Program Behaviour Modelling with Flexible Logical Entity Abstraction

by

David Andrews

BComp (Hons 1)

A dissertation submitted to the School of Information Technology and Mathematical Sciences in full fulfilment of the requirements for the degree of

Doctorate of Philosophy

at the

University of Ballarat

June 2006

Signature of Author: ______________________________
PhD Candidate

Certified by: ______________________________
Philip Smith, Senior Lecturer, Principal Supervisor

____
David Stratton, Senior Lecturer, Associate Supervisor

____
John Wharington, Senior Research Fellow, Associate Supervisor

ABSTRACT

The use of program behaviour modelling (PBM) is widespread in several fields of computing including security and software development. The goal of PBM is to accurately and simply represent the defining behaviour of a computer program. Numerous techniques for PBM have been developed in the software development and computer security domains. Software development PBM techniques define a program's behaviour in terms of the interactions between its constituent components, while computer security PBM techniques most often use operating system interactions. The operating system provides a generic interface for all programs, which makes these techniques widely applicable. However, a typical program's operating system interactions are too numerous to easily manage, difficult for a human to comprehend, and individually insignificant.

Recent techniques for computer security PBM, employ a slightly more abstract approach. Increased abstraction allows a simpler, and more powerful view of program behaviour. These more abstract techniques enable groupings of operating system interactions into 'events'. These abstractions are purely theoretical, and the PBM models that employ them are still defined in terms of concrete operating system interactions.

This study extends the use of abstraction in PBM, and provides a flexible abstraction technique that allows modelling in terms of the logical abstract concepts with which a programs operates. This new technique is called *Logical Entity Abstracted Program Behaviour Modelling* (LEAPBM). The LEAPBM technique is evaluated within the High Level Architecture (HLA) distributed simulation domain. The HLA encompasses suitably complex programs which employ a range of system functionality; and is pervasive in important industries including aerospace and defence.

It is shown that LEAPBM, with the appropriate identification of abstract program concepts, provides improved accuracy and maintains acceptable complexity. It is intuitively expected that the human comprehensibility of LEAPBM models is simpler, due to the reduction in model size that flexible abstraction allows.

I, the author, certify that this dissertation and the thesis it describes, are my original work and have not been submitted, in part or in full, as part of the requirements for any other degree at any other academic institution. I also certify that, in my belief and to the best of my knowledge, this dissertation does not contain any material written or published by any other person, except as correctly acknowledged and cited. The work this dissertation presents was conducted during my period of candidature.

This dissertation may be freely copied and distributed for private and educational purposes. However, any future work by any parties which utilises the thesis contained herein, in any way, must fully and correctly acknowledge this dissertation as it applies.

Signature of Author: ______________________________

PhD Candidate
School of Information Technology and Mathematical Sciences
June 2006

ACKNOWLEDGEMENTS

The author's most sincere thanks go to a number of people whose unwavering assistance and support made this study possible and at times, even enjoyable. Most importantly the author wishes to thank his supervisors: Phil Smith, for his constant and friendly support, thoroughness and excellent feedback and guidance; Dave Stratton, for his constant enthusiasm and kind words of encouragement; Johnny Wharington, for his inspirational work, tutorage, encouragement and for the genesis of the author's post graduate studies.

Sincere and humble thanks go to Professor Sidney Morris for his kind assistance in the initial stages of this study, and for graciously spending a quantity of his valuable time assisting in preparation of the author's confirmation presentation.

The author would like to thank his fellow students, Tim Pokorny, Mick Fraser, John Avery and the rest of the late-night ITMS post graduate crew, for their warm companionship, easy laughter, and joint venture into the world of research. The author owes a debt to Anthony Cramp from the Defence Science Technology Organisation, who selflessly took time out from his own doctoral studies to assist in the provision of test environments for this study.

Finally, the author would like to acknowledge his friends, family and amazing girlfriend Heather. Their kind advice and understanding ears were an ever-present source of comfort and a support through-out this study.

TABLE OF CONTENTS

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 Goals . . . 2
1.2 Approach . . . 3
1.3 Organisation . . . 4
1.4 Outcomes . . . 5

Chapter 2: Background: Program Behaviour Modelling 6

2.1 History and Applications . . . 7
2.1.1 Software Development . . . 8
2.1.2 Computer Security . . . 10
2.1.3 Summary . . . 16
2.2 Model Basis . . . 16
2.2.1 Internal Software Components . . . 17
2.2.2 Operating System Audit Trails . . . 18
2.2.3 Operating System Calls . . . 21
2.2.4 Abstraction . . . 27
2.2.5 Summary . . . 29
2.3 Definition Techniques . . . 30
2.3.1 Order Inclusive . . . 31
2.3.2 Order Exclusive . . . 37
2.3.3 Partially Order Inclusive . . . 41
2.3.4 Summary . . . 44
2.4 Performance . . . 45
2.4.1 Accuracy . . . 45
2.4.2 Complexity . . . 47
2.5 Summary . . . 48

Chapter 3: Logical Entity Abstracted Program Behaviour Modelling 50

3.1 Design Methodology . . . 52
3.2 Logical Abstract Entities . . . 52
3.2.1 Software Library Terms . . . 53
3.2.2 LAE Classes . . . 57

3.2.3 LAE Methods . . . 57
3.2.4 LAE Properties . . . 60
3.2.5 Summary . . . 61
3.3 LEAPBM Models . . . 61
3.3.1 Program Identification and Representation Boundaries . . . 62
3.3.2 LEAPBM Interaction Finite State Automata . . . 62
3.3.3 LAE Instances and Use . . . 65
3.3.4 Summary . . . 65
3.4 Definition Formats . . . 66
3.4.1 Diagrams . . . 66
3.4.2 XML . . . 75
3.5 Generation Algorithm . . . 77
3.5.1 Stage 1 - Capturing Software Library Elements . . . 78
3.5.2 Stage 2 - Identifying LAEs . . . 80
3.5.3 Stage 3 - Recording Program Behaviour Flow and LAE Usage . . . 81
3.5.4 Summary . . . 85
3.6 Processing Algorithm . . . 85
3.6.1 Stage 1 - Identifying Allowed LAE Interactions . . . 86
3.6.2 Stage 2 - Identifying Allowed Software Library Elements . . . 88
3.6.3 Stage 3 - Checking Type and Name . . . 89
3.6.4 Stage 4 - Checking Data Values . . . 89
3.6.5 Summary . . . 90
3.7 Application Specifications . . . 90
3.8 Summary . . . 92

Chapter 4: Investigation Outline **93**
4.1 Domain: High Level Architecture Distributed Simulation . . . 93
4.1.1 Background . . . 94
4.1.2 Suitability for Investigation . . . 97
4.1.3 Test Environment . . . 99
4.2 Subject Systems . . . 101
4.2.1 Matis: LEAPBM Application Architecture . . . 102
4.2.2 STIDE: Existing Modelling Technique . . . 109
4.3 Investigations . . . 112

Chapter 5: Model Precision Investigation **113**
5.1 Task Description . . . 113
5.2 Flexible Model Basis . . . 115
5.2.1 Potential Choices . . . 115
5.2.2 Selection Considerations . . . 117
5.3 Demonstrations . . . 118
5.3.1 Further Investigation: STIDE Differentiation of HLA Federates . . . 122

5.3.2 Average-Branching Factor 123
5.4 Discussion 124
5.4.1 Limitations 126
5.4.2 Extensions 126

Chapter 6: Model Focus Investigation **128**
6.1 Task Description 128
6.2 Multiple Path Encapsulation 129
6.2.1 Encapsulation via Abstraction 130
6.2.2 Order Importance Modifications 133
6.3 Demonstrations 135
6.4 Discussion 138
6.4.1 Limitations 139
6.4.2 Extensions 139

Chapter 7: Model Complexity Investigation **141**
7.1 Task Description 141
7.2 Small and Fast Models 142
7.3 Human Comprehension 143
7.4 Model Generation 144
7.5 Demonstrations 145
7.5.1 Human Comprehensibility 149
7.5.2 Model Generation 153
7.6 Discussion 155
7.6.1 Limitations 156
7.6.2 Extensions 156

Chapter 8: Conclusion **157**
8.1 Contributions 157
8.1.1 Accuracy 157
8.1.2 Human Comprehension 159
8.1.3 Modelling Novel Program Types 159
8.2 Limitations 159
8.3 Implications and Future Directions 160

Chapter 9: References **164**

Appendix A: Utilised Software **184**
A.1 Air Transport Operations Federation 184
A.2 STIDE 186
A.2.1 Real-Time Processing Modifications 186
A.3 CPU Cycle Measurement 187

Appendix B: Laptop Computer **188**
B.1 Operating System Configuration 188

Appendix C: Additional Demonstrations **190**
C.1 Simple GTK Program System Call Usage 190
C.2 Demonstration: Simple HLA Federate System Call Usage 192

LIST OF FIGURES

2.1 Car State Machine 7
2.2 First Technique for Audit-Trail Examination 19
2.3 Second Technique for Audit-Trail Examination 20
2.4 Direct and Indirect Program System-call Usage 23
2.5 System-call Usage Captured via Static Analysis 25
2.6 Abstraction Example: C Code 28
2.7 Giffin et al. (2002) Example C Code 32
2.8 Giffin et al. (2002) Example Non-deterministic Finite Automaton 33
2.9 Giffin et al. (2002) Example Impossible Path through NFA 34
2.10 Giffin et al. (2002) Example Push-Down Automaton 35
2.11 Strictly Ordered Automaton with Order Unimportant Event 42
2.12 Partially Ordered Automaton 43

3.1 LAE Method FSA Example 58
3.2 LAE Method FSA with Exception Example 59
3.3 Multiple Sequence FSA Example 60
3.4 Standard Finite State Automaton 68
3.5 Entries Extended Finite State Automaton 69
3.6 Multiple Pathways Extended Finite State Automaton 71
3.7 Flexible Order Extended Finite State Automaton 72
3.8 LAE Interaction Referenced Extended Finite State Automaton 73
3.9 Example LAE Instance Usage Diagram 75
3.10 LAE Instance Interaction XML Element 76
3.11 Parsing Software Library Method Call Definitions 79
3.12 Parsing Software Library Variable Definitions 80
3.13 Interaction FSA Branching 83
3.14 Interaction FSA Converging 84
3.15 Interaction FSA Looping 85
3.16 Interaction FSA Normal Flow 87
3.17 Interaction FSA Leaving Unordered Group 88

4.1 High Level Architecture Federation Topology 99
4.2 High Level Architecture Logical Abstract Entities 108
4.3 Example STIDE Finite Automata: Length 3 110

5.1 STIDE Model Generation Convergence 121

7.1 LEAPBM ACM Finite State Automaton 151
7.2 STIDE Sequence Length 6 ACM Finite State Automata 152

A.1 Air Transport Operations Federation Object Model Hierarchy 185

LIST OF TABLES

2.1 Modelling Techniques Categorised by Use of Order 31

5.1 Comparative System and Abstract Library Call Usage 116
5.2 Precision: Observations of ACM Model Comparisons 119
5.3 STIDE ACM Model Generation Combinations 120
5.4 Precision: ACM Comparison Results 122
5.5 Precision: STIDE HLA Federate Differentiation Results 123
5.6 Precision: Average-Branching Factor Results 124

6.1 Focus: Observations of ACM Model Comparisons 137
6.2 Focus: ACM Comparison Results 138

7.1 Complexity: Detection Time of ACM Behaviour 147
7.2 Complexity: Compressed Size of ACM Models 149

ABBREVIATIONS

The list below details the abbreviations and acronyms used within this study.

PBM	Program Behaviour Modelling
LAE	Logical Abstract Entity
LEAPBM	Logical Entity Abstracted PBM
HLA	High Level Architecture
STIDE	Sequence TIme-Delay Embedding
OMG	Object Modelling Group
UML	Unified Modelling Language
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
GUI	Graphical User Interface
FSA	Finite State Automata
NFA	Non-deterministic FSA
DFA	Deterministic FSA
PDA	Push-Down Automata
EFSA	Extended FSA
AI	Artificial Intelligence
XML	Extensible Markup Language
DTD	Document Type Definition
DMSO	Defence Modelling Simulation Office
SOM	Simulation Object Model
FOM	Federation Object Model
MOM	Management Object Model
RTI	Run Time Infrastructure
LRC	Local Run-time Component
RTIEXEC	RTI Executive
DSILI	Distributed Simulation Interposition Library Infrastructure
MCC	Model Carrying Code
ATO	Air Transport Operations
ACM	Aircraft Manager

Chapter 1

INTRODUCTION

Our natural world abounds in complexity. The interactions between plant and animals in an ecosystem, the behaviour of weather systems, and the function of a human brain are all examples of systems whose behaviours can be considered complex. In addition to these natural systems, an increasing number of man-made systems continue to grow in complexity. Examples of these include: the dynamics of vehicles' movements (the handling of an automobile or the manoeuvrability of an aeroplane), and the binary digits which encode the behaviour of computer systems.

The complexity of these systems makes them difficult to understand in their entirety, and limits development and improvement. Models are created for a range of reasons: to aid understanding, to make predictions, and to facilitate analysis and development. Models encapsulate complex details and provide a more abstract, more easily understood, and more easily manipulated representation of a system's behaviour. For example, weather models encapuslate highly complex heat transfer amongst the atmosphere, ocean and sea-ice and provide weather forecasts.

This study specifically concerns the modelling of complex computer software systems. As a result of increasing processing power and storage space, the size and complexity of programs which can be executed by a computer system is growing. This complexity inhibits a human's understanding of computer software requirements, behaviour, and impact on a computer system. It is advantageous to increase human understanding of complex computer programs for two reasons:

1. Simpler communication and collaboration between software developers during **development**.

2. Simpler decision-making about the **security** impact of a computer program.

Development models assist software creators in producing more correct software in less time. Development models are abstract and detail the internal interactions between program components.

They are used as a guide during a program's development. These models are created *before* the program is developed, and do not include representations of program's interactions with the operating system environment. They are therefore of little use in computer security applications.

Models of program behaviour for computer security aim to identify, and control, damaging behaviour of computer software. Computer security models often focus on a program's interactions with the computer operating system, and are used to simplify the decision-making process concerning a program's security impact. These models are created *after* a program's development, and focus on a program's external interactions. They are commonly too complex to be easily understood by a human, as they often maintain a large number of such interactions.

Computer security models are sometimes simplified, and their performance degraded, to meet the performance requirements of real-time applications. This simplification is achieved by discarding the details of each program interaction. This exclusion of details, limits the accuracy with which program behaviour can be represented by such models. The limited accuracy of existing computer security program behaviour models, is the driving motivation of this study.

1.1 Goals

This study is concerned with program behaviour modelling (PBM) applied to computer security. The goal of this study is *to investigate PBM using flexible abstract basis-terms to represent the logical concepts understood by a program.* The expected benefits are:

Improved accuracy : A more accurate way of modelling program behaviour can improve computer security. Ensuring security of computer systems is an important requirement in today's environment of costly computer security attacks; over the past 4 years the cost of computer abuse to an organizations has averaged US$425,325 a year (Richardson, 2003; L. A. Gordon and Richardson, 2004, 2005, 2006).

Improved human comprehensibility : Human comprehension of a program's behaviour has the potential to allow users to verify the behaviour of computer software they use and develop, and augment existing computer security processes. Models of complex systems, be they programs or raw data, often have poor human comprehensibility (R. Santos and Freitas, 2000).

The success and improvement of these benefits depends greatly on whether usable complexity is maintained. Thus, ensuring acceptable complexity and usable performance are also crucial goals of this study.

Finally, this study also aims to investigate the applicability of flexibly-abstracted PBM to a range of software not previously examined. Previous research focuses on either privileged server software such as those responsible for delivering email, or mobile programs such as those employed on the World Wide Web. While these programs are important and commonly used, they are not representative of all complex computer programs.

1.2 Approach

This study approaches these goals with the proposition of flexible abstraction. This approach allows the flexible identification of the program interactions used to model behaviour, where previous efforts utilise set interaction groups only. The chosen interactions are tailored to represent the external logical abstract concepts employed by a program. It is proposed that these concepts can be represented by a program's interactions with external software libraries.

The program interactions understood by a model are the terms used to define a program's behaviour. The flexible identification of these interactions is to provide a reduction in the number of interactions a model must represent to capture program behaviour. This reduction will logically allow more detailed representation of each interaction, which is to enable more accurate comparisons concerning the security impact of program behaviour. It is also expected that a model consisting of a smaller number of more detailed interactions will be more easily understood by a human, and of lower complexity. The goals of maintained performance and complexity are expected to be fulfilled by this reduced number of interactions, and the performance improvements that follow.

The final goal of this study, to apply PBM to new types of programs, can be achieved through the choice of an appropriate investigation test domain. This goal requires that the selection of the test domain is neither mobile code, nor privileged server processes (historically the typical PBM domain (Ko, 1996; Sekar et al., 1998; Wagner and Dean, 2001; Eskin et al., 2001)), and is representative of non-trivial and commonly used programs. The domain of High Level Architecture (HLA) distributed simulation meets these requirements. HLA programs are appropriate for testing the proposed flexible

abstraction technique due to their suitable size and complexity, common use in industry, and utilisation of a range of system functionality. Their distributed nature also enables the controlled alteration of distinct components and simplifies the investigation.

1.3 Organisation

This study is organised to provide a logical investigation of the merits of flexible identification of logical abstract concepts for the basis of program behaviour model definitions. Chapter 2 provides a critical review of the field of program behaviour modelling. This chapter is divided into four sections: applications, basis-terms, definition techniques, and performance. These sections cover the different aspects and factors concerning PBM techniques.

Chapter 3 details a technique for the flexible, abstract identification of a model's basis-terms, called Logical Entity Abstracted Program Behaviour Modelling (LEAPBM). This chapter details the specification and identification of the logical abstract entities which reflect the concepts used by a program, how they are used to define a program's behaviour model, and two different formats for this model definition. Additionally this chapter defines the requirements for a security application of LEAPBM, such as an anomaly-prevention system.

Chapter 4 outlines the investigations into whether the proposed LEAPBM approach achieves the expected benefits and goals: improved accuracy, human comprehensibility and maintained complexity. This chapter details the chosen specifics of these investigations: the domain within which they are performed, and the specific subject systems they concern. The chosen domain is High Level Architecture distributed simulation and the subject systems are the proposed LEAPBM and the existing Sequence TIme-Delay Embedding (STIDE) PBM techniques.

Chapters 5, 6 and 7 detail the individual investigations into, respectively, the relative precision accuracy, focus accuracy, and complexity of the two investigation subjects.

Chapter 8 concludes this study and reviews the degree to which the goals are achieved, the implications of these results, and the potential future directions for research on the subject of flexible and abstracted PBM.

1.4 Outcomes

The demonstrations for the investigations in this study, show positive results for the LEAPBM technique when compared with the existing STIDE technique. The use of flexible abstraction allows the LEAPBM technique to model HLA federate program behaviour that more precisely identifies anomalous behaviour, and correctly equates distinct program implementations of functionally equivalent behaviours. These two results indicate improved accuracy of the LEAPBM technique and fulfil the first goal if the study.

The investigation into the relative complexities showed that, compared with the STIDE technique, the LEAPBM technique produces relatively small and comprehensible models of HLA federate behaviour. The LEAPBM technique also provides substantially smaller diagrammatic models of the same program's behaviour. This investigation also demonstrates that both the LEAPBM and STIDE techniques require little storage space.

The flexible identification of abstract concepts for the basis of program behaviour models has produced promising results in this investigation. Future directions for ongoing research are described in section 8.3.

