen, Paul C. and Erickson, Carl, *Object-Oriented Integration*, Communications of the A.C.M., September 1994, Vol. 37, No. 9.
- [Kirani 94] Kirani, S. and Tsai, W. T., *Method sequence specification and verification of classes*, Journal of Object-Oriented Programming, Vol. 7, No. 6, pp. 28-38.
- [Lalonde 90] Lalonde, Wilf and Pugh, John, *Inside Smalltalk*, Vol. 1, Prentice-Hall, 1990.
- [Lalonde 91] Lalonde, W., White, P., and McGuire, K., *Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects*, Technical Report, School of Computer Science, Carleton University, April 1991.
- [Lesk 75] Lesk, M.e., *LEX - A Lexical Analyzer Generator*, CSTR 39, Bell Laboratories, Murray Hill, NJ, 1975.
- [McGregor 94] McGregor, J.D. and Korson, T.D., *Integrated Object-Oriented Testing and Development Processes*, Communication of the ACM, Vol. 37, No. 9, September 1994.
- [MSC 92] Recommendation Z.120, *Message Sequence Chart (MSC)*, CCITT, 1992.
- [MSC 96] Recommendation Z.120, *Message Sequence Chart (MSC)*, ITU-T, 1996.
- [Myers 79] Myers, Glenford J., *The Art of Software Testing*, Wiley-Interscience, 1979.

- [Regnell 95] Regnell, B., et al., *Improving the Use Case Driven Approach to Requirements Engineering*, Proc. of the Second IEEE Int. Symp. on Requirements Engineering, York, March 1995.
- [Rumbaugh 91] Rumbaugh, J. et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Rumbaugh 94] Rumbaugh, J., *Getting Started Using Use Cases to Capture Requirements*, Journal of Object-Oriented Programming, Vol. 7, No. 5, September 1994.
- [Rumbaugh 97] Rumbaugh, J., Booch, G., Jacobson, I., *Unified Modeling Language Reference Manual*, Addison-Wesley, Available winter 1997.
- [Schlaer 92] Schlaer, S. and Mellor, S., *Object Lifecycles Modeling the World in States*, Yourdon Press, 1992.
- [Sommerville 95] Sommerville, I., *Software Engineering*, Addison-Wesley, fifth edition, 1995.

Appendix A: Loading And Running DIVEt

To run DIVEt, some basic knowledge of VisualWorks' programming environment is necessary. First insert the Tgen disk in drive A:. Then, with the file browser provided, locate the file *FileMeIn.st* on that disk, and file it in. A dialog box will ask you for the directory containing the source code of Tgen with the default being 'A:\', click OK. The code will be loaded into the image. Insert the DIVEt disk in drive A:, and repeat the same procedure. After the DIVEt code is filed in to the image, another dialog box pops up telling you to follow the instructions on the Transcript. Go to the Transcript, select 'DIVEt open' and 'do it'. The main DIVEt window will open with nothing in it. At this point you can either click *open* under the *file* menu to open one of the BankVerX.msc files located on the DIVEt disk, or you may begin entering your own testing data.

Note: If you do not file in the code from the A: drive, make sure that you specify the proper path to the source code when installing both disks. For example if you're installing from 'C:\Tgen\' and 'C:\DIVEt\' be sure to replace the default 'A:\' and type these into their respective dialog boxes.

Appendix B: MSC Grammar For Tgen

Token Classes

```

<space> :
    [\s\t\r]+ {ignoreDelimiter} ;
<name> :
    ([a-zA-Z0-9#\`@\[\]\{\}\|\~^] | .)+
    (_ [a-zA-Z0-9#\`@\[\]\{\}\|\~^]
    ([a-zA-Z0-9#\`@\[\]\{\}\|\~^] | .)*)* ;
<ambiguousName> :
    _([a-zA-Z0-9#\`@\[\]\{\}\|\~^] | .)*
    (_ [a-zA-Z0-9#\`@\[\]\{\}\|\~^]
    ([a-zA-Z0-9#\`@\[\]\{\}\|\~^] | .)*)* ;
<characterString> :
    '([a-zA-Z0-9#\`@\[\]\{\}\|\~^]?&%\+-
    !/\>\*\`\"\\<=\\(\\),\\;\\:._\\s\t\r] | '' ) * ' ;
<text> :
    \<\< [a-zA-Z0-9#\`@\[\]\{\}\|\~^]?&%\+-
    !/"\>"\*\`\"\\<=\\(\\),\\;\\:._\\s\t\r']* \>\> ;

```

"Message Sequence Chart Document"

```

MSCDocument : DocumentHead MessageSequenceChartList
{mscDocumentDocumentHead:mscList:} ;
End : Comment ';' {doNothingWith:}
    | ';' {doNothing} ;
Comment : 'comment' CharacterString {doNothingWith:} ;
TextDefinition : 'text' CharacterString End
{doNothingWith:with:} ;
DocumentHead : 'mscdocument' Name End {doNothingWith:with:}
    | 'mscdocument' Name 'related to' SDLReference End
{doNothingWith:with:with:} ;
SDLReference : Identifier {doNothingWith:} ;
Identifier : Qualifier Name {identifierQualifier:name:}
    | Name {answerArgument:} ;
Qualifier : Text {doNothingWith:} ;