Chapter 2

BACKGROUND: PROGRAM BEHAVIOUR MODELLING

Models play a critical role in facilitating investigations and on-going development of complex systems. Complex systems are often difficult or impractical to investigate and manipulate in reality. Representative models are created to overcome these difficulties. For example, traffic flow models allow civic engineers to investigate different configurations of cities' motorways without expensive real-life trial-and-error.

Behaviour models are designed to encapsulate and abstract the complexities of the systems they represent and provide a more simple representation of behaviour. They are commonly defined as a state machine, which distinguishes between the different internal configurations in which a system can exist. The basic premise of a state machine behaviour model is that there exist a set of interactions made by the system that change its logical state. A state refers to a unique internal configuration during a system's behaviour. The interactions that change a system's state are termed the model's *basis-terms*, and can identify operations with other internal components of the system itself or with external components.

State machine system behaviour models associate basis-term interactions with states. A system is considered to exist in exactly one of a set of defined states. Each state maintains which basis-term interactions can occur, and the effect of each such interaction on the system's state. To illustrate these concepts, consider a state machine which models the behaviour of a car. The basis-terms chosen for this state machine are *speed* and *acceleration*. These basis-terms are internal as they represent attributes of the car itself. Three states: *stopped*, *accelerating* and *cruise* can be defined by values of these basis-terms and linked by transitions to produce the state machine behaviour model shown in Figure 2.1.

The remainder of this chapter covers behaviour modelling of computer programs: its history and applications, the terms upon which it is based, the techniques used for its definitions, and how its performance is measured and compared.

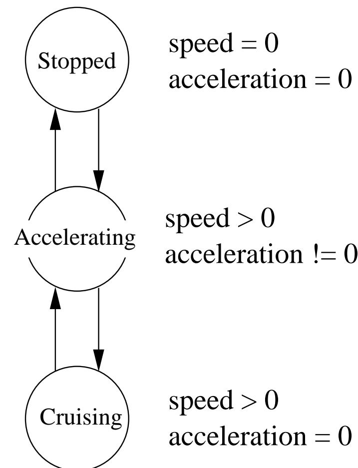

Figure 2.1: Car State Machine

2.1 History and Applications

The earliest program behaviour models were used in the development of computer programs. Applications of PBM have occurred after the emergence of computers capable of executing complex software. This application of PBM is still pervasive today. Program behaviour models are used to assist developers to specify and plan the functionality of a computer program, before it becomes a complex implementation.

The techniques used to define these program behaviour plans have evolved with the programming languages used to implement them. Older procedural type programming languages, such as BASIC, required little more than simple pseudo-code descriptions (Seefeld, 2002), or simple diagrams such as decision trees (Mind Tools, 2006), data flow diagrams (Bruza and Van der Weide, 1989) and flow charts (Bennett, 2004). Program behaviour models for software development are designed primarily for use by humans, and consequently are often abstract and generic.

The goal of software development is to create computer programs to fulfil tasks. This goal does not take into consideration the possibility of computer programs being misused to perform tasks other than what they were designed for. The ability to misuse a computer program, and perform functionality other than intended, was not initially important to software developers working on single-user

computer systems.

The advent of multi-user computer systems however, quickly led to the identification of computer security as an important requirement (Anderson, 1972). Computer security seeks to control which different functionality and information of a computer system is available to different users. As computer security matured, concepts such as authentication (to confirm a user's claimed identify) and access control (to enforce rules about the programs and data available to users) were implemented in computer systems to which multiple users had access (Whitmore et al., 1974).

The misuse of computer programs to perform extra and uncontrolled functionality, was soon identified as a means to circumvent traditional computer security access control (Foster et al., 2005; Casad, 2004; Grover, 2003). Misuse continues to be a common security attack made against computer systems (SANS-Institute, 2005).

The second pervasive application of PBM is within the field of computer security, and seeks to (amongst other things) minimise this misuse of computer software. This PBM application is termed *intrusion detection*, and seeks to identify and recognise software misuse. Intrusion detection was originally proposed by Denning (1987) to identify intrusions, and is based on the hypothesis that they result in abnormalities in system usage. PBM is used in intrusion detection to enable the comparison of occurring behaviour with normal behaviour, to identify anomalies which can signify security attacks.

The software development and computer security domains, and their different uses of PBM, are further reviewed in the following sections.

2.1.1 Software Development

Software development is constantly developing to reflect the functionality of new and emerging programming concepts. Software development can be generally described by a number of stages (Christerson et al., 1992; Henderson Sellers and Edwards, 1990; Raccoon, 1997):

Analysis : Examine the requirements for the software to be built. This includes analysing *what* functions it must perform.

Design : Define the specifications for the software system that will fulfil the analysed requirements. This specification identifies *how* functions will be performed.

Construction : Build the software system based on the defined software specifications. Consideration should still be given to ensuring the analysed requirements are fulfiled.

Testing : Verify through practical use, and theoretical examination, the constructed software correctly implements the designed specifications, and fulfils the analysed requirements.

PBM for software development has evolved in parallel with the processes and languages that are used in the construction phase. Simple models using pseudo-code and flow charts were sufficient for equally simple procedural programming languages. However, the advent and continued growth of object-oriented and other high-level programming languages (Henry, 2001; Nasir, 1996), requires more powerful ways for creating planning models for program behaviour. The industry standard for planning object-oriented programs is called the Unified Modelling Language (UML) (Siegel, 2005; Object Management Group, 2005). The UML standard is designed to enable the planning of software constructed with modern, object-oriented programming languages, such as C++ (Stroustrup, 1986).

While the planning (analysis and design) phases of software development utilises PBM heavily, PBM is also employed during the verification (testing) phase. Both these applications of PBM are discussed in the following sections.

Planning: Analysis and Design

Program behaviour models are employed during the planning stages (analysis and design) of software development to enable developers to better understand and tailor the functionality of a system to its requirements. Models facilitate this task by abstracting the specific programming language syntax that will be used to define the program, and simplifying the developer's view of the program.

The UML is the standard industry technique for planning object-oriented programs, such as those written in C++[1]. The UML utilises numerous diagrams to define a model for planned computer software. These diagrams represent the interactions of various parts of the planned software:

- With each other (Martin, 1998c, 1997).

[1] This study is primarily concerned with C++, as it is object-oriented, widely used, and has a native interface in the chosen HLA test domain.

- With the users of the software (Martin, 1998d,b,a).

Program behaviour models for software planning, such as a set of UML diagrams, are distinct from software implementations in that they define *what* happens, but not *how* it happens. This allows the abstract software requirements to be specified, investigated and developed prior to any complex implementation. UML utilises state machines[2] heavily to represent the behaviour of the planned computer software. State machines are employed to model the dynamic nature of software components (Ambler, 2004) in several types of UML components: activity diagrams, state machine diagrams, collaboration diagrams, communication diagrams, and interaction overview diagrams.

UML does not allow flexibility in the basis-terms of its models of software component behaviour. Recall that a program behaviour model's basis-terms are the elements whose interaction with by a system represent its behaviour. The basis-terms for UML behaviour models are limited to other planned software components, and the users of the software.

Verification: Testing

Program behaviour models are also employed during the testing stage of software development to enable the speedy checking of correct program functionality. A model of the program's intended behaviour is created from its specifications or documentation. This procedure is becoming more automated, with the increase in size of modern programs making manual definitions impractical and arduous (Holzmann and Smith, 2001). The program's behaviour definition can then be checked for conformance to this model, often through analysis of its code (Bates and Wileden, 1983; Auguston, 1995; Hangal and Lam, 2002; Sterlicchi, 2004; Andrews et al., 2006).

PBM is also employed in software testing architectures that focus on the security related behaviour of a program. In this application, PBM is used to check that a program does not violate a predefined safe policy (Chen and Wagner, 2002; Ganapathy et al., 2004; Schwarz et al., 2005). This is also an example of the use of PBM for computer security applications, which is more fully discussed in the following section.

---

[2] Similar to the introductory behaviour model illustrated in Figure 2.1.

2.1.2 Computer Security

Computer security is a well established field of study with roots in the United States of America defence forces. Some of the earliest research into this area was conducted by the Electronic Systems Division of the Air Force Systems Command in the 1970s (Schell et al., 1973; Karger and Schell, 1974; Whitmore et al., 1974; Bell and La Padula, 1975). In one of the earliest such papers, the issue of computer security is said to have originated from the emergence of complex shared computer systems and networks, and from the use of these systems by a population of users with non-uniform security clearance (Anderson, 1972). Thus, computer security spawned by the need to protect sensitive military data, which is also a guiding motivation of this study[3].

Since the 1970s, the charter for computer security has grown from "the protection of classified information on military systems", to encompass the "protection of a computer system from all hostile acts, influences or dangers" (Olovsson, 1992; Garfinkel et al., 1996). This more challenging goal now includes requirements such as the assurance of availability (ensuring a system is appropriately accessible) and integrity (ensuring a system is not manipulated). A secure computer system is one that is safe from all hostile influences that may affect its normal operation. Hostile influences are termed *attacks*, and the persons responsible termed *attackers*.

Providing computer security is a question of cost versus benefit. It is not a simple or cheap task to ensure that all aspects of a computer system including software, users, physical location and connectivity, are 100% secure. This is primarily due to two reasons:

1. The majority of existing software was written without a major emphasis on security[4]. For example, most software suppliers include the facility for future upgrades or "patching", specifically to address security issues that may arise.

2. Writing secure code is difficult and requires an in-depth understanding of the target computer system and language.

---

[3] The protection of sensitive military simulation data is possible through improved distributed simulation security, which is addressed later in this study.

[4] Recall that programs are often developed with the primary goal being the solution to a problem, and little emphasis placed on the potential for misuse.

The cost of re-writing all existing computer programs, which stretch back decades, is prohibitively high. In some highly sensitive domains however, the cost of total computer security is justifiable; for example, in the banking and defence sectors.

Approaches for ensuring computer system security, as with all forms of security, can be categorised as follows (Axelsson, 1998; Halme and Bauer, 1995):

Prevention : Hindering or completely blocking an attacker's intrusion attempt.

Preemption : Identifying a potential attacker, and removing their ability to cause harm before they have the opportunity to successfully intrude.

Deterrence : Increasing the necessary effort required for successful intrusion, increasing the perceived risk associated with intrusion, and decreasing the perceived gains.

Deflection : Convincing an attacker they have succeeded where they have not.

Detection : Distinguishing attacks and intrusions from legitimate activity and raising alarms.

Countermeasures : Automatically countering an intrusion as it is being attempted.

Of these approaches, the two most commonly employed are also the two within which PBM is used: prevention and detection (Sekar and Uppuluri, 1999; Li, 1997). The following sections discuss these two approaches to providing computer security and their use of PBM.

Intrusion Detection

Intrusion detection is performed by monitoring the activity of a computer system to identify abuse (attacker activity) after it has occurred. Intrusion detection is built upon the theory that the use of a computer system by an attacker produces detectable changes in the system audit trail, which was originally proposed by Anderson (1980). The term 'system audit trail' refers to a record of functions that are performed by the operating system for users, or programs. Some of the most influential subsequent research in this field is by Denning and Neumann (1985), during the development of the first intrusion detection system (IDS) (Denning, 1987).

Changes in the system audit trail caused by attackers are identified by intrusion detection systems in one of two ways (Axelsson, 1998):

Anomaly Detection : Observed behaviour is compared to a model of normal behaviour. Any behaviour that does not match the normal model is considered anomalous and a potential intrusion (Javitz and Valdes, 1994; Ko et al., 1994; Sekar et al., 1998).

Anomaly intrusion detection provides, in theory, the best level of protection due to the ability to detect previously unknown intrusions. However, some techniques of representing normal program behaviour focus too heavily on low-level details that often leads to behaviour which *is* normal, but is not strictly equivalent to the representation, being incorrectly identified as an intrusion. This incorrect identification is termed a *false-alarm*. Historically, anomaly IDS' main weakness is the high rate of false-alarms they produce, ofttimes betwen 50 and 80 per day, which can necessitate a prohibitively high investigative workload (Lundin and Jonsson, 1999).

Misuse Detection : A specific set of small models representing known intrusions, termed *signatures*, is compared to observed behaviour. A match indicates the presence of anomalous behaviour that is known to be indicative of an intrusion (ISS, 1999; CISCO, 1999; Paxson, 1998).

The main advantage of misuse intrusion detection is the low rate of false-alarms, which is a result of detailed signature specifications and the potential detail of the computer system audit trail. However, misuse intrusion detection has two main weaknesses:

1. Only intrusions that have been configured may potentially be detected. This means any intrusion developed in the future will never be detected. The solution to this issue means constant and time consuming updating of known intrusion signatures.
2. Identifying the potentially broad range of equivalent signatures for a particular intrusion is difficult. Thus, often a small change to an attack by an attacker results in the incorrect identification of this attack as normal behaviour (Cohen, 1998). The incorrect identification of intrusive behaviour as normal is termed a *false-acceptance*. Misuse intrusion detection is prone to suffer from a high rate of false-acceptances.

Historically, the models employed by both misuse and anomaly intrusion detection systems did not include representations of program behaviour. Instead the activities of either the network to which the computer was connected (Handley and Paxson, 2001; Cansian et al., 1997; Jou et al., 2000; Chang et al., 2001), or the user interface (Neumann and Porras, 1999; Handley and Paxson, 2001; Lane and Brodley, 1997; Teng et al., 1990), were used as the basis for their models.

More recent intrusion detection systems, however, do incorporate program behaviour as part of the behaviour models upon which they operate. The terms upon which program behaviour models for intrusion detection systems are commonly based are[5]:

- Operating system records, such as a program's processor, memory and file usage.

- Interactions with the computer operating system kernel (Axelsson, 1998)

Early work by Forrest et al. (1996) and others (Kosoresow and Hofmeyr, 1997; Forrest et al., 1997; Hofmeyr et al., 1998) identified the potential to distinguish between programs based on their interactions with the operating system kernel. These interactions occur via system library function calls (commonly termed *system-calls*). Subsequently, a variety of implemented and proposed intrusion detection systems utilise system-calls as basis-terms for the definition of program behaviour models (Kosoresow and Hofmeyr, 1997; Lee et al., 1999; Warrender et al., 1999; Ju and Vardi, 2001; Somayaji and Forrest, 2002; Eskin, 2000; Ghosh and Schwartzbard, 1999).

Misuse detection systems employ these terms only in conjunction with others. Thus, in the strictest sense they do not employ program behaviour models, but rather their models incorporate portions of program behaviour to define the more broad behaviour of a computer system.

Anomaly detection systems however, utilise program behaviour models exclusively to define the model of normal behaviour from which anomalies are identified (Ko, 1996; Somayaji and Forrest, 2002; Sekar et al., 1998; Ghosh et al., 2000; Wespi et al., 1999; Wagner and Dean, 2001; Liu, 2002; Maxion and Tan, 2002; Eskin et al., 2001). Each model refers to a specific program, in contrast with to misuse detection systems whose models more commonly cover an entire computer system.

---

[5] Refer to section 2.2 for a full discussion of PBM basis-terms.

Intrusion Prevention

Intrusion prevention has its origins in early multi-user operating systems and, along with more recent advancements, employs PBM. Intrusion prevention is performed by restricting access to computer system functions or capabilities, based on who (or what) is requesting it. By restricting and controlling access, intrusion prevention aims to stop attacks before they succeed. The earliest intrusion prevention systems were implemented as part of some of the first multi-user operating systems: Multics (Whitmore et al., 1974), and UNIX (Ritchie, 1977). These same techniques are still employed today.

Intrusion prevention is a class of functionality that includes network appliances called Intrusion Prevention Systems (IPS). In today's technology environment, IPS refers to a misuse-detection system, responsible for preventing network-based attacks from reaching a computer system.

The mechanisms that support standard intrusion prevention are *access control* and *authentication* (Ross, 1999). Access control is designed to ensure that information or computer resources are not viewed, utilised, or modified, by unauthorised users or programs. Access control requires that different users and programs can be distinguished: a task that is provided by authentication, which aims to ensure specific identities. Authentication can be achieved in a number of ways, from simple passwords (Sandhu and Samarati, 1996), to encryption keys (Freier et al., 1996), and complex biometrics (Dugelay et al., 2002). Access controls for computer system resources are specified in terms of authenticated identities, making the success of intrusion prevention dependent on correct and consistent authentication.

The increasing maturity of anomaly detection algorithms and processes (employed in intrusion detection) has more recently led to the exploration of the identification of anomalies as an intrusion prevention measure (Sekar et al., 1998; Somayaji and Forrest, 2000; Herrmann et al., 2002; Wagner and Dean, 2001). The use of anomaly detection as an access control technique is termed *anomaly prevention*.

Anomaly prevention differs from anomaly detection in the stage of program execution at which the identification of anomalous behaviour is performed. Anomaly detection, as utilised by intrusion detection systems, focuses on the impact and the results that are visible *after* a program's execution. Anomaly prevention however, examines a program's behaviour *before* and *during* its execution.

The examination of program behaviour before it is executed enables identified security attacks to be prevented from occurring. This requires that some way of examining the program in its distributed state prior to execution is available. Techniques for this have been proposed for the Java programming language (Sun Microsystems, 1998), which take advantage of several Java attributes: the Java byte-code, and the well-defined and standardised Java library interface (Necula, 1997; Necula and Lee, 1998; Sekar et al., 2001b).

However, an assurance that a program performs no attacks before it is executed does not protect against the common security attack of malicious code injection, which can occur *during* execution.

Examination of program behaviour during the program's execution, is used to enforce the model that was previously checked for security attacks. Any program behaviours that deviate from the approved model of behaviour, are denied or blocked. This run-time examination requires that program interactions with external components are intercepted for analysis before they are executed. This can be performed in several ways:

- Adding functionality to the program to perform self-checking, before interacting with external components.
- Adding functionality to the external component to check program interaction requests.
- Adding software that controls the interface or medium between the program and the requested external component.

An example of the first approach for behaviour interception is the addition of functionality to the operating system kernel and is employed for the system-call-based anomaly-prevention system *pH* (process Homeostasis) developed by Somayaji and Forrest (2000, 2002). The popularity of system-call sequences as the basis-terms for PBM definitions in anomaly-detection is also present in anomaly prevention. The pH kernel modification provides the automated real-time identification of potential anomalies in every program that is executed. This approach is limited to the operating systems for which this additional functionality is implemented.

A more widely applicable technique, called *interposition* (Jones, 1993; Curry, 1994), is an example of the second approach to behaviour interception. Interposition facilitates the transparent interception of calls made by programs on a wide range of software components, such as the operating system

kernel (Kuperman and Spafford, 1999; Jain and Sekar, 2000; Gonzalez et al., 2000) and third party software libraries (Wharington and Andrews, 2002; Andrews et al., 2002b).

PBM techniques are commonly utilised by anomaly-prevention techniques for the automated generation, checking, and enforcement of the program behaviour.

2.1.3 Summary

This section has introduced PBM from its genesis in software development planning, and discussed its current applications in this field, and that of computer security.

Program behaviour modelling is pervasively used in the planning phase of software development, and in the detection of computer security intrusions. More recent applications include the verification phase of software development, and in the prevention of computer security intrusions.

The basis-terms for PBM varies from other internal components of the program, to operating system logging application output, and direct operating system interactions. These different approaches to the definition of program behaviour are discussed in the following section.

2.2 Model Basis

Different basis-terms are employed for different PBM applications. To review, a program behaviour model's basis-terms are the set of terms representing the elements with which the program interacts. Internal basis-terms refer to software components of a program itself, while external basis-terms refer to other computer system elements with which a program interacts.

The PBM applications discussed in the previous section, commonly utilise one of three different choices in model basis-terms:

1. Internal software components.

2. Operating system audit trails.

3. Operating system library calls.

Each of these basis-term choices is discussed in the following sections. Additionally, the recent trend in PBM research towards *abstractions* of these model basis-terms is discussed.