```

"Message Sequence Chart"

```

MessageSequenceChartList : {OrderedCollection}
    | MessageSequenceChart MessageSequenceChartList
{mscListMsc:mscList:} ;
MessageSequenceChart : 'msc' MSCHead MSCBody 'endmsc' End
{mscMscHead:mscBody:end:}

```

```

    | 'msc' MSCHead 'expr' MSCExpression 'endmsc' End
{mscMscHead:mscExpression:end:} ;
MSCHead : Name End MSCInterface
{mscHeadName:end:mscInterface:} ;
MSCInterface : MSCInstInterface MSCGateInterface
{doNothingWith:with:}
    | MSCGateInterface {doNothingWith:} ;
MSCInstInterface : 'inst' InstanceList End
{doNothingWith:with:} ;
InstanceList : Name {doNothingWith:}
    | Name (',' InstanceList) {doNothingWith:with:}
    | Name (':' InstanceKind) {doNothingWith:with:}
    | Name (':' InstanceKind) (',' InstanceList)
{doNothingWith:with:with:} ;
InstanceKind : KindDenominator Identifier
{doNothingWith:with:}
    | Identifier {doNothingWith:} ;
KindDenominator : 'system' {doNothing}
    | 'block' {doNothing}
    | 'process' {doNothing}
    | 'service' {doNothing}
    | Name {doNothing} ;
MSCGateInterface : MSCGateDefList {doNothingWith:} ;
MSCGateDefList : {doNothing}
    | MSCGateDef MSCGateDefList {doNothingWith:with:} ;
MSCGateDef : 'gate' MsgGate End {doNothingWith:with:}
    | 'gate' OrderGate End {doNothingWith:with:} ;
MsgGate : DefInGate {doNothingWith:}
    | DefOutGate {doNothingWith:} ;
OrderGate : DefOrderInGate {doNothingWith:}
    | DefOrderOutGate {doNothingWith:} ;
MSCBody : MSCStatementList {mscBodyMscStatementList:} ;
MSCStatementList : {OrderedCollection}
    | MSCStatement MSCStatementList
{mscStatementListMscStatement:mscStatementList:} ;
MSCStatement : TextDefinition {doNothingWith:}
    | OldInstanceHeadStatement InstanceEventList
{mscStatementOldInstanceHeadStatement:instanceEventList:}
    | Name ':' InstanceHeadStatement End InstanceEventList
{mscStatementName:instanceHeadStatment:end:instanceEventList:}
    | Name ':' InstanceHeadStatement End
{doNothingWith:with:with:}
    | Name ':' InstanceEventList
{mscStatementName:instanceEventList:}
    | Name ':' MultiInstanceEventList
{mscStatementName:multiInstanceEventList:}
    | Name ',' InstanceNameList ':' MultiInstanceEventList
{mscStatementName:instanceNameList:multiInstanceEventList:}
    | 'all' ':' MultiInstanceEventList {answerArgument:} ;

```

```

InstanceEventList : InstanceEvent End
{instanceEventListInstanceEvent:end:}
  | InstanceEvent End InstanceEventList
{instanceEventListInstanceEvent:end:instanceEventList:} ;
InstanceEvent : OrderableEvent {answerArgument:}
  | NonOrderableEvent {answerArgument:} ;
OrderableEvent : OrderableEvent2 {answerArgument:}
  | AmbiguousName OrderableEvent2 {doNothingWith:with:}
  | OrderableEvent2 ('before' EventNameList)
{doNothingWith:with:}
  | AmbiguousName OrderableEvent2 ('before'
EventNameList) {doNothingWith:with:with:} ;
OrderableEvent2 : MessageEvent {answerArgument:}
  | IncompleteMessageEvent {doNothingWith:}
  | Create {answerArgument:}
  | TimerStatement {doNothingWith:}
  | Action {doNothingWith:} ;
EventNameList : Name {doNothingWith:}
  | Name (',' EventNameList) {doNothingWith:with:} ;
NonOrderableEvent : Coregion {answerArgument:}
  | SharedCondition {doNothingWith:}
  | SharedMSCReference {doNothingWith:}
  | SharedInlineExpr {doNothingWith:}
  | InstanceEndStatement {doNothingWith:}
  | Stop {doNothingWith:} ;
InstanceNameList : InstanceNameList2 {doNothingWith:}
  | 'all' {doNothing} ;
InstanceNameList2 : {doNothing}
  | Name InstanceNameList2 {doNothingWith:with:}
  | ',' Name InstanceNameList2 {doNothingWith:with:} ;
MultiInstanceEventList : MultiInstanceEvent End
{multiInstanceEventListMultiInstanceEvent:end:}
  | MultiInstanceEvent End MultiInstanceEventList
{multiInstanceEventListMultiInstanceEvent:end:multiInstanceE
ventList:} ;
MultiInstanceEvent : Condition {doNothingWith:}
  | MSCReference {answerArgument:}
  | InlineExpr {answerArgument:} ;
OldInstanceHeadStatement : 'instance' Name End
{oldInstanceHeadStatementName:end:}
  | 'instance' Name InstanceKind End
{oldInstanceHeadStatementName:instanceKind:end:}
  | 'instance' Name ':' InstanceKind End
{oldInstanceHeadStatementName:instanceKind:end:}
  | 'instance' Name Decomposition End
{oldInstanceHeadStatementName:decomposition:end:}
  | 'instance' Name InstanceKind Decomposition End
{oldInstanceHeadStatementName:instanceKind:decomposition:end
:}