2.2.1 Internal Software Components

Internal basis-terms for program behaviour models place the focus of a program's behaviour on interactions between its various components. This is particularly useful to assist software developers in understanding, and making decisions about, the design of a planned program.

Early forms of these internal based program behaviour models, such as flow charts and data-flow diagrams, were used in the software development planning for procedural programs. The main type of internal basis-term for procedural languages are function calls. That is, function calls are the most logical division of procedural programs. Thus, flow charts and data-flow diagrams often distinguish between program behaviour states by the name of the current function call (Raccoon, 1997).

More recent programming languages, such as C++, incorporate both functionality and data into logical components, termed *objects*. Objects provide more meaningful division of a program's components and are the main type of internal basis-term used in models for planning object-oriented programs. Examples of this include UML sequence and collaboration diagrams (Martin, 1998c, 1997), which describe the dynamic behaviour of a program in terms of interactions between objects.

Beyond the realms of software development, the usefulness of program behaviour models that utilise internal basis-terms is limited. For example, the internal behaviours of a program are meaningless concerning the security of a computer system, as they do not provide information about the wider effect of a program.

2.2.2 Operating System Audit Trails

Audit trails have been utilised for defining the behaviour of computer system programs since the paper on intrusion detection by Anderson (1980). *Audit trail* is a relatively loose term that can be used to describe the results of program interactions on objects within a computer system, such as files, other programs and peripheral hardware devices (Denning, 1987). While many different proposals concerning the use of audit-trail data in PBM exist (Axelsson, 1998; Lunt, 1988; Wetmore, 1993), the definition of an audit-trail object is commonly: "an element controlled by the computer's operating system". The US Department of Defence Trusted Computer System Evaluation Criteria (Department of Defence, 1985) defines auditting as a requirement of *C2* or higher classified systems. The audit-trail data required for C2 systems includes operating system objects, such as files, programs and

authentication events.

Thus, the interactions used to define audit-trail-based program behaviour models, are those that can occur between a program and the operating system. The process of collecting audit-trail events is termed *logging* (Wetmore, 1993; Bishop, 1989). Logically, a program which performs this function is termed a *logger*. Loggers collect the audit-trail events (operating system interactions) which occur as a result of a program's behaviour during its execution.

The interactions between a program and the operating system are examined in one of two ways. The first technique examines the audit information provided by the operating system for each audit object after an interaction by a program. This is illustrated in Figure 2.2. This audit information typically concerns how an object was used by a program. The computer processor, input and output devices, memory and files are considered audit objects (Anderson et al., 1995; Wetmore, 1993; Endler, 1998).

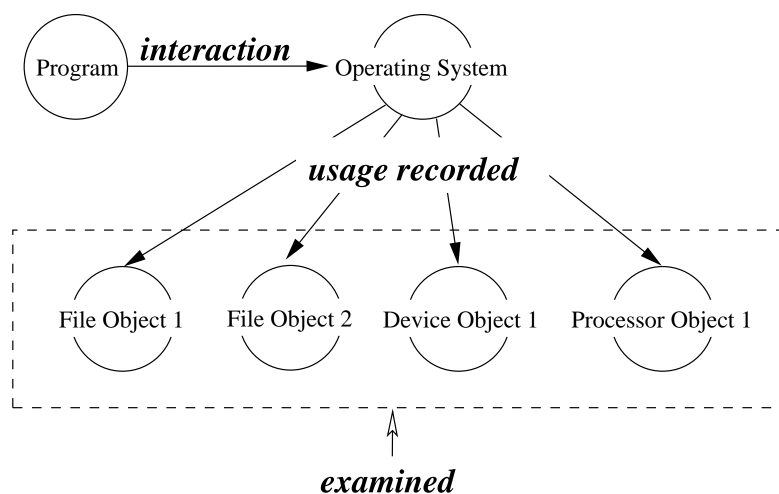

Figure 2.2: First Technique for Audit-Trail Examination

The second technique directly examines a program's interactions with the operating system software. The operating system software is termed the *kernel* (Webopedia, 2003). This technique is illustrated in Figure 2.3.

While these two techniques provide essentially the same information, the first technique emphasises the audit object's data, whereas the second technique emphasises the interaction with the audit

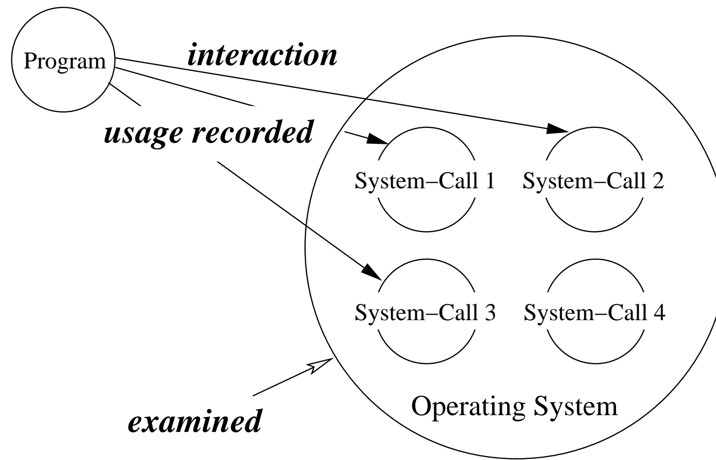

Figure 2.3: Second Technique for Audit-Trail Examination

object. That is, the first technique might describe a program behaviour by saying "*file X* had 4 bytes read by the program A", while the second technique may say "the program A *read 4 bytes* from the file X". The same information is provided in both cases, but the organisation and emphasis is different.

Examples of the first technique commonly employ logging software to monitor the operating system objects for interactions by a program, while examples of the second technique directly monitor the operating system for interactions. The interactions monitored by the second technique are commonly those that occur with the kernel software via system library function calls, termed *system-calls*. The use of system-calls as PBM basis-terms is further discussed in the following section.

The first technique for PBM, utilising audit-trail basis-terms, has several limitations.

- The audit trails for this technique are captured by monitoring operating system objects, which requires additional system resources to execute the logger software.

- The large number of potential objects for an operating system[6] means that practically, a smaller subset must be selected (Lane and Brodley, 1998).

Discarding a significant number of these operating system objects, to reduce the amount of audit

[6] For example, a typical Linux operating system, such as that utilised in this study, has over *9500* objects representing the operating system's processor and input and output devices and over *430,000* fi le objects.

data, increases the likelihood that potentially important program behaviour is overlooked (Lunt, 1993). For example, if a program's behaviour is best defined by its interactions with a file 'Y', and this file object is discarded, it will not form part of the program's behaviour model using this audit-trail-based technique. The oversight of this file then reduces the model's accurate representation of program behaviour.

Even with the reduction in objects for which program interactions are monitored, the amount of data generated is still large: on a 1987 computer system, up to 135 megabytes per user per day can be generated day (NCSC, 1988; Picciotto, 1987; Wetmore, 1993; Lunt, 1993). The large amount of audit data necessitates substantial processing for the model for three reasons:

1. It is difficult to identify a single program's interactions amongst all other programs on the computer system.

2. A program's uses must be collected and often further processed to ensure correct ordering.

3. Duplicate entries for individual program interactions must be both identified and removed (Wetmore, 1993).

The large amount of audit data is composed of representations of numerous program interactions with the operating system. Each individual interaction has little meaning, which complicates the task of understanding such a model (Lunt, 1993).

These limitations are a result of this technique's focus on the potentially large number of operating system objects with which programs interact, and the resulting large number of interactions which must be represented.

2.2.3 Operating System Calls

The interactions between a program and the core operating system software (kernel) are achieved via function calls to the system library. This specialised type of audit-trail information is commonly employed as basis-terms for PBM.

The use of system-calls as basis-terms for the definition of PBM was initially proposed by Kuhn (1986), who asserted that the capture of audit-trails for computer security applications should occur

at as low a level as possible to minimise the ease with which auditing can be bypassed by an attacker. However, his proposal did not address the details of how system-call audit trails might be utilised as basis-terms for PBM.

The work which addressed some of these details was undertaken by Forrest et al. (1996). Their proposal, in support of Kuhn (1986), asserts that the important behaviour of programs is reflected by their use of system-calls. This assertion is logical as the operating system provides the interface for programs to all important aspects of a computer system—files, memory, input and output devices—are only accessible via the operating system (Hofmeyr et al., 1998).

This assertion by Forrest et al. (1996) logically requires the examination of a program's use of system-calls, which has been performed by two different methods. The first is to employ tools (either existing or customised operating system interface software) to record these interactions as they occur *during* the execution of a program (Forrest et al., 1996; Kosoresow and Hofmeyr, 1997; Lee et al., 1997; Debar et al., 1998; Sekar et al., 1999; Ghosh and Schwartzbard, 1999). This is termed *run-time analysis*, and can be performed by existing tools such as:

- Sun's Basic Security Module for both SunOS and Solaris(Sun Microsystems, 2000).
- Berkley Software Distribution UNIX's *syscall()* function (Picciotto, 1987).
- The *strace* software (Sladkey, 1995), originally written for SunOS, available for SunOS and Solaris (Mauro and McDougall, 2001; Watters, 2002), Linux (Siever et al., 2005), System V Release 4.0 (Sobell, 1994), and Irix (SGI, 1998).

The second method to examine a program's use of system-calls is by investigation of the program source code *before* it is executed (Wagner and Dean, 2001; Giffin et al., 2002; Murthy, 2003; Sekar et al., 2001b). This is termed *static analysis*, and is performed on the program's definition as either source code, or binary machine code. Wagner and Dean (2001) propose the examination of program source code definitions using simple parsing procedures[7], while Giffin et al. (2002) and Murthy (2003) employ existing software for examining and editting binary programs[8].

---

[7] The procedure for source code parsing and examination are not unlike those employed by programming language compilers (Wikipedia, 2006b).

[8] Executable Editing Library (EEL) (Larus and Schnarr, 1995), and its Linux equivalent, Linux EEL (LEEL) (Xun, 1999).

Using system-calls as the basis-terms for PBM offers some advantages over other operating system audit-trail techniques (refer to section 2.2.2). The number of distinct system-calls with which a program can interact is much lower than other audit-trail objects. For example, typical UNIX operating systems have approximately 280 system calls, the majority of which they share in common. This represents a substantial reduction from the number of devices, files and processes on a computer system. A typical Linux operating system understands 289 system calls but can have well over 440,000 general audit-trail objects.

The smaller number of audit-trail objects, and their well-defined interface through numerous system documentations and manuals (Linux, 2002; Sun Microsystems, 1999), means the entire set of system-calls can be analysed for program interactions during model generation. This removes the possibility which exists using more generic audit-trail objects: an object that is important to a program's behaviour is not examined. Additionally, the processing required to examine this smaller number of objects and to store a program's interactions is also reduced.

The use of system-calls as the basis-terms for PBM does however have limitations. System-calls are still a form of operating system interaction, each of which individually has very little meaning (Lunt, 1993), and the groups that occur for single application-level actions tend to obfuscate the program's actual intention (Picciotto, 1987).

The recommendation by Kuhn (1986), that PBM audit-trail basis-terms should focus on low-level operating system services, has been used to justify system-call basis-terms as the only alternative to much higher-level command line basis-terms. However, the software libraries employed by a program can also be logically considered as operating system services, similarly to the lower-level system library. Software libraries are portions of the underlying operating system environment, and while they can be aliased and manipulated, it is much more difficult to do so than it is with command line elements.

It is possible to guarantee the identity, integrity and correctness of software libraries via a number of methods:

- Additional PBM to capture and restrict the program behaviour defined by a software library (Giffin et al., 2005).

- Comparing computed hash values that uniquely represent a program's code, termed *checksums*

(Horne et al., 2001).

Run-time Analysis

The two methods of examining program's system-call interactions (run-time analysis and static analysis) have different advantages and disadvantages. Run-time analysis captures all system-calls made either directly by the program, or by other software (such as libraries) called by the program. The concept of direct and indirect system-call usage is illustrated in Figure 2.4, which shows both a direct call by the program to the kernel's *write()* function, and two system-calls, *open()* and *read()*, made indirectly by the program via a software library.

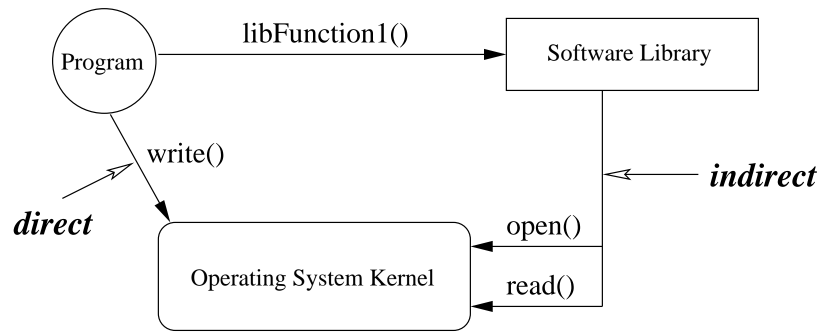

Figure 2.4: Direct and Indirect Program System-call Usage

Both the direct, and indirect interactions illustrated in Figure 2.4 are captured by the run-time analysis of a program. This is an advantage in that no system-calls which stem from a program are overlooked in its analysis.

The primary disadvantage of run-time analysis of programs is that a single execution of the program is not guaranteed to cover all the potential paths of execution by the program. Programs often have multiple execution paths, each of which constitutes potentially distinct program behaviour. Performing run-time analysis requires as many executions of distinct execution paths being examined as possible. This is often termed *training* a behaviour model.

Complete and accurate training of program behaviour models potentially requires significant effort, such as the alteration of any variables used to determine execution paths. These variables can

include:

The range of values gathered during interactions with other components : such as configuration files, input data files, user input, and external software inputs.

The execution periods during which they can occur : can change depending on network speed, users' interactions, and computer processor utilisation by other programs.

In some cases these variables are difficult to control which increases the training effort required. For example, when the timing and values of interactions is dependent on uncontrollable network speed and distributed processes[9].

An interesting proposal by Debar et al. (1998) suggests using the executions of the small functional-verification tests provided by a software developer, as the specification for a program's behaviour. This proposal has the potential to simplify and improve the training of run-time analysed program behaviour models, however it assumes the existence of thorough functional verification tests.

Static Analysis

The second method for examining a program's system-call interactions, static analysis, contrastingly requires no control or manipulation of variables which influence the program's execution path. Static analysis focuses on the definition of the program, either in its human readable source code form, or as binary machine code. Static analysis of a program's definition can capture every potential execution path it contains (Wagner and Dean, 2001), and therefore avoid the training limitations of run-time analysis.

Statically analysed program behaviour models do have several limitations, caused by program variables which are indeterminate until run-time. These variables can affect a program's behaviour, but do not have specific values in its definition. For example, operating system environment variables (Giffin et al., 2005). Other run-time dependent variables which make the static analysis of programs difficult are the software libraries utilised. These software libraries are most commonly external to the program's definition, and can potentially be changed at run-time for different executions via dynamic linking (Wagner and Dean, 2001; Giffin et al., 2002; Murthy, 2003).

---

[9] This scenario exists in the domain investigated in this study. Refer to Chapter 4 for more details.

Referring back to the example illustrated in Figure 2.4, the examination performed by static analysis is limited to that shown in Figure 2.5.

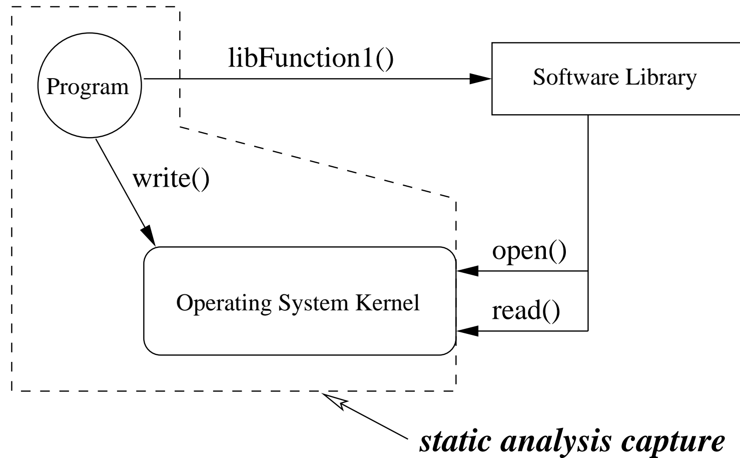

Figure 2.5: System-call Usage Captured via Static Analysis

To solve this deficiency, static analysis proposals in current research suggest either the inclusion of complementary run-time analysis tools, or the modelling of every library utilised by a program (Wagner and Dean, 2001; Murthy, 2003; Giffin et al., 2005). Both these proposed solutions complicate any system which seeks to employ statically generated system-call-based program behaviour models[10]:

- Modelling every library utilised by a program potentially increases the model size and generating processing required by a substantial amount. This increase is all prior to the run-time of a program.

  Modelling each library also requires that the individual models for the single program are combined to provide a seamless representation of behaviour. This increases the complexity of any application.

- Run-time analysis tools examine the execution a program to determine the values of variables which cannot be determined by a static analysis system. This increases the run-time processing required.

[10] For example, the application to anomaly-prevention systems for computer security.

Another solution recently proposed by Giffin et al. (2005) simulates the linking for programs that are distributed with the software libraries to which they will be dynamically linked at run-time. This solution is limited in its applicability and begs the question: If a program is going to be linked at run-time with a particular software library (indeed, a specific version thereof), why is it not just statically linked to avoid ambiguity and the potential for errors during the dynamic linking process?

Arguments

Much previous research on the use of system-calls as PBM basis-terms focuses on the approach proposed by Forrest et al. (1996) called Sequence TIme-Delay Embedding (STIDE). This technique utilises multiple, fixed-length sequences of system-calls to model program behaviour (refer to section 2.3.1 for more details). Importantly however, all the arguments supplied to each system call during a program's interaction are discarded and play no part in the behaviour model. This severely limits the accuracy of such program behaviour models (Wagner and Dean, 2001; Wagner and Soto, 2002; Kruegel et al., 2003; Sekar et al., 2001b). For example, if the arguments to a *unlink* system-call are ignored, STIDE is unable to distinguish between the removal of a temporary file, or a system configuration file from the file system.

This limitation has been addressed in a several ways. Kruegel et al. (2003) have applied an existing statistical technique for modelling program behaviour to the task of modelling the different arguments which occur to system-calls. Interestingly, this approach focuses *solely* on the arguments to system-calls and does not mirror the STIDE technique which represents the order and sequencing of system-calls.

The proposal by Kruegel et al. (2003) utilises data (system-call arguments) as the basis-terms for the definition of a program behaviour model. This research sacrifices the '0 false-alarm rate', common to statically generated models which represent program behaviours (Wagner and Dean, 2001), for improvements to precision. False-alarms are often considered the most significant weakness of behaviour modelling applications (Lundin and Jonsson, 1999).

The limitation from discarding system-call arguments is also addressed in other research by a different approach, which includes the arguments to system-calls in the definition of existing statically analysed program behaviour models (Jain and Sekar, 2000; Giffin et al., 2002). This approach

has been reported to show an increase in model precision, while still maintaining the desirable '0 false-alarm rate' (Wagner and Dean, 2001; Giffin et al., 2004, 2005). Various approaches to maintaining argument values can be found in prior research. These approaches have differing levels of complexity. Initial research allowed for only static values, that is, only arguments with constant values (Wagner and Dean, 2001; Sekar et al., 2001b). For example, both arguments in the system call *open("/var/lib/syslog", O_RDONLY);* would be maintained by this method; but only the second argument to *open(filenameString, O_RDONLY);* would be maintained as the first is dynamically set *during* the execution of the call and depends on the value of the variable *filenameString*.

Recent and ongoing research by Giffin et al. (2002) proposes data-flow diagrams and algorithmic algebra to represent changes that occur to the variables used in system-call arguments (Giffin et al., 2002, 2004, 2005). Using this technique the value of each variable can be determined at the point in a program that it is used. This technique has parallels with reverse engineering as internal program behaviours are determined from its binary form (Weiser, 1981; Beck and Eichmann, 1993).

The inclusion of arguments to system-calls has been reported to enable the most precise forms of program behaviour models to date. However the effort required to perform this is considerable and can result in "increased size of program models" (Giffin et al., 2005) and could potentially significantly impact the performance of systems which employ it. This remains an open question as the technique has only been tested for trivial programs.

2.2.4 Abstraction

The term abstraction refers to generalising some existing basis-terms leaving formerly concrete details undefined (Wikipedia, 2006a). The concrete details which are lost during the abstraction of PBM basis-terms are ideally those which represent unimportant program behaviour details. For example, consider the C code in Figure 2.6. The system-calls in this program write the string constant "hello there." to a file, then read it back and assign it to a character buffer. In this example, the specific file which is utilised in the bold statements is irrelevant as long as it is shared. A more abstract view of these system-calls can leave the first arguments to the *open* calls undefined.

Abstraction is a technique for removing the inconsequential information from program behaviour

```
myfileId = open (''tmpdata'', O_WRONLY);
write(myfileId, ``hello there'', 11);
close(myfileId);

afileId = open(``tmpdata'', O_RDONLY);
read(afileId, &buf, 11);
close(afileId);
```

Figure 2.6: Abstraction Example: C Code

models, and has been developed to improve the accurate representation of the behaviour of programs[11] for use in both software development, and computer security.

As previously discussed[12] limitations in PBM with audit-trail basis-terms (both generic audit-trail objects and system-calls) are caused by: the large number of interactions between the program and auditted operating system objects, the large volume of information represented by these detailed interactions, and the little meaning attached to each (Lunt, 1993). The necessity of large numbers of program interactions with audit-trail basis-terms to represent meaningful program behaviour highlights the potential for abstraction improvements (Neumann and Porras, 1999).

The initial development of abstracted PBM by Bartussek and Parnas (1977) was called the abstract trace method of program specification, and was developed for the field of software development. The abstract trace method allows the definition of software's behaviour (*what* it does), without necessitating an algorithm for *how* it occurs (Bartussek and Parnas, 1977; Heitmeyer and McLean, 1983; McLean, 1984). This trace method abstracts from the specification all the elements from particular language for presenting algorithms (such as a programming language), thus making the specification easier to read and removing misunderstandings that result from the "attempted gleaning of essential features from a mass of extraneous details" (McLean, 1984).

The same abstractions are also present in the UML object-oriented design technique (Siegel, 2005; Object Management Group, 2005). The UML concentrates on defining the abstract interactions between internal software components, but does not require the definition of the algorithms which im-

[11] Refer to section 2.4 for a more detailed discussion on PBM performance.

[12] Refer to section 2.2.2

plement these interactions.

The use of abstraction of PBM basis-terms is not limited to the field of software development. A substantial amount of research has also been done in the field of intrusion detection, more specifically: misuse detection. This research defines the abstraction of audit-trail details into higher level concepts to aid in the correct focus of misuse detection program behaviour models on interactions which constitute an attack, as opposed to individual audit-trail details (Lin et al., 1998). This abstraction has been shown to reduce the number of false-acceptance errors made by a misuse detection system.

These abstract misuse signatures are defined in terms of abstract *events*, which are themselves defined in terms of audit-trail basis-terms. The onus for determining if an audit-trail interaction constitutes an event is placed on the human developer of the event, typically a system security officer (Lin et al., 1998; Lin J., 1998; Ning et al., 2001). This required manual definition complicates any system which employs such abstract misuse signatures.

Abstractions potentially encapsulate the low-level differences between multiple program interactions into more generalised and powerful events. For example, Sekar et al. (1999) reiterate a point made by Wetmore (1993), that it is cumbersome to represent an event worth analyzing as low-level system-calls, and furthermore, that different levels of abstraction may be desirable in different contexts. This observation is a key motivation for the flexible abstraction technique proposed in this study.

2.2.5 Summary

This section detailed the different terms upon which program behaviour models are based. The basis-terms of a PBM represent the elements of a computer system with which a program is considered to interact. Proposals exist in previous research for several different types of basis-terms for PBM:

- Internal terms.
- Operating system audit-trial objects (files, memory, hardware devices etc.).
- Operating system kernel function calls (commonly termed system-calls).
- Abstractions of the above types.

The use of internal basis-terms provides an isolated view of a program's behaviour, with no information concerning its interaction with other computer system components.

Operating system basis-terms, both audit-trail objects and system-calls, typically result in large volumes of information from which it is difficult to deduce a program's actual intent (Wetmore, 1993; Lunt, 1993). This reduces the usefulness of such models whose purpose is to improve understanding and decision making.

The option of abstractions has, for computer security applications, most commonly been applied to operating system basis-terms. These abstractions have facilitated definitions in terms of higher-level concepts, though these concepts still require association and definition in terms of low-level terms.

A point made by Wetmore (1993), and adopted in this study, notes that the use of application-level audit data (and the implicit abstractions it contains), can both "drastically reduce the volume of audit-data, and make it easier to comprehend intentions". The use of application-level audit data is enabled by interposition and illustrated by Kuperman and Spafford (1999) who utilise interposition to successfully gain access to application-level audit events (software library function calls). These software library calls provide the application-level equivalent of operating system level system-calls, and potentially, ready-made abstract basis-terms for use in PBM.

For each basis for PBM discussed in this section, numerous different techniques have been developed for representation. While different techniques offer different characteristics, it is possible that the choice of a model's basis-terms has more of an influence on its performance than other choices (Warrender et al., 1999). The following section provides a review of the most commonly employed techniques for representing program behaviour models.

2.3 Definition Techniques

Different techniques are employed to define models of specific programs' behaviour. A modelling technique determines how individual program interactions with basis-terms are organised to provide a representation of program behaviour. As discussed in the previous section, basis-terms identify the program interactions that are available to a model.

There are too many types of PBM techniques proposed in previous research to attempt to review them all. Therefore, this section is split into categories based on these proposed techniques' inclusion

of ordering information in its model representations. Program behaviour model's ordering information refers to information about the sequencing and order of program interactions. Within each of the categories reflecting a PBM's inclusion of order, similar proposals are grouped together and an overview of each group provided.

The groups of PBM techniques, ranging from order inclusive through to order exclusive, are shown in Table 2.1. Each of these model representation techniques is discussed in the following sections.

Order Category	**Modelling Techniques**
Inclusive	Finite Automata, Formal Grammars
Exclusive	Statistics, Artificial Intelligence
Partially Inclusive	Specialised Finite Automata

Table 2.1: Modelling Techniques Categorised by Use of Order

2.3.1 Order Inclusive

The inclusion of ordering interaction information in PBM representation techniques is common throughout previous research. Order inclusive PBM techniques maintain distinctions between the different points a program's execution can exist in. That is, it reflects the different steps in the program's execution path, each of which is termed a *state*. The order and sequences in which program interactions occur can then be represented by creating links between states, termed *transitions*.

Representing the order of program interactions then requires the identification of program execution states and transitions and their replication in the model. Primarily, two distinct types of PBM techniques include the order of program behaviour in their representations: finite state automaton and formal grammar. These two techniques are discussed in the sections below.

Finite Automata

Finite automata are defined by states and transitions and hence provide a natural technique for representing behaviour as state machines (Kent et al., 1991). They are also termed Finite State Automata (FSA, used in this study) and Finite State Machines. FSA are one of the most commonly employed techniques for representing order inclusive program behaviour within both computer security (Kosore-

sow and Hofmeyr, 1997; Sekar and Uppuluri, 1999; Ghosh et al., 1999; Wagner, 2000; Sekar et al., 2001b,a; Wagner and Dean, 2001; Sekar et al., 2003; Feng et al., 2004; Giffin et al., 2004, 2005) and software development (Martin, 1998b,a).

In a FSA model each state is represented by a circle, and each transition is represented as an arrow linking two states. States and transitions are also termed *nodes* and *edges* respectively. States at which the automaton ends are represented by two concentric circles. FSA states (nodes) represent points in a program's execution within which no basis-term interactions occur. FSA transitions (edges) represent the basis-term interactions which a program performs, and how the program state changes as a result.

FSA can be divided into two categories: *deterministic*, and *non-deterministic*. A FSA is considered deterministic if for each state there exists one, and only one, transition leading to a subsequent state. Deterministic FSA (DFA) are not well suited to representing the behaviour of a program, which almost always includes branching, looping, or recursion.

Employing a non-deterministic FSA (NFA), where a state can have multiple transitions, is also imprecise in its representation of potential program behaviour: there can exist paths through a NFA that are impossible in the program it represents. Giffin et al. (2002) provide an example which illustrates the problem of impossible paths. Consider the small C program shown in Figure 2.7.

The NFA representing this program's behaviour includes additional states for the start and end of internal program functions and is shown in Figure 2.8.

The statements in bold from Figure 2.7 are those which constitute basis-terms for this example. These statements identify the program interactions which are represented as transitions in the NFA. The transitions to and from internal program function start and end states are labelled ϵ to indicate that they represent no program interaction with basis-terms. For readability, Figure 2.8 shows the grouping of program states in terms of local program functions. In this example a possible path exists in the NFA which does not exist in the actual program source code. This impossible path is illustrated in Figure 2.9.

Impossible paths arise because NFA do not record a history of automaton transitions, in contrast with a program whose use of functions follows a stack structure. Thus, in a program it is impossible to return from a function that was not called. However, in a NFA nothing is known about previous transitions, so no equivalent restriction can be imposed.

```
main (int argc, char ** argv) {
   if (argc > 1) {
      write(1,argv[1],10);
      line(1);
      end(1);
   } else {
      write(1,"none\n",6);
      close(1);
   }
}

line (int fd) {
   write(fd, "\n", 1);
}

end (int fd) {
   line(fd);
   close(fd);
}
```

Figure 2.7: Giffin et al. (2002) Example C Code

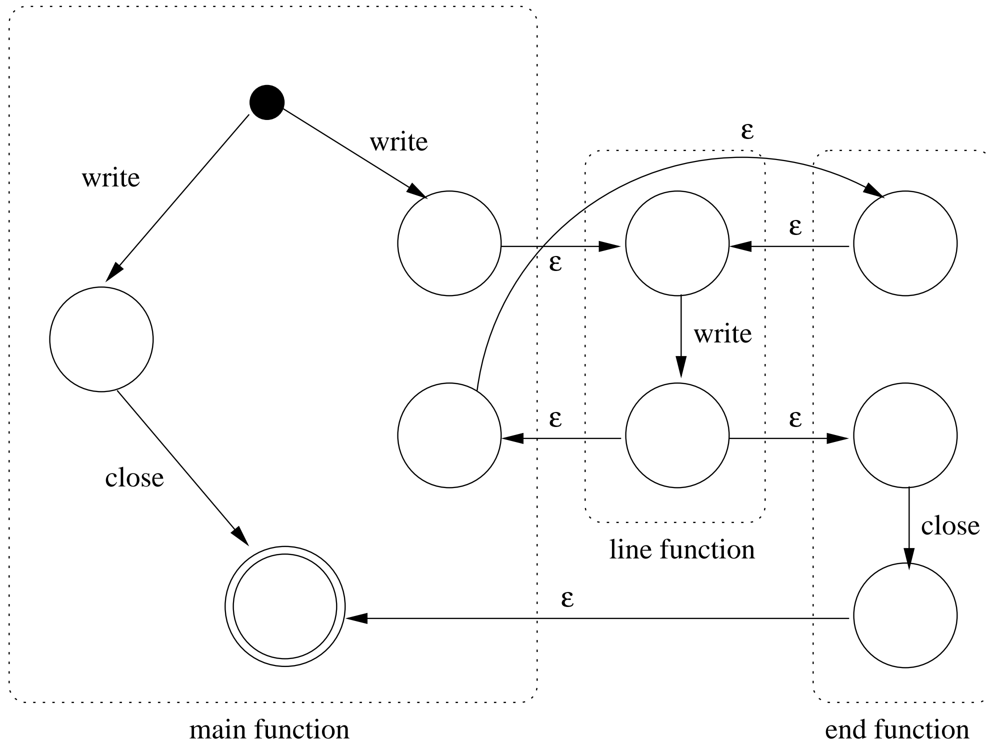

Figure 2.8: Giffin et al. (2002) Example Non-deterministic Finite Automaton

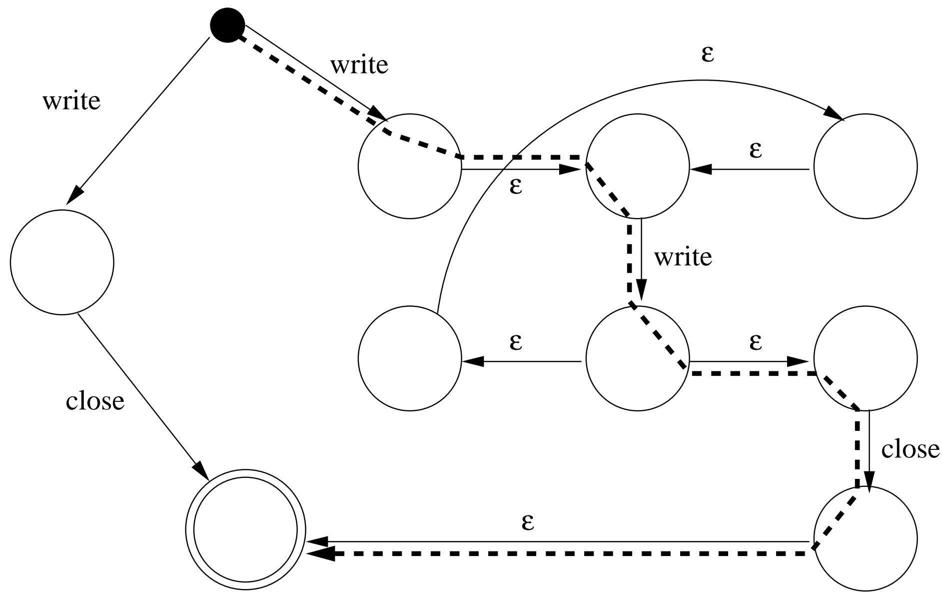

Figure 2.9: Giffin et al. (2002) Example Impossible Path through NFA

The problem of impossible paths can be solved by push-down automata (PDA). Illustrated in Figure 2.10, a PDA solves the example impossible path problem by recording the entrance and exit from the internal program function group automata. These entrances and exits are maintained in a stack structure composed of a unique value for each entrance-exit node pair. This additional information allows the restriction of transitions to better reflect a program's behaviour.

For example, the previous impossible path shown in Figure 2.9 through the equivalent PDA shown in Figure 2.10 attempts to 'pop' the value 'C' from the stack before it has been 'pushed' onto it. This impossible request correctly indicates an impossible path.

The use of both non-deterministic FSA and PDA in PBM is due to the efficiency of the former, and the precision of the latter. The algorithms to generate NFA from a program's definition, and make comparisons with, are of linear complexity (Sekar and Uppuluri, 1999). However, a NFA lacks the precision to correctly representing all program semantics, as illustrated by the previous impossible path example.

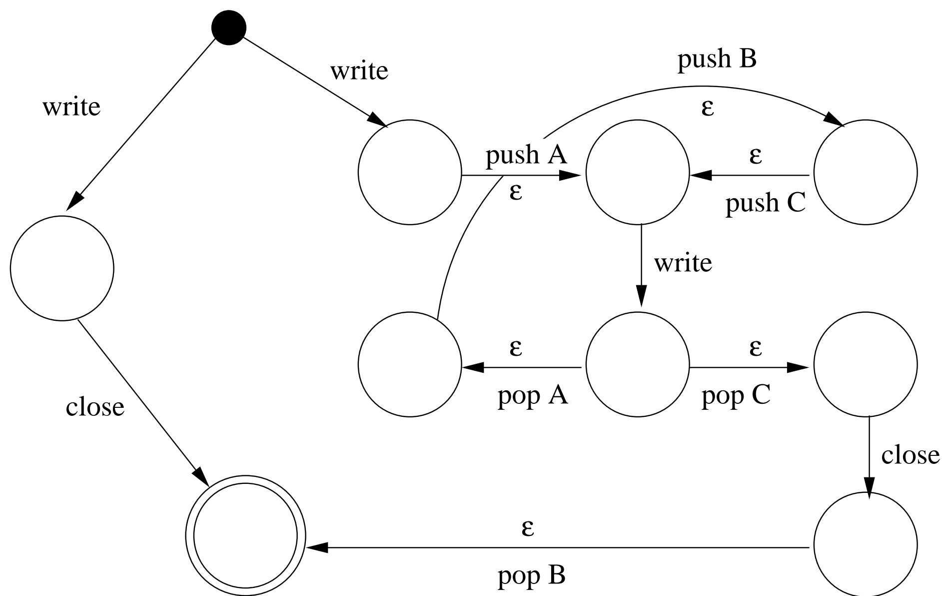

Figure 2.10: Giffin et al. (2002) Example Push-Down Automaton

A PDA[13] however, has excellent precision, and can be generated from any NFA though the operational complexity of PDA is prohibitive (Wagner and Dean, 2001; Wagner, 2000; Giffin et al., 2002; Feng et al., 2004; Giffin et al., 2004).

The majority of research on the topic of FSA-based PBM focuses on the development and application of new representations, designed to offer both precise modelling of program characteristics and low complexity. These new representations include:

- Extended FSA (EFSA).
- Dyck model.

Proposed by Sekar and Uppuluri (1999), an EFSA maintains a series of variables and their values for each state in a NFA. These variables are used to identify impossible paths, in a manner similar

[13] PDA are also referred to as 'abstract stack models' by Wagner and Dean (2001); Wagner (2000).

to the stack structure used by a PDA. Sekar and Uppuluri (1999) admit that the worst-case complexity, however unlikely for smaller applications, is large for EFSA. Expressed in big-O notation, this complexity is $O(n \times m_1 \times m_2 \times ... \times m_k)$ where n represents the number of states and m_i are distinct variable values $1 \leq i \leq k$ where k is the number of variables. An important side benefit of this proposal is the inclusion of arguments for program interactions with basis-terms represented by each transition, an improvement in precision also adopted by others (Wagner and Dean, 2001; Giffin et al., 2004, 2005).

A recent proposal by Giffin et al. (2004, 2005) and Feng et al. (2004) utilises a *Dyck model*, and claims to offer the precision of PDA, with operational complexity similar to that of NFA. A Dyck model employs the same concept as a PDA for the elimination of such issues as impossible paths: each call to a local automaton which represents an internal program function results is assigned a unique value and maintained in a stack structure.

The distinction is that these values are inserted into the program itself, via binary editting libraries such as LEEL (Xun, 1999), as opposed to being maintained by the model, such as with a PDA. This alteration of the program effects no behavioural change, and simplifies the task of the (Dyck) model: instead of both assigning, maintaining and checking the stack structure as occurs with a PDA, it need only check what is present in the program.

However, the Dyck model appears only usable in PBM applications where the program itself is both accessible in its binary or source form, and can be modified.

Formal Grammar

Formal grammars provide the definition of a set of sequences of expressions, which are termed a language. A grammar defines the rule of this language. In the domain of PBM, formal grammars utilise program interactions as expressions, and define a program's behaviour as a set of sentences representing the program's expression sequences.

The semantics and application of formal grammars to PBM are equivalent to those observed in the previous section on automata: an equivalent formal grammar can be defined for each different FSA-based technique defined in the previous section. For example, behaviour represented by a FSA can also be defined as a regular formal grammar, and a PDA can also be defined as a context-free

formal grammar.