```

```

    | 'instance' Name ':' InstanceKind Decomposition End
{oldInstanceHeadStatementName:instanceKind:decomposition:end
:} ;

```

"Instance"

```

InstanceHeadStatement : 'instance' {doNothing}
    | 'instance' InstanceKind {doNothingWith:}
    | 'instance' Decomposition {doNothingWith:}
    | 'instance' InstanceKind Decomposition
{doNothingWith:with:} ;
InstanceEndStatement : 'endinstance' {doNothing} ;

```

"Message"

```

MessageEvent : MessageOutput {answerArgument:}
    | MessageInput {answerArgument:} ;
MessageOutput : 'out' MsgIdentification 'to' InputAddress
{messageOutputMsgIdentification:inputAddress:} ;
MessageInput : 'in' MsgIdentification 'from' OutputAddress
{messageInputMsgIdentification:outputAddress:} ;
IncompleteMessageEvent : IncompleteMessageOutput
{doNothingWith:}
    | IncompleteMessageInput {doNothingWith:} ;
IncompleteMessageOutput : 'out' MsgIdentification 'to lost'
{doNothingWith:}
    | 'out' MsgIdentification 'to lost' InputAddress
{doNothingWith:with:} ;
IncompleteMessageInput : 'in' MsgIdentification 'from found'
{doNothingWith:}
    | 'in' MsgIdentification 'from found' OutputAddress
{doNothingWith:with:} ;
MsgIdentification : Name {answerArgument:}
    | Name ',' Name {doNothingWith:with:}
    | Name '(' ParameterList ')'
{msgIdentificationName:parameterList:}
    | Name ',' Name '(' ParameterList ')'
{doNothingWith:with:with:} ;
ParameterList : {doNothing}
    | Name ParameterList {doNothingWith:with:}
    | ',' Name ParameterList {doNothingWith:with:} ;
OutputAddress : Name {answerInstanceName:}
    | 'env' {environment}
    | ReferenceIdentification {doNothingWith:}
    | 'env' 'via' Name {environmentViaName:}
    | ReferenceIdentification 'via' Name
{doNothingWith:with:} ;
ReferenceIdentification : 'reference' Name {doNothingWith:}
    | 'inline' Name {doNothingWith:} ;
InputAddress : Name {answerInstanceName:}

```



```

    | 'env' {environment}
    | ReferenceIdentification {doNothingWith:}
    | 'env' 'via' Name {environmentViaName:}
    | ReferenceIdentification 'via' Name
{doNothingWith:with:} ;

```

"Environment and Gates"

```

ActualOutGate : 'out' MsgIdentification 'to' InputDest
{doNothingWith:with:}
    | Name 'out' MsgIdentification 'to' InputDest
{doNothingWith:with:with:} ;
ActualInGate : 'in' MsgIdentification 'from' OutputDest
{doNothingWith:with:}
    | Name 'in' MsgIdentification 'from' OutputDest
{doNothingWith:with:with:} ;
InputDest : 'lost' {doNothing}
    | 'lost' InputAddress {doNothingWith:}
    | InputAddress {doNothing} ;
OutputDest : 'found' {doNothing}
    | 'found' OutputAddress {doNothingWith:}
    | OutputAddress {doNothing} ;
DefInGate : 'out' MsgIdentification 'to' InputDest
{doNothingWith:with:}
    | Name 'out' MsgIdentification 'to' InputDest
{doNothingWith:with:with:} ;
DefOutGate : 'in' MsgIdentification 'from' OutputDest
{doNothingWith:with:}
    | Name 'in' MsgIdentification 'from' OutputDest
{doNothingWith:with:with:} ;
ActualOrderOutGate : Name 'before' OrderDest
{doNothingWith:with:} ;
OrderDest : Name {doNothing}
    | 'env' 'via' Name {doNothingWith:}
    | ReferenceIdentification 'via' Name
{doNothingWith:with:} ;
ActualOrderInGate : Name {doNothingWith:} ;
DefOrderInGate : Name 'before' OrderDest
{doNothingWith:with:} ;
DefOrderOutGate : Name {doNothingWith:} ;
InlineOutGate : DefOutGate {doNothingWith:}
    | DefOutGate 'external out' MsgIdentification 'to'
InputDest {doNothingWith:with:with:} ;
InlineInGate : DefInGate {doNothingWith:}
    | DefInGate 'external in' MsgIdentification 'from'
OutputDest {doNothingWith:with:with:} ;
InlineOrderOutGate : Name {doNothingWith:}
    | Name 'external before' OrderDest
{doNothingWith:with:} ;

```

```

InlineOrderInGate : Name 'before' OrderDest
{doNothingWith:with:}
    | Name 'before' OrderDest 'external'
{doNothingWith:with:} ;