While it is important to note the alternate definition format, and its use in representing program behaviour in previous research, a discussion of formal grammar provides little extra information. To illustrate this, consider the often cited work by Ko et al. (1994) that proposes the definition of program behaviour using two concepts:

Alphabets sets : to distinguish program events in terms of the chosen basis-terms.

Sentences : to define the order and sequencing of the events.

It is logical that these alphabet sets correlate with FSA states: both FSA states and alphabet sets are used to distinguish parts of a program's execution. It is equally logical that formal grammar sentences correlate with FSA transitions: both FSA transitions and sentences are used to represent the changes in programs during execution. These two concepts, and their direct mapping to FSA concepts, illustrate that formal grammar techniques can be considered an alternate definition format for FSA diagrams.

2.3.2 Order Exclusive

The final type of PBM techniques is that which excludes explicit storage of the order and sequencing of program interactions. This type of technique is distinct from order inclusive (of any degree) models: order exclusive PBM techniques do not simulate a program's changes in internal state, but rather provide a 'black box' representation allowing comparisons. These models are sometimes termed *hidden* in reference to the obscured definition of behaviour.

Order exclusive models enable comparisons with the program behaviour they internally represent, from which information is gleaned. This characteristic makes order exclusive PBM techniques ideally suited to computer security applications such as intrusion detection where comparisons are the primary function; but poorly suited to software development where conveying understanding of the modelled behaviour is crucial.

Order exclusive modelling techniques are created from existing observations of program behaviour. These observations are often captured during the program's run-time execution. As previously discussed, the generation of models from run-time analysis is termed *training*, and a more thoroughly trained model is more likely to be able to correctly predict program behaviour.

There are primarily two distinct techniques for defining predictive, order-exclusive models from observed behaviours: statistics and artificial intelligence. These techniques are discussed in the following sections.

Statistics

The initial paper on intrusion detection by Anderson (1980) proposed a simple statistical averaging technique for the comparison of observed behaviour with the trained model: summing the squares of the differences between observed and trained program interactions[14].

Much previous research has proposed the application of different statistical methods to the task of predicting the probability of an observed behaviour occurring in the profile. These methods include:

- Mean and standard deviation (Denning, 1987).
- Multivariate (Denning, 1987; Javitz and Valdes, 1994; Anderson et al., 1995).
- Markov process (Denning, 1987; Warrender et al., 1999; Maxion and Tan, 2002).
- Time series (Denning, 1987; Kohout et al., 2002).
- Histogram (Endler, 1998).
- Probability distribution (Eskin, 2000; Eskin et al., 2000).

All of these statistical methods utilise a metric to convert a program's observable behaviour (its program interactions with model basis-terms) into a numerical score suitable for statistical processing.

The numerical quantity generated from the PBM basis-terms can be either a collection of multiple individual program interactions, or a simple translation from one interaction only to a numerical value. For example, a metric may collect multiple system-call interactions into a metric which represents their frequency (Warrender et al., 1999) or rarity (Ju and Vardi, 2001), or a metric may simply translate the basis-term interaction to a numerical value based on a static lookup table (Forrest et al., 1994).

[14] A trained statistical program behaviour model is sometimes referred to as a statistical "profile".

The assumption made by statistical PBM techniques is that the real distributions of program behaviour metrics follow standard statistical distributions. Depending on the particular metrics and distributions chosen, this can lead to poor precision in the PBM (Lunt, 1993; Endler, 1998).

The run-time analysis of program behaviour for training statistical program behaviour models, also introduces the potential for false-alarms due to training limitations or difficulties (Endler, 1998). These limitations arise from the potentially wide and unknown range of program execution paths, as discussed in section 2.2.3.

When making a comparison between modelled behaviour and an observed behaviour, a statistical PBM technique provides a measure of the probability that the observed behaviour is part of the model. This value is continuous between 0 and 1. In contrast, order inclusive PBM techniques, due to their formal nature, provide a discrete output of 0 *or* 1. The use of this continuous value requires the specification of a threshold value, which enables its translation to an usable discrete result. A threshold value defines the probability (between 0 and 1) at which behaviour can no longer reasonably be considered part of the model. That is, the chosen threshold value distinguishes conforming and non-conforming behaviour.

The initial development of statistical behaviour models were applied to representing users' behaviour more commonly than programs' behaviour (Anderson, 1980; Lane, 1999). These models are commonly self-modifying and slowly incorporate new observations which fall below the threshold value (and are considered distinct from those in the model) into their definitions. While this technique can work for users who may change their behavioural habits slowly, changes to the behaviour of a program tend to occur in more discrete steps. For example, the introduction of a new version of the program, or the reconfiguration of the program to function in a different way.

Statistical PBM techniques have two choices in this scenario, discard the existing model and totally retrain, or set the threshold value to recognise much more unlikely behaviours as part of the model. Each choice has negative impacts on accuracy and performance: discarding a model requires training of a new model from scratch, while setting a higher threshold value increases the possibility of false-acceptance errors.

Artificial Intelligence

Artificial intelligence techniques attempt to simulate the problem solving and learning abilities an intelligent being exhibits when perform functions (Frank, 1994). Human beings can quickly learn to perform a range of functions from working examples. Artificial intelligence (AI) PBM techniques similarly attempt to 'learn' the behaviour of programs, based on training examples.

Two types of AI techniques are commonly applied to PBM:

- Artificial neural networks (Ghosh et al., 1998, 1999; Ghosh and Schwartzbard, 1999).
- Rule-based induction (Lee et al., 1997; Lee and Stolfo, 1998; Lee et al., 1999).

Artificial neural networks aim to simulate the biological behaviour of a brain, through a series of highly interconnected processing elements (Anderson and McNeil, 1992; Braspenning et al., 1995). The specific type of neural network most commonly applied to PBM is called *back propagation* (Ghosh and Schwartzbard, 1999). Back propagation networks are provided with a set of values to input elements, the values are processed and passed onwards according to the current element's connections. The total transformation applied to these input values is determined by the processing characteristics of each element, and the weighting of the connections between elements. Once the values have been passed to designated output elements, the network's processing is considered finished, and the results are gathered.

Rule-based induction uses specialised algorithms called *machine learning algorithms* to find patterns in training information (Cohen, 1995; Domingos, 1996). In the field of PBM, these algorithms identify patterns that the observed program behaviour follows. Patterns are identified during observation of the program behaviour, and are discarded when an observed behaviour is contradictory. After processing, the only patterns that remain are those that are never contradicted by observed behaviour, and become the rules which represent the program's behaviour. For example, a rule might be generated from the system-call analysis of a program executing on the SunOS operating system which states: a *vtrace* call occurs 5 calls after a *vtimes* (Lee et al., 1997). This rule might then be considered representative and defining of the program's behaviour.

AI techniques must, like statistical techniques, manipulate the result of comparisons between modelled and observed program interactions into a form that is representative of the result. This is easily

achievable for neural networks given the flexibility of their implementation, and requires that the results of all output elements are consolidated appropriately. For rule-based inductive models, decisions are required about which and how many of the rules must remain unbroken, if observed behaviour is to be considered conforming.

The use of AI techniques for order exclusive PBM, as with statistical techniques, is heavily dependent on thorough training examples. However, an additional complication for AI techniques is the necessity to be provided with examples of what *is not* a program's behaviour, in addition to what *is* (Ghosh and Schwartzbard, 1999). This is necessary to discard patterns which commonly occur in all program behaviour. For example, given understanding of the *meaning* of the behaviours involved, it is logical that all programs perform a *open* system-call before a *read* or *write* one. However, AI does not have this understanding and so will identify this as a pattern of a the particular program's behaviour. Many such common and meaningless patterns can potentially be identified and result in a needlessly complicated AI representation.

To prevent this, an AI technique must be provided with training examples which show this occurring for behaviours which are specified as being not from the program being represented. Adequate training is more difficult to provide for AI techniques, while its absence can impose limitations on their performance.

2.3.3 Partially Order Inclusive

The order inclusive finite automata and formal grammar PBM techniques commonly weigh precision against simplicity. A more precise technique, such as PDA, is less simple and has more operational overhead. These characteristics result from the amount of information these techniques maintain; not only *what* a program does, but *when* must be represented.

Frincke et al. (1998) propose a partially ordered PBM technique which optimises the maintenance of *when* program interactions occur: ordering information is only maintained for those interactions whose order is important and defining. In representing a partially ordered program behaviour model as a state machine, this optimisation reduces the number of transitions that must be maintained for state sequences that include unordered interactions. For example, consider the FSA shown in Figure 2.11 (provided by Frincke et al. (1998)) which strictly represents the ordering of a program's interactions.

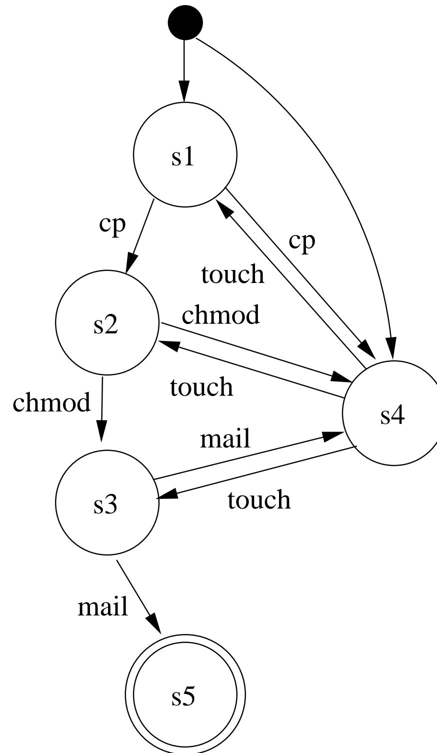

Figure 2.11: Strictly Ordered Automaton with Order Unimportant Event

In this FSA the state 's4' represents the program state just prior to performing the 'touch' interaction. This interaction is unordered, with the restriction that it occurs before the 'mail' interaction after state 's3'. Thus, this example FSA[15] is forced to represent the sequences:

$$touch \rightarrow cp \rightarrow chmod \rightarrow mail$$

$$cp \rightarrow touch \rightarrow chmod \rightarrow mail$$

$$cp \rightarrow chmod \rightarrow touch \rightarrow mail$$

Frincke et al. (1998) recognise that this model can be simplified by better representing the un-

[15] This automaton is non-deterministic and suffers from the impossible path problem described in the previous section. The automata allows recursive sequences, in which circular transitions are allowed but are not representative of the program's behaviour. For example, $cp \rightarrow touch \rightarrow chmod \rightarrow touch \rightarrow mail$.

ordered nature of the 'touch' event. The partially ordered FSA is shown in Figure 2.12. This automaton is augmented with information on the restriction of the unordered event: the state 's4' is restricted to occurring before the state 's3' has been passed.

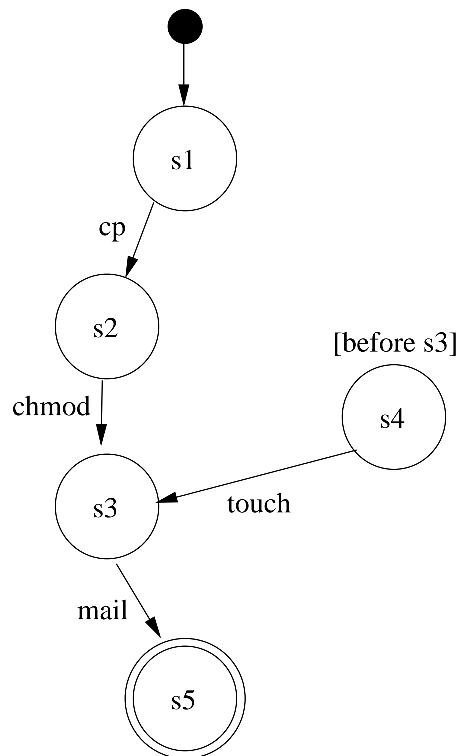

Figure 2.12: Partially Ordered Automaton

By augmenting the representation of specific program behaviour interactions to capture their unordered nature, partial ordering can offer reduced complexity from strictly order inclusive representations. Partially ordered representations, whilst presented here, have not received much attention by other PBM researchers. Regardless, partial ordering is adopted in this study to optimise program behaviour interactions[16].

---

[16] Refer to section 3.2.3 for more details

2.3.4 Summary

This section detailed the different techniques used to represent program behaviour models. These techniques define different ways a PBM technique's selected basis-terms are maintained to provide a representation of a program's behaviour. Proposals in previous research can be categorised into three different types of PBM technique; order inclusive, order exclusive, and the relatively recently developed, partially order inclusive.

Order inclusive PBM techniques represent behaviour by simulating the order and sequence of changes in a program's state. Program states are distinguished by basis-term interactions. Ordered PBM techniques often employ state machines, such as finite state automata, for representation. For example, UML collaboration diagrams (Martin, 1997) are used by software developers to understand the interaction between internal program components. Ordered PBM techniques can be generated either by examining a program's behaviour during run-time, or beforehand by static analysis.

The difficulty in order inclusive PBM techniques is getting a level of precise representation with a model that is of low enough complexity to be practical for the chosen application. The most demanding application is arguably that of real-time anomaly-detection systems used in computer security. Initially proposed non-deterministic finite state automata were found to be too imprecise while the next formal class of precision, push-down automata (according to the Chomsky hierarchy (Chomsky, 1956)), where found to be too complex for real-time applications. The recent development of a Dyck representation has been reported to solve the problem of compromising precision and complexity, but such results have not yet been independently corroborated. Additionally, Dyck representation requires permanent modifications to the original program.

Order exclusive PBM techniques do not simulate the behaviour of a program, and hence offer little assistance to understanding. These techniques allow comparisons between observed program behaviour and the behaviour represented by a model. This limits their usefulness to primarily computer security applications. Order exclusive program behaviour models are commonly trained by run-time analysis of program behaviour. The models are limited in their precision by both the thoroughness of such training examples (the training examples coverage of potential program behaviour) and exposure to examples of impossible program behaviour. Ensuring both a sufficient quantity and coverage of training examples can be a difficult task.

An optimisation of order inclusive techniques seeks to reduce their complexity, by identifying specific program behaviour interactions whose order is unimportant. These techniques are considered partially order-inclusive. The more flexible ordering of these interactions can then be captured in a less complex fashion. Partial order-inclusive techniques often share a large portion of their characteristics with order inclusive techniques.

In summary, the discussed categories and techniques for representing program behaviour have varying characteristics. These characteristics affect the potential performance of the technique. The following section details PBM technique performance, how it is indicated, and what it signifies.

2.4 Performance

There are different performance measurements and characteristics for PBM techniques. To review, a PBM technique utilises a particular set of chosen basis-terms (elements with which a program interactions) to represent program behaviour in a fashion that is usable for the intended application, most commonly software development or computer security.

Two characteristics affect the performance of PBM techniques:

Accuracy : how well a model exactly represents a program's behaviour.

Complexity : how difficult a model is to use.

Both these characteristics are critical to a high performance PBM technique and are discussed in the following sections.

2.4.1 Accuracy

The accuracy of a PBM technique indicates how well a model represents a program's behaviour and can be determined by two measures. These two measures are precision and focus, which cover both potential errors made by PBM technique comparisons: false-alarms, and false-acceptances. Recall that false-alarms occur when a PBM technique incorrectly identifies observed behaviour as not conforming, while false-acceptances occur when a PBM technique incorrectly identifies observed behaviour as conforming.

Precision

The precision of a PBM language has previously been measured by the *average-branching factor* metric defined by Wagner (2000); Wagner and Dean (2001). The branching factor metric is only applicable to ordered PBM techniques and is defined as the number of distinct program behaviour interactions[17] that are acceptable from any point (state) in the model. For example, for a non-deterministic finite state automaton the branching factor for each state X, is the number of unique states linked to it by transitions which stem from X.

In considering what the branching factor represents for PBM applied to computer security, Wagner (2000) proposes that a small branching factor leaves an attacker with smaller choice of future interactions that will evade detection. He also proposes that the branching factor for all program states be averaged to provide a model-wide measure of precision. It is however acknowledged that the average-branching factor metric does not adequately reflect potential areas of high branching within the model.

The precision for a wider range of PBM techniques (partially ordered and unordered alike) can also be measured by the number of false-acceptance errors made during its use (Eskin, 2000; Sekar et al., 2001a; Ghosh and Schwartzbard, 1999). False-acceptance errors can occur for many reasons (partially dependent on the particular PBM technique), though they most commonly occur when the models' definitions of program interactions being compared are similar, but the actual behaviour represented is distinct.

To recap, the precision of a PBM technique can be indicated by two measurements: the average-branching factor as defined by Wagner (2000); Wagner and Dean (2001), and the number of false-acceptance errors made. Higher average branching factors (limited to applications within the field of computer security) and more false-acceptance errors indicate lower model precision.

Focus

The focus of a PBM technique refers to the technique's ability to correctly focus on equal defining behaviours and ignore irrelevant differences. The focus of a PBM technique represents the complement of its precision: the two measures together provide a complete view of a PBM's precision

---

[17] System-calls are used as the basis-terms for Wagner and Dean's work.

performance.

The focus of PBM techniques can be measured by the number of false-alarm errors made during its use. False-alarms errors are also the complement of false-acceptance errors. A false-alarm error can occur for many reasons, depending on the particular PBM technique employed. It commonly occurs when the definitions of program interactions being compared are distinct, but the actual behaviour represented is the same. High false-alarm rates have historically been the main downfall of PBM techniques applied to anomaly-detection systems (Ghosh and Schwartzbard, 1999).

High false-alarm rates often indicate inadequate training in order exclusive PBM techniques. An order exclusive model cannot correctly identify equivalent behaviour if there exist equivalent program execution paths that were not analysed during training. For an order inclusive PBM technique model, a high false-alarm rate often indicates that too many details were discarded to reduce complexity.

The adoption of static analysis and ordered PBM techniques by researchers in the field of computer security has been reported to reduce the false-alarm rate to 0 (Wagner and Dean, 2001). This is an important and valuable characteristic of recent order inclusive PBM techniques.

2.4.2 Complexity

The complexity of a PBM technique can be measured by its time and space requirements. These two computational complexity measures are used to quantify the effects of complexity on program performance, and can be used similarly for PBM techniques.

The more complex a PBM technique is, the less efficient it is at representing program interactions. Reduced efficiency results in more space being occupied by its models, and in using these models to make comparisons taking longer. The time and space complexity of computer programs and PBM techniques is often measured both practically and theoretically.

Time complexities are most often measured by isolating the time spent by the PBM technique application performing its intended function. For example, consider anomaly prevention systems: during the execution of programs, the PBM technique must make comparisons between observed behaviour and modelled behaviour. These real-time comparisons add to the program's execution time and thus, a comparison between the time taken by the program when its behaviour is being observed, and when it is not, provides an indication of the PBM technique's time complexity (Wagner and Dean,

2001; Giffin et al., 2002).

More specific time measurements may be drawn from only the time taken to make the comparisons by the PBM technique within the anomaly prevention system (Feng et al., 2004). In the case of order exclusive PBM techniques, time measurements can also be provided to indicate the required training's complexity.

Space complexities can be measured simply by examining the amount of storage space used by a PBM technique. This space can be in terms of:

Hard disk storage : employed for long-term uses of a model, such as representing it on the computer system.

Memory storage : employed for short-term uses of a model, such as that required by a PBM application to make comparisons.

In addition to practical measures, theoretical asymptotic analysis of complexity such as big-O notation (Weiss, 1999) can be used to indicate the rate at which the complexity of a model grows as the program size grows. This rate is commonly noted as a function of the number of program interactions modelled (Feng et al., 2004).

2.5 Summary

The performance section discusses the metrics by which PBM techniques' performance is quantified. These metrics are categorised by two aspects of PBM techniques: accuracy, and complexity.