```

"Condition"

```

SharedCondition : ConditionIdentification Shared
{doNothingWith:with:} ;
ConditionIdentification : 'condition' ConditionNameList
{doNothingWith:} ;
ConditionNameList : {doNothing}
    | Name ConditionNameList {doNothingWith:with:}
    | ',' Name ConditionNameList {doNothingWith:with:} ;
Shared : 'shared' {doNothing}
    | 'shared' 'all' {doNothing}
    | 'shared' SharedInstanceList {doNothingWith:} ;

SharedInstanceList : Name {doNothingWith:}
    | Name ',' SharedInstanceList {doNothingWith:with:} ;
Condition : ConditionIdentification {doNothingWith:} ;

```

"Timer"

```

TimerStatement : Set {doNothingWith:}
    | Reset {doNothingWith:}
    | Timeout {doNothingWith:} ;
Set : 'set' Name {doNothingWith:}
    | 'set' Name ',' Name {doNothingWith:with:}
    | 'set' Name '(' Name ')' {doNothingWith:with:}
    | 'set' Name ',' Name '(' Name ')'
{doNothingWith:with:with:} ;
Reset : 'reset' Name {doNothingWith:}
    | 'reset' Name ',' Name {doNothingWith:with:} ;
Timeout : 'timeout' Name {doNothingWith:}
    | 'timeout' Name ',' Name {doNothingWith:with:} ;

```

"Action"

```

Action : 'action' CharacterString {doNothingWith:} ;

```

"Instance Creation"

```

Create : 'create' Name {createName:}
    | 'create' Name '(' ParameterList ')'
{createName:parameterList:} ;

```

"Instance Stop"

```

Stop : 'stop' {doNothing} ;

```

"Coregion"

```

Coregion : 'concurrent' End CoeventList 'endconcurrent'
{coregionEnd:coeventList:} ;
CoeventList : {OrderedCollection}
    | Coevent CoeventList {coeventListCoevent:coeventList:}
;
Coevent : OrderableEvent End {coeventOrderableEvent:end:} ;

```

"Instance Decomposition"

```

Decomposition : 'decomposed' {doNothing}
    | 'decomposed' SubstructureReference {doNothingWith:} ;
SubstructureReference : 'as' Name {doNothingWith:} ;

```

"Inline Expression"

```

SharedInlineExpr : SharedLoopExpr {doNothingWith:}
    | SharedOptExpr {doNothingWith:}
    | SharedAltExpr {doNothingWith:}
    | SharedParExpr {doNothingWith:}
    | SharedExcExpr {doNothingWith:} ;
SharedLoopExpr : 'loopbegin' Shared End InstanceEventList
'loopend' {doNothingWith:with:with:}
    | 'loop' LoopBoundary 'begin' Shared End
InstanceEventList 'loopend' {doNothingWith:with:with:with:}
    | 'loopbegin' Name Shared End InstanceEventList
'loopend' {doNothingWith:with:with:with:}
    | 'loopbegin' Shared End InlineGateInterface
InstanceEventList 'loopend' {doNothingWith:with:with:with:}
    | 'loop' LoopBoundary 'begin' Name Shared End
InstanceEventList 'loopend'
{doNothingWith:with:with:with:with:}
    | 'loop' LoopBoundary 'begin' Shared End
InlineGateInterface InstanceEventList 'loopend'
{doNothingWith:with:with:with:with:}
    | 'loopbegin' Name Shared End InlineGateInterface
InstanceEventList 'loopend'
{doNothingWith:with:with:with:with:}
    | 'loop' LoopBoundary 'begin' Name Shared End
InlineGateInterface InstanceEventList 'loopend'
{doNothingWith:with:with:with:with:with:} ;
SharedOptExpr : 'optbegin' Shared End InstanceEventList
'optend' {doNothingWith:with:with:}
    | 'optbegin' Name Shared End InstanceEventList 'optend'
{doNothingWith:with:with:with:}
    | 'optbegin' Shared End InlineGateInterface
InstanceEventList 'optend' {doNothingWith:with:with:with:}

```

```

    | 'optbegin' Name Shared End InlineGateInterface
InstanceEventList 'optend'
{doNothingWith:with:with:with:with:} ;
SharedExcExpr : 'excbegin' Shared End InstanceEventList
'excend' {doNothingWith:with:with:}
    | 'excbegin' Name Shared End InstanceEventList 'excend'
{doNothingWith:with:with:with:with:}
    | 'excbegin' Shared End InlineGateInterface
InstanceEventList 'excend' {doNothingWith:with:with:with:with:}
    | 'excbegin' Name Shared End InlineGateInterface
InstanceEventList 'excend'
{doNothingWith:with:with:with:with:with:} ;
SharedAltExpr : 'altbegin' Shared End InstanceEventList
AltList 'altend' {doNothingWith:with:with:with:with:}
    | 'altbegin' Name Shared End InstanceEventList AltList
'altend' {doNothingWith:with:with:with:with:with:}
    | 'altbegin' Shared End InlineGateInterface
InstanceEventList AltList 'altend'
{doNothingWith:with:with:with:with:with:with:}
    | 'altbegin' Name Shared End InlineGateInterface
InstanceEventList AltList 'altend'
{doNothingWith:with:with:with:with:with:with:with:} ;
AltList : {doNothing}
    | 'alt' End InstanceEventList AltList
{doNothingWith:with:with:}
    | 'alt' End InlineGateInterface InstanceEventList
AltList {doNothingWith:with:with:with:with:} ;
SharedParExpr : 'parbegin' Shared End InstanceEventList
ParList 'parend' {doNothingWith:with:with:with:with:}
    | 'parbegin' Name Shared End InstanceEventList ParList
'parend' {doNothingWith:with:with:with:with:with:}
    | 'parbegin' Shared End InlineGateInterface
InstanceEventList ParList 'parend'
{doNothingWith:with:with:with:with:with:}
    | 'parbegin' Name Shared End InlineGateInterface
InstanceEventList ParList 'parend'
{doNothingWith:with:with:with:with:with:with:with:} ;
ParList : {doNothing}
    | 'par' End InstanceEventList {doNothingWith:with:}
    | 'par' End InlineGateInterface InstanceEventList
{doNothingWith:with:} ;
InlineExpr : LoopExpr {answerArgument:}
    | OptExpr {answerArgument:}
    | AltExpr {answerArgument:}
    | ParExpr {parExpr:}
    | ExcExpr {doNothingWith:} ;
LoopExpr : 'loopbegin' End MSCBody 'loopend'
{loopExprEnd:mscBody:}
    | 'loop' LoopBoundary 'begin' End MSCBody 'loopend'
{loopExprLoopBoundary:end:mscBody:}