Accuracy is indicated by two measures: precision and focus, which can be taken by the number of false-acceptance and false-alarm errors respectively. Another more application-specific metric exists, called average-branching factor, which quantifies the scope within which an attacking program can function without being detected by an anomaly-detection system.

The complexity of a PBM technique is measured by the computational time and space it requires, and by the asymptotic notation in terms of program interaction quantities. Time requirements can be measured for applied PBM techniques by the time taken to use a PBM technique: either to make comparisons between the model and observed behaviours, or to generate a model. Space requirements

can be measured by the amount of long-term hard disk space, and temporary memory occupied by a PBM technique for the same uses.

In summary of this chapter, program behaviour modelling is pervasively used in the planning phase of software development, and in the detection of computer security intrusions. These different applications of PBM utilise distinct basis-terms, and distinct techniques for representation. Despite these differences however, metrics exist to quantify and compare PBM performance.

This chapter identifies several promising characteristics for improving PBM performance. They are:

Abstract Basis-Terms : Application-level program interactions, are suggested to successfully encapsulate low-level program behaviour into a more meaningful, and human comprehensible event.

Partially Order Inclusive Definition : specialized FSA, are reported to reduce the complexity below that of an order inclusive technique, whilst including the information needed to maintain excellent precision, when it is warranted.

Whilst both these characteristics are well recognized, currently no PBM technique utilizes both. The following chapter develops the proposed innovative technique for program behaviour modelling, which incorporates both abstract basis-terms, and a partially order inclusive definition.

Chapter 3

LOGICAL ENTITY ABSTRACTED PROGRAM BEHAVIOUR MODELLING

The PBM technique proposed by this study is termed *Logical Entity Abstracted Program Behaviour Modelling* (LEAPBM). The LEAPBM technique represents the innovative employment of two **key concepts** discussed in the background review from the previous chapter:

Flexible Abstraction of Basis-Terms : the computer system components that the LEAPBM technique uses to define program behaviour are flexible, and variably abstract. Where previous techniques have employed static terms for the definition of models, the LEAPBM technique introduces another layer of abstraction. This additional separation allows a modeler to vary the level of abstraction, of the terms in which a LEAPBM model is defined.

The concept of abstraction has received substantial attention from researchers in the field of PBM. The identification of abstractions has previously demonstrated improvements in PBM applied to computer security through misuse intrusion detection systems (Lin et al., 1998; Lin J., 1998; Ning et al., 2001).

Partially Ordered Finite State Automata : the internal representation of program state employed by LEAPBM takes the form of a modified state machine. This state machine definition is based on a finite state automaton, with modifications to allow flexibility in representing program behaviour whose ordering is not critical (Frincke et al., 1998).

Additionally, LEAPBM has the following characteristics regarding application, generation, and applicability:

Application to Computer Security : the primary application domain for the LEAPBM technique is computer security, more specifically, an anomaly-prevention system. This signifies that the LEAPBM technique represents the entire behaviour of a program, and that it focuses primarily on the interaction of the program with external computer system components.

Static Analysis Generation : the generation of a LEAPBM model is performed statically, prior to execution, via analysis of the program source-code. This concept is also utilised by several other recent PBM techniques and removes the potentially tedious requirement of run-time training during model generation (Wagner and Dean, 2001; Giffin et al., 2002; Murthy, 2003).

Modern Programming Language Support : the LEAPBM technique understands the additional programmatic structures of more modern programming languages, such as C++, Java and Smalltalk. The C programming language has previously been the principal focus of PBM techniques as it is used to define the interface between programs and the commonly employed system-call basis-terms.

A LEAPBM model, being primarily applicable to computer security, captures the behaviour of a program in terms of its interactions with external abstract concepts. This focus on a program's external behaviour makes the LEAPBM technique ill suited to application in the field of software development, as it provides little understanding of the interactions between internal program components or their design.

The LEAPBM technique allows the flexible definition of model basis-terms. That is, a LEAPBM model defines both *what* a program interacts with, and *how* it interacts with it. The suggested basis-terms for a LEAPBM are the abstract concepts in terms of which a program's behaviour is defined, and are called *Logical Abstract Entities* (LAEs). The definition of a particular program and how interacts with a defined set of LAEs is termed a *LEAPBM model*.

A LEAPBM model must incorporate or reference LAE definitions to identify the basis-terms it employs. These LAEs are a critical prerequisite which allow a modeler to take advantage of the flexibility the LEAPBM technique offers.

This chapter details the LEAPBM technique, its constituent components, overall structure, the definition of the flexible abstract basis-terms, two definition formats, and the procedures by which a LEAPBM model can be generated and processed. Finally, specifications are provided for software applications of the LEAPBM technique.

3.1 Design Methodology

The innovative design of the LEAPBM technique was devised after analysis of the concepts currently utilized in PBM, and their short-comings. The first key innovation (enabling the flexible definition of PBM basis-terms as part of the model definition) grew out of identification of the potential for abstraction to target the PBM definition.

This design decision was made based on the logical assumption that enabling targetting to occur on a per-model basis, will facilitate better performing models. This assumption is supported by Sekar et al. (1999), who suggest an advantage in enabling tailoring of abstraction, and is logical as each model can be tailored to utilize the best representative terms.

The second key concept of providing partial order inclusive optimizations to the FSA model representation, is a result of several factors:

Goal of this study ; to enable improved human comprehensibility. This goal necessitates the rejection of any elements of a modelling technique which lead to complexity or opacity.

Indicated optimizations in a review of previous literature Frincke et al. (1998), with a notable point being overcoming the exponential growth of FSA paths with model growth.

Simple trials of modelling programs in the experimental test domain[1]. During these trials the benefits of partial order inclusion on model size were verified, and an apparent improvement in human comprehensibility was observed.

3.2 Logical Abstract Entities

The flexible basis-terms for the LEAPBM technique are called *Logical Abstract Entities*. LAEs form the foundation of a LEAPBM model, and are defined to reflect the abstract external concepts a program interacts with. These external concepts take the form of software libraries utilised by programs. A software library is a store of functionality provided by the computer system. Although typically libraries encapsulate lower-level calls into a more powerful, and abstract, set of functions, the system-call kernel interface can also be considered a software library.

---

[1] Refer to chapter 4.1.

The decision to define LAEs in terms of software libraries was driven by the application of LEAPBM to computer security, and the fact that software libraries provide the interface between a program and the underlying computer system.

LAEs are defined in terms of the external software library components. Multiple, logically-related LAEs can be grouped to form *LAE sets*. LAE sets are referenced by LEAPBM models to identify the abstract external interactions will be used to define a program's behaviour.

The definition of LAEs borrows heavily from the object-oriented design paradigm, which allows the composition of behaviour and data into meaningful constructs called classes. LAE classes are composed of both methods, and properties. Methods and properties are the object-oriented terms commonly given to functions, and variables respectively, that form part of an object. As the LAE class definition draws from object-oriented design, it is capable of capturing newer object-oriented libraries in addition to older simpler software libraries.

The terms used to define the LAEs used by a LEAPBM model are the programmatic elements of software libraries. After these elements' definitions have been captured by a LAE set, they can be used to define pseudo-object-oriented LAE class structures. LAE class structures do not provide some object-oriented functions: inheritance, polymorphism, and distinctions between class and instance members. LAE classes consist only of logical groups of methods and properties. It is via interactions with these abstract LAE methods and properties that a LEAPBM model captures a program's behaviour.

3.2.1 Software Library Terms

The terms in which a LAE is defined are the various elements of both traditional and object-oriented software libraries: functions, variables, methods and properties. LAEs are potentially required to refer to any software library, within a selected programmatic domain. The domain for this study is the C++ programming language.

Each type of C++ software library element must be able to be represented by a LAE class, in order to be used as a basis-term for a LEAPBM model. The software library elements, and their details, are recorded as a preliminary step in the generation of LAE classes. Maintaining an internal representation of each software library element simplifies the potentially numerous references to them

in the definition of LAE classes.

The components of a software library are defined in a programming header file. In object-oriented programming languages, a number of different elements can be defined by software libraries:

1. Functions and methods[2].

2. Variables and properties[3].

3. Extended variable and property types.

4. Object classes.

5. Exceptions.

Functions and Methods

Software library function and method call definitions include the call's name, argument details, and both normal and exceptional return types. These definitions have a standard structure: every function or method call has a name, a return variable type, a set of argument variable types, and a set of exceptional return variable types. Each part of this structure is captured in the LAE representation. An object-oriented method call has one additional complication: its association with an object class.

Both function and method call definitions can refer to extended variable types via their return values, or argument variables.

Variables and Data Members

Software library variables and properties are also defined in the library header files, and can exist in a number of places in a software library:

1. Simple variables.

---

[2] C++ terminology for class functions is *member functions*

[3] C++ terminology for variables is *data members*, while variables associated solely with object classes are termed *static* data members

2. Function and method arguments.

3. Function and method return values.

Simple variables can also exist in several places: within different classes as class or instance properties, and within different namespaces (including the global namespace). Any of the variables or properties used in a software library can be of primitive or extended types. Primitive types are intrinsically understood by C++ programming language compilers, while extended types require further representation.

Extended Variable and Property Types

The types of variables used in a software library can be extended beyond the standard set of value types understood by C++ programming language compilers, termed *primitive types*. Non-primitive or *extended* variable types can be added to a software library through several programmatic constructs such as structures, unions, and object classes. Extended variable types are often defined to add meaning to maintained data. For example, the number 24 means little in isolation, but when part of a structure called ‘Person’ and associated with a variable called ‘Age’, it becomes meaningful information.

The set of extended variable types is dynamic, and likely to vary between software libraries. The potential complexity and variation in the semantics of each variable type is too large to translate to a format which could easily be internally captured by a LAE definition. Extended variable type semantics are best represented in the same way they are defined: computer software. To achieve this, LAEs refer to specialised external software components called *plug-ins*.

Plug-ins are defined to capture the semantics of non-primitive variable types employed in a software library. Plug-ins are referenced by a LAE definition but are external to it. They require packaging, distribution with the LAE if it is transferred, and configuration on the target operating system. Plug-ins should be dynamically loaded by a LEAPBM application and used to compare observed values with modelled values concerning extended variable types. The interface with plug-ins is standardised and occurs though the *checkValues* method call, which is defined in C++ as:

```
bool checkValues(void *aValue, std::string byteValue);
```

The first argument to *checkValues()* specifies a pointer to the variable value that has been observed during a program's execution. The second argument is the string-formatted value present in the LEAPBM model. The *checkValues()* function returns a boolean value indicating whether the two values match. This function must be implemented by all plug-ins, and be accessible via a LEAPBM application. The responsibility for plug-in loading, configuration and communication is placed on the LEAPBM application developer.

Object Classes

Software library object classes provide logical groupings of behaviour and data. Behaviours are defined in methods, while data is stored in properties. These groupings provided by software library object classes can potentially be used as a guide for the definition of the program's LAE set. This is encouraged if the software library object classes reflect the abstract concepts understood by a program.

Capturing the object class structures in a LAE is only one option and is not a requirement. LAEs can refer directly to the methods and properties they contain.

Exceptions

Exceptional returns from software library functions are specialised object classes, designed to represent potential run-time errors and mistakes in program execution. Exceptional returns are an important software library element to capture in a LAE, as they can identify different program execution paths. Exceptional returns are distinguished by their names.

Summary

To summarise, the software library terms utilised in the definition of a LAE are:

methods and functions : their parameters, and their normal and exceptional return values.

Variables and properties : their types, and association with an object class.

Extended non-primitive types : their semantics are captured in external software components called plug-ins. Plug-ins are accessible via a standardised *checkValues* function interface.

Exceptions : their names.

These software library terms are represented as part of a LAE to allow the definitions of other LAE elements, to easily refer to software library elements.

3.2.2 LAE Classes

A LAE class is a pseudo-object-oriented construct, designed to contain logically associated abstract methods and properties. Methods and properties are logically associated if they concern the same abstract concepts. A LAE class represents both the data, and behaviour, of the concepts understood by a program, while a set of LAE classes represent the terms used to model a program's behaviour.

A LAE class is instantiated in a LEAPBM model. Each instance of a particular LAE class can then refer to distinct abstract concepts of the same type. For example, a program with a graphical user interface can reasonably be expected to understand the abstract concept of a button, but is also likely to have more than one button. In this example, a 'Button' LAE class can be defined, and 'Button' instances created for each button in the program.

3.2.3 LAE Methods

A LAE method is defined within a LAE class, and defines a behaviour involving the abstract concept the LAE class represents. For example, consider the abstract concept of a GUI program's buttons. A method of one of these buttons might be 'draw', in reference to the behaviour of a program in displaying such a button.

An abstract LAE method can also encapsulate multiple interactions with software library terms. Abstract concepts that require multiple program interactions are thus represented as a single, abstract program interaction. This allows more flexibility and power in the definition of LAE methods.

Each LAE method maintains the sequencing and order in which the software library interactions it encapsulates occur. The internal structure which is used to represent these interactions is a modified finite state automaton, similar to the structure used to represent a program's abstract interactions in a LEAPBM model. LAE methods' FSA represent individual software library interactions as states, and the sequence in which such states occur is reflected transitions that connect states in ordered pairs.

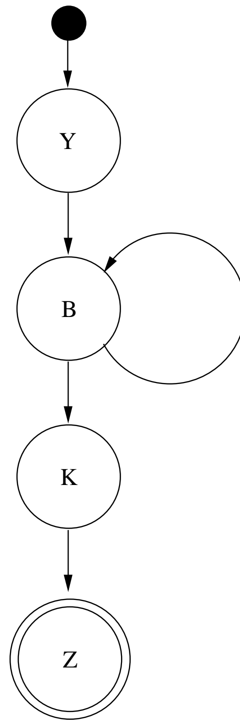

Figure 3.1: LAE Method FSA Example

The software library interactions that can be used in the definition of a LAE method are: method and function calls, and variable and property assignments.

For example, consider a LAE method which encapsulates a sequence of software library method calls: a single Y call, followed by any number of B calls, then a call to K, and finally a Z call. This sequence can be illustrated as shown in Figure 3.1.

Any potential sequence of software library method calls in older programming languages (such as C) can be represented in this simple manner. The possible program behaviour that a LAE method's internal FSA is required to represent is complicated by the C++ programming language. The C++ programming language additionally enables exceptional return types from functions and method calls. Exceptions are often used to distinguish between program execution paths, each of which must be represented in the FSA.

Exceptional transitions between a LAE method's states are identified with the exception that causes them. This identification allows transitions which result from exceptional returns to be dis-

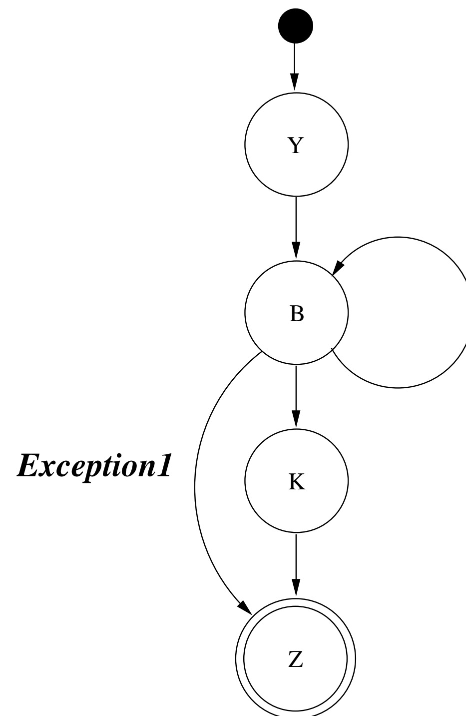

Figure 3.2: LAE Method FSA with Exception Example

tinguished from the single transition which results from a normal return. Continuing the example from Figure 3.1, Figure 3.2 illustrates an exceptional return **Exception1** from state B which causes the software library interaction K to be skipped.

The FSA structure used to define a LAE method is designed to correctly represent any potential sequence of C++ software library interactions, as a single abstract interaction.

Multiple, unconnected sequences can also be maintained in a LAE method FSA. Additional transitions and states can be added with no requirement that all states must be linked. For example, the FSA specified in Figure 3.1 can be extended with states H and R and transition $H \rightarrow R$, as shown in Figure 3.3. This allows multiple software library method interaction sequences to be encapsulated in the one FSA specification, and allows for multiple equivalent definitions of LAE methods.

An additional complication for LAE methods is the potentially unrestricted order in which the software library interactions they encapsulate can occur. That is, situations exist where a software library calls are important to a program's behaviour, but *when* the calls occur during a program's

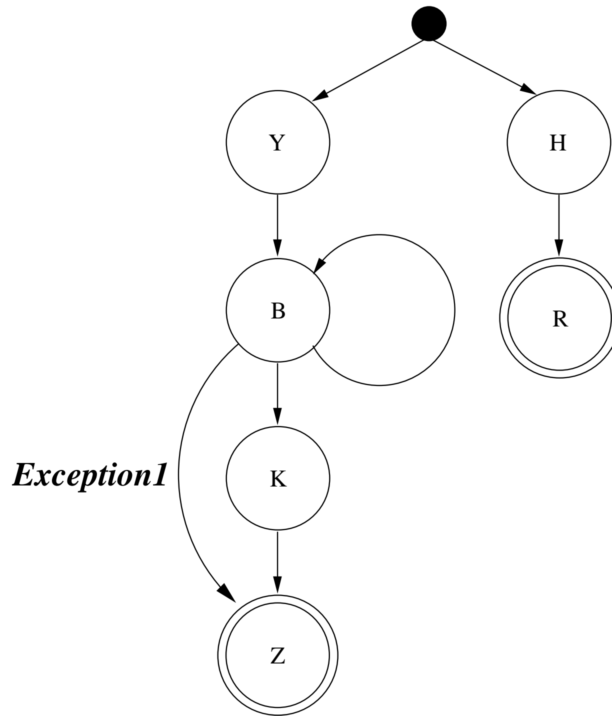

Figure 3.3: Multiple Sequence FSA Example

execution is of little or no importance.

In cases where encapsulating such a software library call is required, the LAE method is given the special flag *unrestricted*. This flag denotes the order-irrelevance of the LAE method. As a result, the order of program interactions with such a LAE method is not considered in determining its conformance with the LEAPBM model. Any additional conditions for the unrestricted behaviour of LAE methods, are maintained within its definition.

3.2.4 LAE Properties

A LAE property encapsulates an atomic data attribute of the abstract concept represented by a LAE class. That is, a LAE property represents a single piece of a LAE class's data, and the sum of a LAE class's properties represent all of its data. LAE properties are associated with a single LAE class, and are defined in terms of software library variables and properties.

Data is maintained or used in variables and properties within a software library. As previously discussed, data is commonly used in variables, function or property arguments, and normal return

values. In the case of object-oriented programming, some variables contained within object classes are not directly accessible but rather are accessed via special methods termed 'getters and setters'. The behaviour of these methods is respectively: to get the value of the data and return it, and to set the value of the variable passed as an argument. Ergo, a data term of concern to an object-oriented software library can be utilised in method arguments and return values, while not being maintained anywhere accessible.

A LAE property definition lists the places within a software library where data is used that constitutes an abstract LAE class's property.

3.2.5 Summary

To summarise, Logical Abstract Entity classes' methods and properties provide the terms used by a LEAPBM model to define program behaviour. LAEs classes capture the abstract concepts understood by a program in abstract behaviours (termed *methods*) and data (termed *properties*). These methods and properties are defined in terms of software library elements. LAE classes group these abstract methods and properties into pseudo-object-oriented classes, which can then be instantiated by LEAPBMs to represent a program's interactions with specific examples of its understood abstract concepts.