```

```

        | 'loopbegin' Name End MSCBody 'loopend'
{loopExprName:end:mscBody:}
        | 'loopbegin' End InlineGateInterface MSCBody 'loopend'
{loopExprEnd:inlineGateInterface:mscBody:}
        | 'loop' LoopBoundary 'begin' Name End MSCBody
'loopend' {loopExprLoopBoundary:name:end:mscBody:}
        | 'loop' LoopBoundary 'begin' End InlineGateInterface
MSCBody 'loopend'
{loopExprLoopBoundary:end:inlineGateInterface:mscBody:}
        | 'loopbegin' Name End InlineGateInterface MSCBody
'loopend' {loopExprName:end:inlineGateInterface:mscBody:}
        | 'loop' LoopBoundary 'begin' Name End
InlineGateInterface MSCBody 'loopend'
{loopExprLoopBoundary:name:end:inlineGateInterface:mscBody:}
;
OptExpr : 'optbegin' End MSCBody 'optend'
{optExprEnd:mscBody:}
        | 'optbegin' Name End MSCBody 'optend'
{optExprName:end:mscBody:}
        | 'optbegin' End InlineGateInterface MSCBody 'optend'
{optExprEnd:inlineGateInterface:mscBody:}
        | 'optbegin' Name End InlineGateInterface MSCBody
'optend' {optExprName:end:inlineGateInterface:mscBody:} ;
ExcExpr : 'excbegin' End MSCBody 'excend'
{doNothingWith:with:}
        | 'excbegin' Name End MSCBody 'excend'
{doNothingWith:with:with:}
        | 'excbegin' End InlineGateInterface MSCBody 'excend'
{doNothingWith:with:with:}
        | 'excbegin' Name End InlineGateInterface MSCBody
'excend' {doNothingWith:with:with:with:} ;
AltExpr : 'altbegin' End MSCBody AltExprList 'altend'
{altExprEnd:mscBody:altExprList:}
        | 'altbegin' Name End MSCBody AltExprList 'altend'
{altExprName:end:mscBody:altExprList:} ;
AltExprList : {doNothing}
        | 'alt' End MSCBody AltExprList
{altExprListEnd:mscBody:altExprList:}
        | 'alt' End InlineGateInterface MSCBody AltExprList
{altExprList:inlineGateInterface:mscBody:altExprList:} ;
ParExpr : 'parbegin' End MSCBody ParExprList 'parend'
{parExprEnd:mscBody:parExprList:}
        | 'parbegin' Name End MSCBody ParExprList 'parend'
{parExprName:end:mscBody:parExprList:}
        | 'parbegin' End InlineGateInterface MSCBody
ParExprList 'parend'
{parExprEnd:inlineGateInterface:mscBody:parExprList:}
        | 'parbegin' Name End InlineGateInterface MSCBody
ParExprList 'parend'
{parExprName:end:inlineGateInterface:mscBody:parExprList:} ;

```

```

ParExprList : {OrderedCollection}
  | 'par' End MSCBody ParExprList
{parExprListEnd:mscBody:parExprList:}
  | 'par' End InlineGateInterface MSCBody ParExprList
{parExprListEnd:inlineGateInterface:mscBody:parExprList:} ;
LoopBoundary : '<' InfNatural '>' {loopBoundaryInfNatural:}
  | '<' InfNatural ',' InfNatural '>'
{loopBoundaryInfNatural:infNatural:} ;
InfNatural : 'inf' {infNaturalInf}
  | NameList {infNaturalNameList:} ;
NameList : Name {answerArgument:}
  | Name NameList {nameListName:nameList:} ;
InlineGateInterface : InlineGateList {doNothingWith:} ;
InlineGateList : 'gate' InlineGate End {doNothingWith:with:}
  | 'gate' InlineGate End InlineGateList
{doNothingWith:with:with:} ;
InlineGate : InlineOutGate {doNothingWith:}
  | InlineInGate {doNothingWith:}
  | InlineOrderOutGate {doNothingWith:}
  | InlineOrderInGate {doNothingWith:} ;