3.3 LEAPBM Models

LEAPBM models are built upon the abstract concepts understood by a program, which become the model's basis-terms. Before a model can be generated, these concepts must be identified and specified in LAE classes. LAE classes define abstract behaviours and data in terms of more concrete software library elements. The previous section provides more details concerning the definition of LAE classes.

After a set of LAE classes for a LEAPBM model have been defined, a program's behaviour can be represented in terms of its interactions with these basis-terms. LEAPBM represents the order and sequencing of program interactions with a structure based on finite state automata. However, before any definition of program interactions, a LEAPBM model is required to identify the program it concerns, and the portions of the program's behaviour it represents.

3.3.1 Program Identification and Representation Boundaries

A LEAPBM model identifies which program and which segments of the program's execution it represents. The LEAPBM technique facilitates representation of program segments to represent potential situations where is it not necessary or optimal to model a program in its entirety.

To avoid confusion in matching LEAPBM models to the programs they represent, models refer to programs via the following:

Host : the Internet address or domain name of the machine which executes the program. This identification is however not always possible as often programs can be executed on numerous distinct computer hosts.

File : the absolute path of the file on the operating system where the program code resides. This rather coarse and rudimentary identification of the program file could in future be handled by a checksum value.

The identification of program representation boundaries is achieved with special states that form part of the LEAPBM model's FSA. These states signify the commencement and termination of LEAPBM representation of a program's behaviour, and are termed *entry* and *exit* states respectively. These states refer to execution points in programs, and form the beginning and end respectively, of the LEAPBM model's FSA.

There can be multiple entry and exit points for a given program allowing flexibility for programs with varying control flow paths. Once the program has been identified and LEAPBM representation boundaries established, the program's behaviour is defined by the remainder of the LEAPBM's modified FSA.

3.3.2 LEAPBM Interaction Finite State Automata

The sequence and ordering of a program's LAE interactions is represented in a LEAPBM model by a FSA-based structure. Each state in this FSA represents either a method call, or a property assignment interaction with a LAE instance. Each transition identifies an ordered pair of states and defines an

one-way progression between them. These transitions represents individual steps in the sequence of program behaviour[4].

The following discussion detail the specialisations of the finite state automata construct (originally composed solely of states and transitions) to the task of LEAPBM interaction sequence definition.

States: LAE Interactions

A state in the LEAPBM model's interaction FSA represents either a call on a LAE method, or the assignment of a value to a LAE property[5]. The representation of the LAE interaction holds references to LAE method call or LAE property assignment. Within the model the method calls and property assignments are maintained according to the LAE instance. This provides centralised storage of the use of LAE instances during program behaviour, and is discussed more in the subsequent section 3.3.3.

Transitions: Pathways between Interactions

A transition in the interaction FSA represents a pathway between two states: a source state and a target state. By distinguishing between source and target, the LEAPBM model FSA's transitions are unidirectional.

Transitions whose source state represents a LAE method call, are required to distinguish between normal, and exceptional returns. Thus, multiple transitions can exist between a given pair of states and are distinguished by the return from the source state's LAE method call. Contrastingly, interactions representing LAE property assignments only require one pathway definition.

Flexible Order Importance

Potential situations exist in program behaviour, where the importance of interactions' order varies. These situations must be represented by the LEAPBM finite state automaton structure. As previously discussed, when the order of a LAE method is unimportant, it is flagged *unrestricted*. In the LEAPBM model's FSA-based structure, program interactions with unrestricted LAE methods are

[4] This is similar to the FSA structure employed in the defi nition of LAE methods; with the distinction that states represent interactions with LAE instances, instead of with software library elements.

[5] Fetching the value of a LAE property does not affect the behaviour of the external computer system and is therefore not represented

not maintained. The algorithms which generate and process a LEAPBM FSA should understand the flexible order of these unrestricted LAE methods.

Other situations exist where the order of a program's interactions is only unimportant when compared to a subset of other program interaction sequences. That is, within some groups of program interaction sequences, the order in which interaction sequences occur has no impact on the behaviour they represent. Each such interaction sequence subset is defined as a logical block, instead of explicitly defining all the potential acceptable pathway combinations. Understanding of this flexible order importance block forms part of the LEAPBM generation and processing algorithms.

Additionally, each interaction sequence in this block can be limited in the range of number of times it can be performed. If a particular sequence has not been performed a specified minimum number of times, the processing algorithm is designed to disallow any subsequent transitions. Similarly, if a sequence has been performed the maximum number of times, the processing algorithm is designed to disallow its further use.

This logical block structure (coupled with design modifications to the processing and generation algorithms) represents program situations when different sequences within a subset may be performed both in any order and different numbers of times.

Summary

This section describes the LEAPBM modified finite state automaton used to represent the sequence and ordering of program interactions. To review, these interactions occur with LAE class abstract methods and properties. States within the LEAPBM FSA maintain a reference to the details of the particular LAE method call, or property assignment which are stored centrally according to LAE instance. This storage of the details of program's interactions is detailed in the following section.

The main distinction between the LEAPBM technique and standard finite state automata is the identification of subsets of program interactions whose order is flexible. Representing these flexible interactions requires modification to the LEAPBM FSA and processing and generation algorithms.

3.3.3 LAE Instances and Use

The details of how a program interacts with LAE methods and properties are stored according to each LAE instance. These details are then referred to by the interaction FSA, which allows the implementation of the FSA to be lightweight, as no details of LAE interactions are maintained.

Within a LAE instance, each distinct LAE method call interaction made by a program is represented. This representation stores which LAE method was called, and any property value assignments made as part of its use. Implicit value assignments to LAE properties occur if their source software library terms (parameter, argument, or return value) are used during the LAE method.

Both implicit and explicit LAE property value assignments store which property is set, and the value it is set to. A property's value can either be a constant, or the value of a property of another LAE instance. This allows LEAPBM to correctly capture inter-relationships between run-time specific values.

The LEAPBM representation of property assignment values is less powerful than the approach proposed by Giffin et al. (2004, 2005), which utilises a data-flow diagram to simulate the changing values of variables during program execution. The LEAPBM technique does however, allow for the specification of ranges of values, and sets of values for LAE properties. It is expected that this will allow optimisation and generalisation of an automatically generated model, to make it more widely applicable.

With multiple instances of a single LAE class, some way of distinguishing between them is required. This is achieved through identification of *key* LAE class properties, whose values are unique for each LAE class instance. This simple technique is commonly employed in database systems to uniquely identify data. For example, if a LAE class 'Person' exists, the key properties might be the person's name, and date of birth.

3.3.4 Summary

To summarise, LEAPBM definitions of program behaviour are specified in terms of LAE instances, and a program's interactions with them. The sequence of interactions is maintained in a FSA-based structure, while the details of each interaction are maintained with the LAE instance they utilise.

This section describes the various components that are used by the LEAPBM technique: the LAE

set definition, the LEAPBM model finite state automata, and the LEAPBM model LAE instance usage. Building upon this understanding of the different components of the LEAPBM technique, the following section describes two formats in which a program's model using the LEAPBM technique can be defined.

3.4 Definition Formats

There are potentially many definition formats for the LEAPBM technique. Any format which can define a modified finite state automaton (or the equivalent formal grammar), and a list of object instances and utilisations. Finite state automata are often used to define program behaviour models, as are formal grammars. Contrastingly, previous representations of object instances have primarily been utilised for software development. The LEAPBM technique employs elements of these techniques from fields of computer security, and software development.

Adopting the representation of object instantiations and utilisations for the purposes of PBM, and extending the representation of directed graphs to reference this information, is the primary distinction between a LEAPBM definition format, and existing PBM techniques. Two distinct formats are defined for the LEAPBM technique in the following sections.

3.4.1 Diagrams

Diagrams are a form of specifying information which is primarily human consumable. The diagrammatic representation of LEAPBM models is designed to allow a human to quickly comprehend a program's behaviour, and to provide all the details concerning LAE usage for more thorough scrutiny.

The definition of the set of LAE classes employed by a LEAPBM model as diagrams does not assist in this goal, as the information it provides (the structure of the LAE instances) can be implicitly gathered from the LAE instance usage information. Thus, a LEAPBM model diagram does not include any LAE class definitions. Such a definition involves potentially numerous and complex references to software library terms. The primary purpose of LAE definitions is to simplify and combine such terms into more meaningful constructs. Providing a human consumer with the complex definition of meaningful LAE properties and methods is overly verbose and unnecessary.

The diagrammatic format of LEAPBM models occurs in two parts: the interaction FSA, and the

use of LAE instances. The interaction FSA is heavily based on the representation of a finite state automaton, with extensions to incorporate the additional LEAPBM requirements:

- Defined program representation boundary entries and exits.
- Multiple distinct pathways between states.
- Flexible order importance.
- References to LAE instance utilisations.

The basis for LEAPBM FSA symbols is standard UML notation. For example the entries and exits are illustrated as filled and concentric circles respectively, similar to UML state-chart and activity diagrams.

The LAE object usage part of a LEAPBM diagram, requires the specification of object instances and their use in a diagrammatic format. Although several UML diagram types—collaboration, sequence, activity—are designed to represent interactions with objects of a software system, none allow maintenance of sufficient interaction details. This is potentially due to the exclusive application of UML modelling techniques to software development, and the fact that exact details of changes to object instances are unknown at this time. The diagrammatic definition of LAE object usage, while employing a layout similar to UML class diagrams, has no exact basis in existing work.

Interaction FSA Representation

A LEAPBM interaction FSA has four additional requirements beyond a standard FSA:

1. Program representation boundaries.
2. Multiple pathways distinguished by exceptional returns.
3. Order flexibility.
4. LAE utilisation referencing.

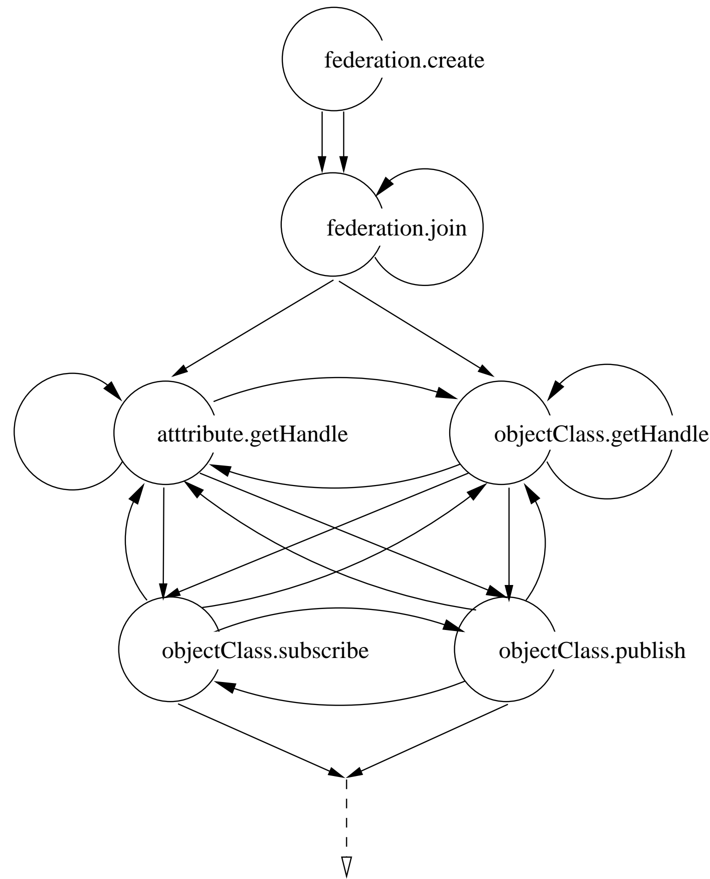

Figure 3.4: Standard Finite State Automaton

These additional requirements are integrated into an example diagram over the following pages. Figure 3.4 shows a segment of a standard finite state automaton diagram. This particular segment represents the initial behaviour of a distributed simulation program. The states in this FSA identify the LAE interactions made by a program. For example, the first state labelled 'federation.create' represents an interaction with an unknown element 'create' of the 'federation' LAE class.

The extension to recognise the entries and exits from the LEAPBM are shown in Figure 3.5. Symbols for entry and exits from based on those from UML activity diagrams are used: a black filled circle, and two black concentric circles respectively. Additionally, the program representation boundaries are identified in terms of internal program function calls.

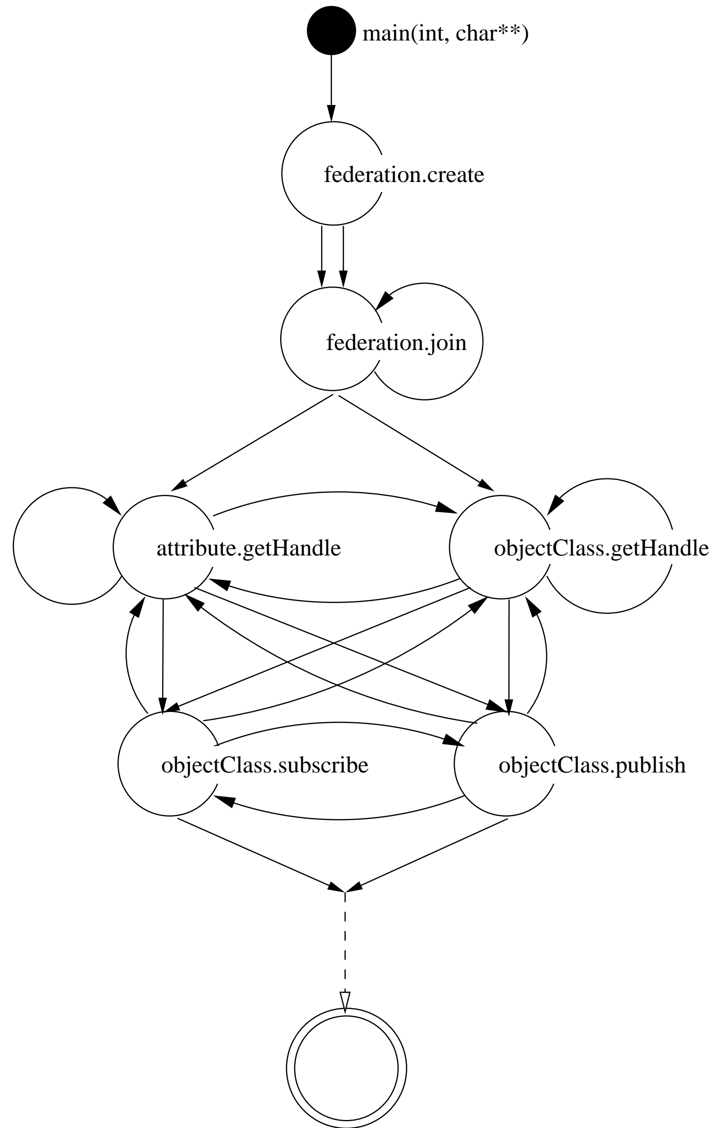

Figure 3.5: Entries Extended Finite State Automaton

Figure 3.6 shows the directed graph further extended to distinguish between multiple pathways for the different returns from the LAE methods represented by states. An exceptional return is identified by a named transition, with the exception in a smaller italicised font. Normal returns are not labelled.

In Figure 3.6 the two previously indistinguishable transitions between the 'federation.create' and 'federation.join' states become associated with different LAE method returns: one is a normal return, the other is a 'FederationExecutionAlreadyExists' exceptional return.

In order to optimise FSA and account for flexibility in the importance of states' order, further extensions are made to represent logical blocks within which order is flexible. In the case of the example FSA there are two interactions with *unrestricted* LAE methods: 'object.getHandle' and 'attribute.getHandle'. Unrestricted interactions are not defined as part of a LEAPBM diagram format. Their presence is implicit given an understanding of the application domain, and inclusion in the LEAPBM diagram format adds complications for little gain. Thus, the states representing these interactions are removed, along with any transitions to or from them.

There also exists in this case a subset of interactions within which order is unimportant. This subset consists of the 'objectClass.subscribe' and objectClass.publish' interactions and is defined as a logical unordered block. The starts of unordered blocks are represented by shaded circles while exit transitions from within unordered blocks to outside are represented by dashed arrows. This is illustrated in Figure 3.7.

In the Figures 3.4, 3.5, 3.6, 3.7 no details are maintained about the LAE interactions made at each state. These details are maintained separately, and referenced by the FSA states appropriately. Figure 3.8 illustrates the FSA extended to maintain these references: each state identifies the LAE instance name, LAE interaction name and type of interaction it represents in colon-separated pairs:

$$\langle LAE\ instance\ name \rangle : \langle interaction\ name \rangle \langle interaction\ type \rangle$$

For LAE method call interactions the 'interaction type' is two parentheses "*( )*", while for LAE property set interactions the 'interaction type' is an assignment operator "=". For example, the 'federation.create' LAE method call on a LAE instance is represented as:

$$fed_XA : create_Fed()$$

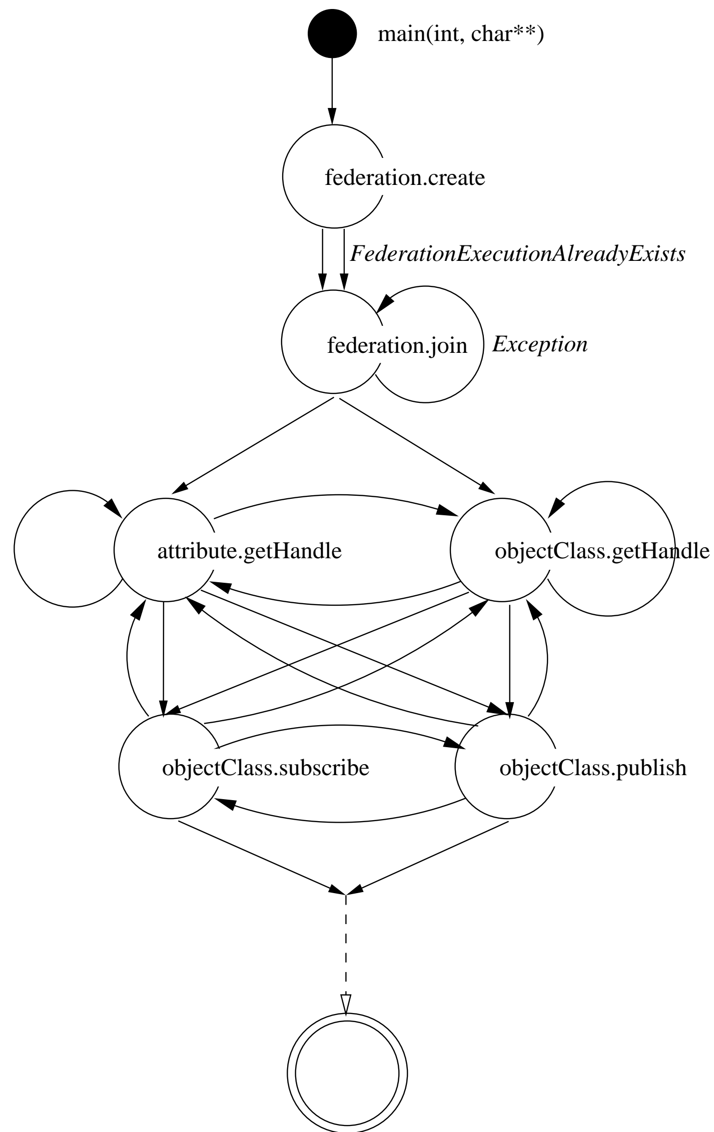

Figure 3.6: Multiple Pathways Extended Finite State Automaton

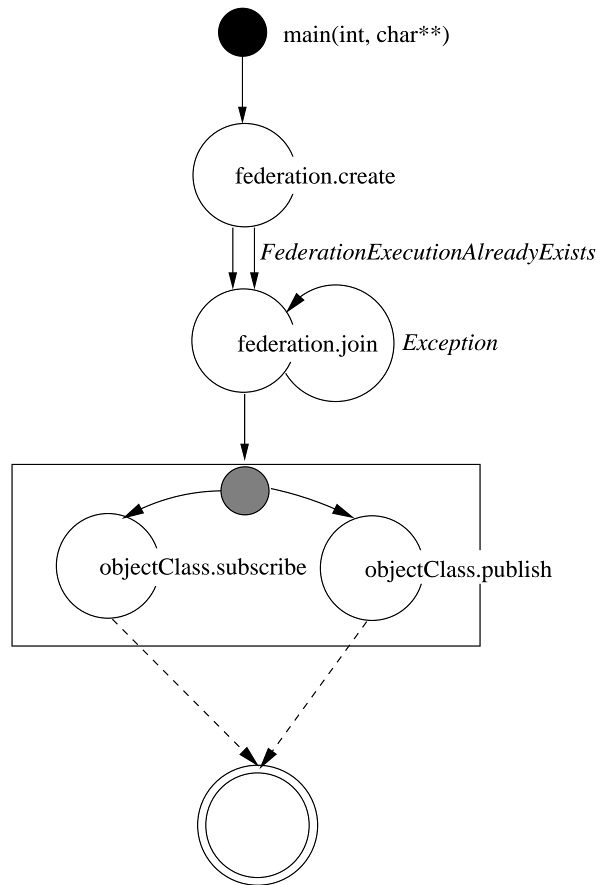

Figure 3.7: Flexible Order Extended Finite State Automaton

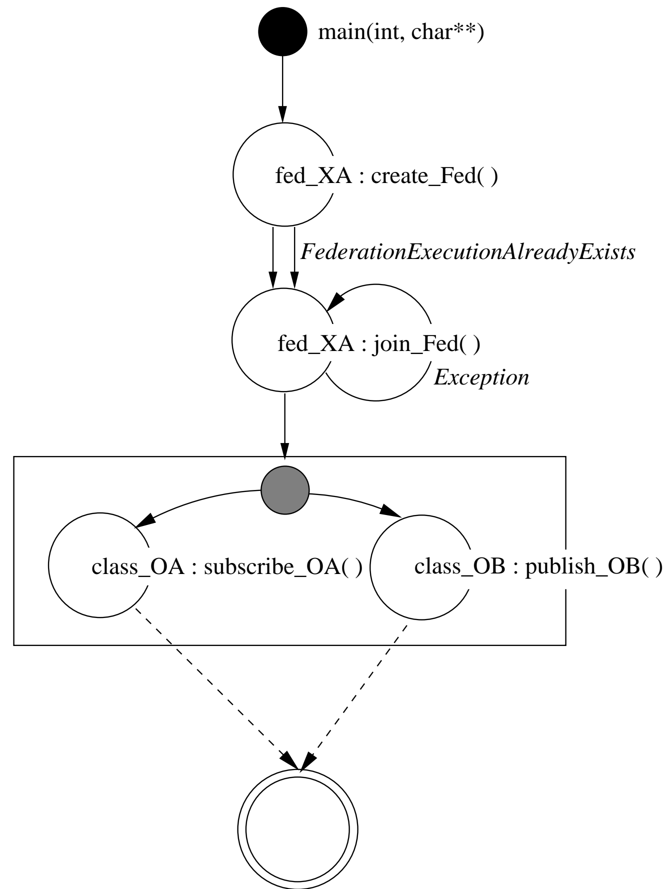

Figure 3.8: LAE Interaction Referenced Extended Finite State Automaton

The details of the LAE object usage referenced by the states in a finite state automaton, such as Figure 3.8, are maintained in a separate structure. This structure is termed the LAE object usage section of a LEAPBM diagram.

LAE Object Usage

Diagrammatic definition of LAE objects' usages is required to represent the details of both method call and property assignment interactions. As previously discussed, a method call interaction can also contain implicit property assignments.

Each LAE object instance is maintained individually as a rectangular box split into three vertically-stacked sections, in much the same fashion as a UML class diagram:

The top section : identifies the LAE class of the object and the name assigned to it in bold font with the format:

$$\langle name \rangle : \langle object\ class \rangle$$

The middle section : represents the property set interactions made with the LAE instance in the following format,

$$\langle interaction\ name \rangle : \langle LAE\ property\ name \rangle = \langle value\ assigned \rangle$$

The bottom section : represents the method call interactions made with the LAE instance, and any included LAE property value assignments, in the following format:

$$\langle special\ flag \rangle \langle interaction\ name \rangle : \langle LAE\ method\ name \rangle$$

$$\rightarrow \langle LAE\ property\ name \rangle = \langle value\ assigned \rangle$$

$$\rightarrow \langle LAE\ property\ name \rangle = \langle value\ assigned \rangle$$

The 'special flag' element present in the bottom section is used to identify the method call interactions which create and destroy the LAE object instance. The flags for instantiation and destruction take the same form as finite state automata entries and exits: a black filled circle, and two concentric

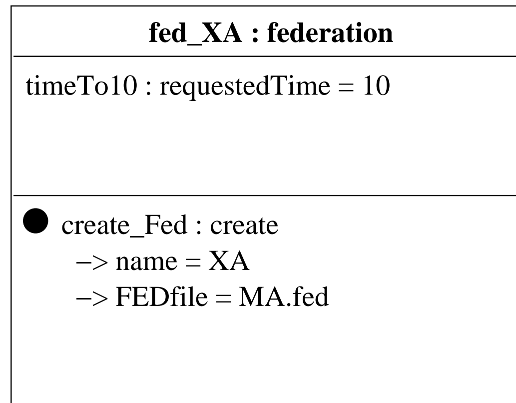

Figure 3.9: Example LAE Instance Usage Diagram

circles respectively. Figure 3.9 illustrates this format for an example LAE object instance representing a distributed simulation object and a subset of its typical usage. Each distinct LAE instance for a LEAPBM model is illustrated in a similar fashion. Their arrangement is at the discretion of the modeler but a general recommendation is to keep the LAE instance usage structures in one area and in proximity with the interaction FSA states which references them.

3.4.2 XML

The representation of LEAPBM models in an XML format provides a definition which is easily utilised by computer software. By complying with XML specification standards, a LEAPBM model can be easy manipulated, consumed, and transferred by computer software. For the same reasons, the definition of LAE classes is also provided in XML format. Unlike the diagrammatic format detailed in the previous section, XML definitions are primarily for computer consumption and are therefore need not consider the potential impact on human comprehension of including details.

An XML definition of a LEAPBM represents each portion of both LAE classes and LEAPBM model definition, as discussed in sections 3.2 and 3.3. The structure of an XML document is hierarchical, which can easily be used to represent a LEAPBM model. XML elements provide containers within which other ‘children’ XML elements can be stored. By logically associating some LEAPBM elements with others, an XML document can define a LEAPBM model. For example, LEAPBM FSA transitions are associated with the LEAPBM FSA state identified by their source.

The attributes of each XML element maintain the direct properties of each LEAPBM element. For example, an XML element representing an interaction with a LAE instance has attributes to maintain the name of the LAE instance, and the name and type of the interaction.

The XML Document Type Definition (DTD) format is designed to specify the allowable XML elements and their structure and attributes in an XML document (Refsnes, 2006). The LEAPBM DTD describes the format for each LEAPBM segment.

XML Example: LAE Method Call Interaction

Consider the previous example of a program's interaction with the 'create_Fed' method call on the 'fed_XA' LAE federation object instance. This interaction is shown in the first state of Figure 3.8. Figure 3.10 shows the XML element which represents this interaction and identifies the represented LEAPBM portions. This XML definition also identifies each XML element to enable referencing between LEAPBM portions.

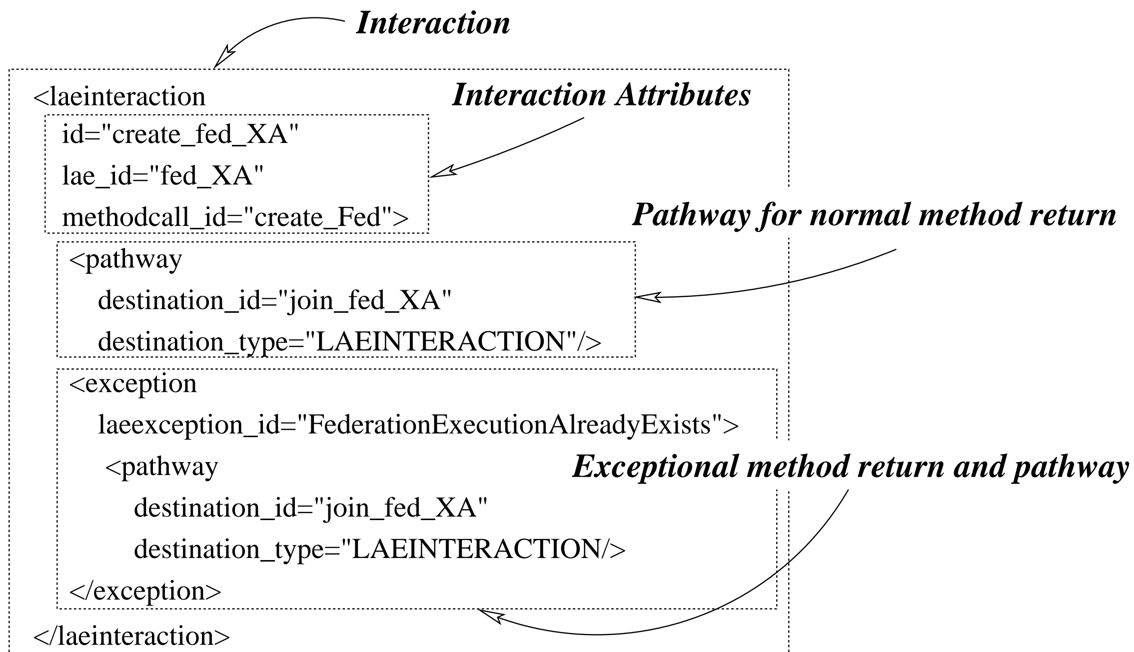

Figure 3.10: LAE Instance Interaction XML Element

The *laeinteraction* XML element shown in Figure 3.10 has several attributes:

id : identifies the unique name given to the interaction.

lae_id : in reference to the LAE involved.

methodcall_id : in reference to the LAE method call involved.

The first sub-element defines the FSA transition, termed a *pathway*, that follows the normal return from the LAE method call. This transition leads to another LAE interaction state of the FSA whose name is 'join_Fed_XA'. The next element defines the same pathway for the LAE method call's exceptional return *FederationExecutionAlreadyExists*. These attributes and pathway sub-elements constitute a valid 'laeinteraction' XML element.

The following section details the algorithm for generating a LEAPBM model (in any definition format) from a program's behaviour.

3.5 Generation Algorithm

The procedure for generating a LEAPBM model definition for a program, requires the expert identification of LAEs, and the definition of a program's behaviour in terms of interactions with them. This procedure can be only partly defined as a computer algorithm. Although standard techniques exist for defining computer algorithms in the interests of brevity the generation algorithm is described here in plain English.

A LEAPBM is generated in a number of stages:

1. Capturing the definition of a software library's elements.

2. Expertly identifying LAEs in terms of software library elements[6].

3. Analysing program source for the flow and usage of LAE interactions.

In some cases however, not all of these stages must be performed. If the set of LAEs which describe a program's behaviour have previously been defined, during the generation of a LEAPBM

[6] This identifi cation is currently performed by a human modeler.

for a similar program, stages 1 and 2 may be skipped. In this case the generation algorithm starts at stage 3, described in section 3.5.3.

The expected time taken to generate a LEAPBM model is expected to depend greatly on the program being modelled, and the complexity of the basis-terms chosen to define its behaviour. However, executing all stages manually (performed as part of this study[7]) took around 60 man hours.

3.5.1 Stage 1 - Capturing Software Library Elements

The definitions of software library terms are captured to allow the definition of LAE classes. Capturing these elements requires the replication of the definition of software library elements: method calls, functions, properties, variables, and exceptions. Although object class constructs often provide a guide for the definition of LAEs in stage 2, they are not required to be referenced and are therefore not captured.

A software library method call definition in the C++ programming language has several elements which must be captured:

- The name of the call.
- Each argument name.
- Each argument type.
- The order of arguments.
- The type of value normally returned.
- The possible exceptional returns.

This can be achieved by string parsing of the definition in the software library header. Figure 3.11 shows how each element of an example string is parsed by the algorithm.

---

[7] Refer to section 7.5.2.

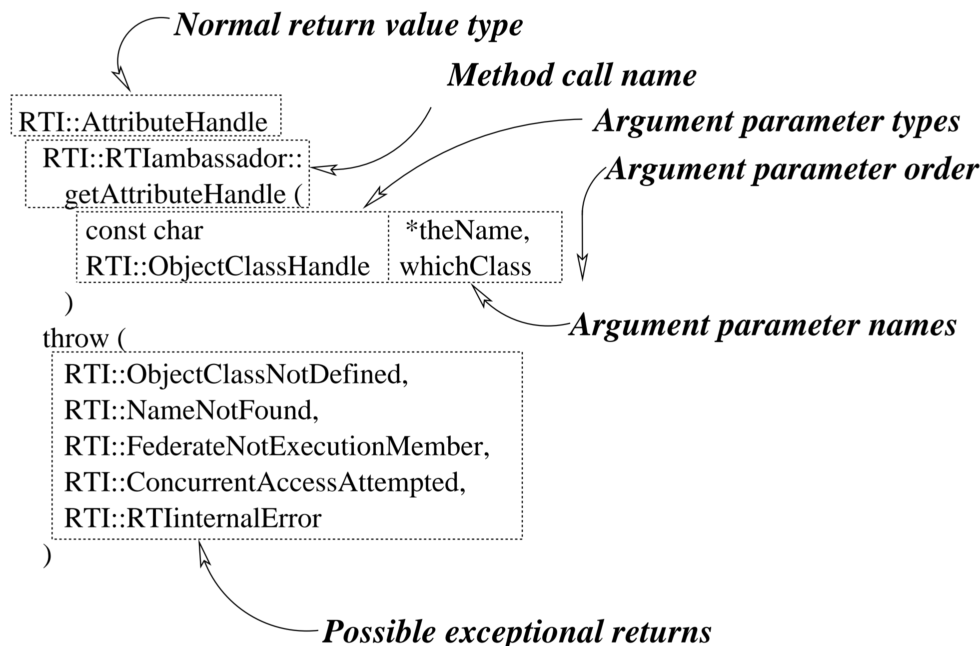

Figure 3.11: Parsing Software Library Method Call Definitions

Variables in C++ software libraries can be of both primitive and non-primitive, or *extended* types. In order for a LEAPBM model to make use of extended types their semantics are captured in external software components called plug-ins, and accessible via a standard function interface[8]. Each variable's name and type are captured in this stage of the LEAPBM generation algorithm.

A software library variable definition takes the general form:

$$\langle variable\ type\rangle\langle variable\ name\rangle$$

The variable name can be followed by parenthesis to indicate the instantiation of an object where the variable type is an object class. This variable format is parsed to identify the name and type as shown in Figure 3.12.

[8] Function definition: *bool checkValues(void *observedValue, std::string modelValue)*.

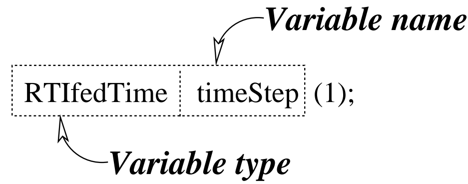

Figure 3.12: Parsing Software Library Variable Definitions

3.5.2 Stage 2 - Identifying LAEs

Stage 2 of this procedure groups together the captured software library elements to define LAE classes. This definition requires the knowledgeable identification of abstract program concepts, and the software library elements that best represent them. The LEAPBM model generation algorithm currently excludes this portion of the generation procedure. Section 3.2 also discusses the identification and definition of LAEs and provides a general guide to assist a human being to perform this process.

This process is expected to be performed by a human with expert knowledge of the software library, and the program domain it is used in. Requiring expert knowledge for the generation of LAEs is acceptable for several reasons:

1. For each domain the LAEs can be reused indefinitely after initial definition.

2. The onus for model generation is expected to fall on the software developer, as is the case with other similar propositions, such as Model Carrying Code (Sekar et al., 2001b). It is also reasonable to expect that the developer of software understands the software and its domain.

Although this stage cannot be implemented as a simple computer algorithm, some computing technologies may be applicable to this task, such as expert systems and optimisation.

Expert systems are a possibility to perform this generation stage, given its emphasis on expert knowledge of the domain, in the appropriate selection of LAE classes. However, fully exploring this possibility is beyond the scope of this study.,

Optimisation is another technique which could potentially be employed in this stage of the LEAPBM generation algorithm. Optimisation attempts to iteratively find the best solution to a problem, as indicated by a function reflecting the ‘cost’ of a selected solution. Thus, in order to use optimisation

for the task of identifying LAE classes, a cost function would have to reflect the performance of a LEAPBM model that utilises a set of LAEs.

This requirement appears unrealistic, as it requires thorough run-time experimentation to test the accuracy and complexity performance for multiple LEAPBM models. The high computational and time cost of this experimentation, coupled with it being required for each individual iteration, means that optimisation is unlikely to be useful for automated LAE identification. Conclusive investigation of this hypothesis is however, beyond the scope of this study.

3.5.3 Stage 3 - Recording Program Behaviour Flow and LAE Usage

Once the LAEs have been defined, the program LAE class interactions can be analysed, and defined in the LEAPBM model. This analysis occurs statically on the program source code to ensure inclusion of all the program's potential execution paths. The generation of a LEAPBM from a program's source code has parallels with the function of program compilers: LEAPBM generation takes a definition of program behaviour and transforms it into a different usable form (Wikipedia, 2006b). This function, and the stage of the generation algorithm it represents, can be implemented as computer software.

The first step for this stage of the generation algorithm is to analyse all the program's execution paths for LAE interactions. During this analysis both the interaction FSA and LAE object usage of the program are defined.

LAE interactions are identified by the program's use of software library elements which form part of the definition of a LAE method call or property. The identification of a program's LAE interaction results in the following steps:

1. The details of the interaction are stored with the LAE instance it occurs on.

2. A new state of the interaction FSA is created to represent the interaction.

3. Transitions are created from the previous state or states in the execution path to the newly created state as required.

Each of these steps is discussed in more detail below.

Step 1 - LAE Object Usage Details

The details of a program's interaction with a LAE object instance are recorded in the LAE usage section of the LEAPBM, within the LAE object instance used. This is determined by instance matching, as discussed in section 3.3.3. A unique key, which is derived from details of a program's LAE interactions, is associated with each LAE instance. This allows the reverse lookup of the exact LAE instance associated with a program's LAE usage. If a new key value is encountered, a new LAE instance is added to the object usage representation, and the LAE interaction being stored is flagged as a constructor.

The interaction's details are stored after the LAE instance matching. These details always include a unique name allocated to the LAE interaction for referencing purposes, but otherwise differ between the two interaction types:

Property set : details the specific property being set and the value it is being set to.

Method call : details the LAE method being called, and includes the details of the (unnamed) property set interactions it may implicitly involve through arguments or return values.

Step 2 - New Interaction FSA State

In this step a new interaction FSA state is created to represent a program's LAE interaction. The new state is named in reference to the LAE interaction's whose usage details were stored in step 1. This new state also maintains a reference to:

- The particular LAE instance.
- The LAE method or property that was utilised.

Step 3 - New Interaction FSA Transition

The creation of a new interaction FSA state represents how the program's LAE interaction may occur in relation to existing LAE interactions. While the majority of these transitions are created to link the state created in step 2 with the state that was created before it, there exist some special cases. Two