```

"MSC Reference"

```

SharedMSCReference : 'reference' MSCRefExpr Shared
ReferenceGateInterface {doNothingWith:with:with:}
  | 'reference' Name ':' MSCRefExpr Shared
ReferenceGateInterface {doNothingWith:with:with:with:} ;
MSCReference : 'reference' MSCRefExpr ReferenceGateInterface
{mscReferenceMSCRefExpr:referenceGateInterface:}
  | 'reference' Name ':' MSCRefExpr
ReferenceGateInterface
{mscReferenceName:mscRefExpr:referenceGateInterface:} ;
MSCRefExpr : MSCRefParExpr MSCRefParExprList
{mscRefExprMSCRefExpr:mscRefExprList:} ;
MSCRefParExprList : {doNothing}
  | 'alt' MSCRefParExpr MSCRefParExprList
{mscRefParExprListMSCRefParExpr:mscRefParExprList:} ;
MSCRefParExpr : MSCRefSeqExpr MSCRefSeqExprList
{mscRefExprMSCRefExpr:mscRefExprList:} ;
MSCRefSeqExprList : {doNothing}
  | 'par' MSCRefSeqExpr MSCRefSeqExprList
{mscRefSeqExprListMSCRefSeqExpr:mscRefSeqExprList:} ;
MSCRefSeqExpr : MSCRefExcExpr MSCRefExcExprList
{mscRefExprMSCRefExpr:mscRefExprList:} ;
MSCRefExcExprList : {doNothing}
  | 'seq' MSCRefExcExpr MSCRefExcExprList
{mscRefExprListMSCRefExpr:mscRefExcExprList:} ;
MSCRefExcExpr : MSCRefOptExpr {answerArgument:}
  | 'exc' MSCRefOptExpr {mscRefExcExprMSCRefOptExpr:} ;
MSCRefOptExpr : MSCRefLoopExpr {answerArgument:}

```

```

    | 'opt' MSCRefLoopExpr {mScRefOptExprMSCRefLoopExpr:} ;
MSCRefLoopExpr : MSCRefLoopExpr2 {answerArgument:}
    | 'loop' MSCRefLoopExpr2
    {mScRefLoopExprMSCRefLoopExpr2:}
    | LoopBoundary MSCRefLoopExpr2
    {mScRefLoopExprLoopBoundary:mScRefLoopExpr2:}
    | 'loop' LoopBoundary MSCRefLoopExpr2
    {mScRefLoopExprLoopBoundary:mScRefLoopExpr2:} ;
MSCRefLoopExpr2 : 'empty' {doNothing}
    | Name {answerArgument:}
    | Name ParameterSubstitution
    {mScRefLoopExpr2Name:parameterSubstitution:}
    | '(' MSCRefExpr ')' {answerArgument:} ;
ParameterSubstitution : 'subst' SubstitutionList
    {doNothingWith:} ;
SubstitutionList : {doNothing}
    | Substitution SubstitutionList {doNothingWith:with:}
    | ',' Substitution SubstitutionList
    {doNothingWith:with:} ;
Substitution : Attribute 'empty' 'by' 'empty'
    {doNothingWith:}
    | Attribute 'empty' 'by' Name {doNothingWith:with:}
    | Attribute Name 'by' 'empty' {doNothingWith:with:}
    | Attribute Name 'by' Name {doNothingWith:with:with:} ;
Attribute : 'msg' {doNothing}
    | 'inst' {doNothing}
    | 'mSc' {doNothing} ;
ReferenceGateInterface : {doNothing}
    | 'gate' RefGate ReferenceGateInterface
    {doNothingWith:with:} ;
RefGate : ActualOutGate {doNothingWith:}
    | ActualInGate {doNothingWith:}
    | ActualOrderOutGate {doNothingWith:}
    | ActualOrderInGate {doNothingWith:} ;

```

"High-Level MSC (HMSC)"

```

MSCExpression : Start NodeExpressionList
    {mScExpressionStart:nodeExpressionList:} ;
NodeExpressionList : {doNothing}
    | NodeExpression NodeExpressionList
    {nodeExpressionListNodeExpression:nodeExpressionList:} ;
Start : Name AltNameList End {doNothingWith:with:with:} ;
AltNameList : {doNothing}
    | 'alt' Name AltNameList {doNothingWith:with:} ;
NodeExpression : AmbiguousName ':' NodeExpression2 End
    {nodeExpressionAmbiguousName:nodeExpression2:end:} ;
NodeExpression2 : Node 'seq' '(' Name AltNameList ')'
    {nodeExpression2Node:name:altNameList:}
    | 'end' {doNothing} ;

```

```

Node : '(' MSCRefExpr ')' {answerArgument:}
      | 'empty' {doNothing}
      | Name {answerArgument:}
      | ParExpression {parExpr:}
      | 'condition' ConditionNameList {doNothingWith:}
      | 'connect' {doNothing} ;
ParExpression : 'expr' MSCExpression 'endexpr'
ParExpressionList
{parExpressionMSCExpression:parExpressionList:} ;
ParExpressionList : {OrderedCollection}
      | 'par expr' MSCExpression 'endexpr' ParExpressionList
{parExpressionListMscExpression:parExpressionList:} ;

```

"Token Classes"

```

Name : <name> {answerArgument:} ;
AmbiguousName : <ambiguousName> {answerArgument:} ;
CharacterString : <characterString> {answerArgument:} ;
Text : <text> {answerArgument:} ;

```


Appendix C: BankVer1.msc

1 MSC Document

```
mscdocument Bank_System;
```

```
msc OpenSavingsAccount;
```

```
inst bank, savingsAccount;
```

```
bank:
```

```
    in openAccount.for. from env;
```

```
    out new to SavingsAccount;
```

```
SavingsAccount:
```

```
    in new from bank;
```

```
    out initialize to savingsAccount;
```

```
savingsAccount:
```

```
    in initialize from SavingsAccount;
```

```
    out newAccountNumber to BankAccount;
```

```
BankAccount:
```

```
    in newAccountNumber from savingsAccount;
```

```
bank:
```

```
    out owner. to savingsAccount;
```

```
savingsAccount:
```

```
    in owner. from bank;
```

```
bank:
```

```
    out addAccount to bank;
```

```
    in addAccount. from bank;
```

```
endmsc;
```

```
msc Deposit;
```

```
inst bankAccount;
```

```
bankAccount:
```

```
    in deposit. from env;
```

```
    out balance to bankAccount;
```

```
    in balance from bankAccount;
```

```
    out balance. to bankAccount;
```

```
    in balance. from bankAccount;
```

```
endmsc;
```

```
msc OpenBank;
```

```
inst bank;
```

```
Bank:
```

```
    in new from env;
```

```
    out initialize to bank;
```

```
bank:
```

```
    in initialize from Bank;
```

```

all: optbegin;
    bank:
        in name. from env;
optend;
endmsc;

msc CheckingAccountCustomers;
inst bank;
bank:
    in customers from env;
    out accounts to bank;
    in accounts from bank;
all: loopbegin;
    checkingAccount:
        in owner from env;
    loopend;
endmsc;

msc MultipleRequestsFromFirstAccount;
inst bank, savingsAccount;
bank:
    in multipleRequestsFromFirstAccount from env;
savingsAccount:
    concurrent;
        in owner from bank;
        in balance from bank;
        in accountNumber from bank;
    endconcurrent;
endmsc;

msc PayMonthlyInterest;
inst bank1;
bank1:
    in payMonthlyInterest from env;
    out accounts to bank1;
    in accounts from bank1;
all: loopbegin;
    all: altbegin;
        checkingAccount:
            in payMonthlyInterest from env;
        alt;
            savingsAccount:
                in payMonthlyInterest from env;
                out interestRate to BankAccount;
            BankAccount:
                in interestRate from savingsAccount;
        alt;
            termDeposit:
                in payMonthlyInterest from env;
                out interestRate to BankAccount;
    altend;
endmsc;

```

```

        BankAccount:
            in interestRate from termDeposit;
        altend;
    loopend;
endmsc;

```

2 Trace grammar

```

Start : OpenSavingsAccount | Deposit | OpenBank |
CheckingAccountCustomers | MultipleRequestsFromFirstAccount
| PayMonthlyInterest ;

```

```

OpenSavingsAccount : ('#openAccount:for:,env,bank' <number>
';' '#new,bank' <number> ',SavingsAccount;'
'#initialize,SavingsAccount,savingsAccount' <number> ';'
'#newAccountNumber,savingsAccount' <number> ',BankAccount;'
'#owner:,bank' <number> ',savingsAccount' <number> ';'
('#addAccount,bank' <number> ',bank' <number> ';')?
('#addAccount:,bank' <number> ',bank' <number> ';')? );

```

```

Deposit : ('#deposit:,env,bankAccount' <number> ';'
('#balance,bankAccount' <number> ',bankAccount' <number>
';')? ('#balance:,bankAccount' <number> ',bankAccount'
<number> ';')? );

```

```

OpenBank : ('#new,env,Bank;' '#initialize,Bank,bank'
<number> ';' ('#name:,env,bank' <number> ';')? );

```

```

CheckingAccountCustomers : ('#customers,env,bank' <number>
';' ('#accounts,bank' <number> ',bank' <number> ';')?
('#owner,env,checkingAccount' <number> ';')* );

```

```

MultipleRequestsFromFirstAccount :
('#multipleRequestsFromFirstAccount,env,bank' <number> ';'
('#owner,bank' <number> ',savingsAccount' <number> ';'
'#balance,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number>
';'
| '#owner,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number>
';' '#balance,bank' <number> ',savingsAccount' <number> ';'
| '#balance,bank' <number> ',savingsAccount' <number>
';' '#owner,bank' <number> ',savingsAccount' <number> ';'
'#accountNumber,bank' <number> ',savingsAccount' <number>
';'
| '#balance,bank' <number> ',savingsAccount' <number>
';' '#accountNumber,bank' <number> ',savingsAccount'
<number> ';')? ('#owner,bank' <number> ',savingsAccount'
<number> ';')? );

```

```

        | '#accountNumber,bank' <number> ',savingsAccount'
<number> ';' '#owner,bank' <number> ',savingsAccount'
<number> ';' '#balance,bank' <number> ',savingsAccount'
<number> ';'
        | '#accountNumber,bank' <number> ',savingsAccount'
<number> ';' '#balance,bank' <number> ',savingsAccount'
<number> ';' '#owner,bank' <number> ',savingsAccount'
<number> ';' ) );

PayMonthlyInterest : ('#payMonthlyInterest,env,bank1;'
('#accounts,bank1,bank1;')?
(('#payMonthlyInterest,env,checkingAccount' <number> ';')
 | ('#payMonthlyInterest,env,savingsAccount' <number>
';' '#interestRate,savingsAccount' <number> ',BankAccount;'
)
 | ('#payMonthlyInterest,env,termDeposit' <number> ';'
'#interestRate,termDeposit' <number> ',BankAccount;' ))* );

```

3 Trace Classes

Bank
BankAccount

Appendix D: BankVer2.msc

1 MSC Document

```

mscdocument Bank_System;

msc PayMonthlyInterest;
inst bank1;
bank1:
    in payMonthlyInterest from env;
    out accounts to bank1;
    in accounts from bank1;
all: loopbegin;
    all: altbegin;
        all: reference
CheckingAccountPayMonthlyInterest;
        alt;
            all: reference
SavingsAccountPayMonthlyInterest;
            alt;
                all: reference TermDepositPayMonthlyInterest;
            altend;
    loopend;
endmsc;

msc CheckingAccountPayMonthlyInterest;
inst checkingAccount;
checkingAccount:
    in payMonthlyInterest from env;
endmsc;

msc SavingsAccountPayMonthlyInterest;
inst savingsAccount;
savingsAccount:
    in payMonthlyInterest from env;
    out interestRate to BankAccount;
BankAccount:
    in interestRate from savingsAccount;
endmsc;

msc TermDepositPayMonthlyInterest;
inst termDeposit;
termDeposit:
    in payMonthlyInterest from env;
    out interestRate to BankAccount;

```

```
BankAccount:
    in interestRate from termDeposit;
endmsc;
```

2 Trace grammar

```
Start : PayMonthlyInterest |
CheckingAccountPayMonthlyInterest |
SavingsAccountPayMonthlyInterest |
TermDepositPayMonthlyInterest ;

PayMonthlyInterest : ('#payMonthlyInterest,env,bank1;'
('#accounts,bank1,bank1;')?
((CheckingAccountPayMonthlyInterest)
 | (SavingsAccountPayMonthlyInterest)
 | (TermDepositPayMonthlyInterest))* );

CheckingAccountPayMonthlyInterest :
('#payMonthlyInterest,env,checkingAccount' <number> ';' );

SavingsAccountPayMonthlyInterest :
('#payMonthlyInterest,env,savingsAccount' <number> ';'
'#interestRate,savingsAccount' <number> ',BankAccount;' );

TermDepositPayMonthlyInterest :
('#payMonthlyInterest,env,termDeposit' <number> ';'
'#interestRate,termDeposit' <number> ',BankAccount;' );
```

3 Trace Classes

```
Bank
BankAccount
```

Appendix E: BankVer3.msc

1 MSC Document

```

mscdocument Bank_System;

msc PayMonthlyInterest;
inst bank1;
bank1:
    in payMonthlyInterest from env;
    out accounts to bank1;
    in accounts from bank1;
all: parbegin;
    all: reference CheckingAccountPayMonthlyInterest;
par;
    all: reference SavingsAccountPayMonthlyInterest;
par;
    all: reference TermDepositPayMonthlyInterest;
parend;
endmsc;

msc CheckingAccountPayMonthlyInterest;
inst checkingAccount;
checkingAccount:
    in payMonthlyInterest from env;
endmsc;

msc SavingsAccountPayMonthlyInterest;
inst savingsAccount;
savingsAccount:
    in payMonthlyInterest from env;
    out interestRate to BankAccount;
BankAccount:
    in interestRate from savingsAccount;
endmsc;

msc TermDepositPayMonthlyInterest;
inst termDeposit;
termDeposit:
    in payMonthlyInterest from env;
    out interestRate to BankAccount;
BankAccount:
    in interestRate from termDeposit;
endmsc;

```

2 Trace grammar

```

Start : PayMonthlyInterest |
CheckingAccountPayMonthlyInterest |
SavingsAccountPayMonthlyInterest |
TermDepositPayMonthlyInterest ;

PayMonthlyInterest : ('#payMonthlyInterest,env,bank1;'
('#accounts,bank1,bank1;')?
((CheckingAccountPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
| (CheckingAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
| (SavingsAccountPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
| (SavingsAccountPayMonthlyInterest)
(TermDepositPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
| (TermDepositPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
| (TermDepositPayMonthlyInterest)
(SavingsAccountPayMonthlyInterest)
(CheckingAccountPayMonthlyInterest)) );

CheckingAccountPayMonthlyInterest :
('#payMonthlyInterest,env,checkingAccount' <number> ';' );

SavingsAccountPayMonthlyInterest :
('#payMonthlyInterest,env,savingsAccount' <number> ';'
'#interestRate,savingsAccount' <number> ',BankAccount;' );

TermDepositPayMonthlyInterest :
('#payMonthlyInterest,env,termDeposit' <number> ';'
'#interestRate,termDeposit' <number> ',BankAccount;' );

```

3 Trace Classes

```

Bank
BankAccount

```


IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved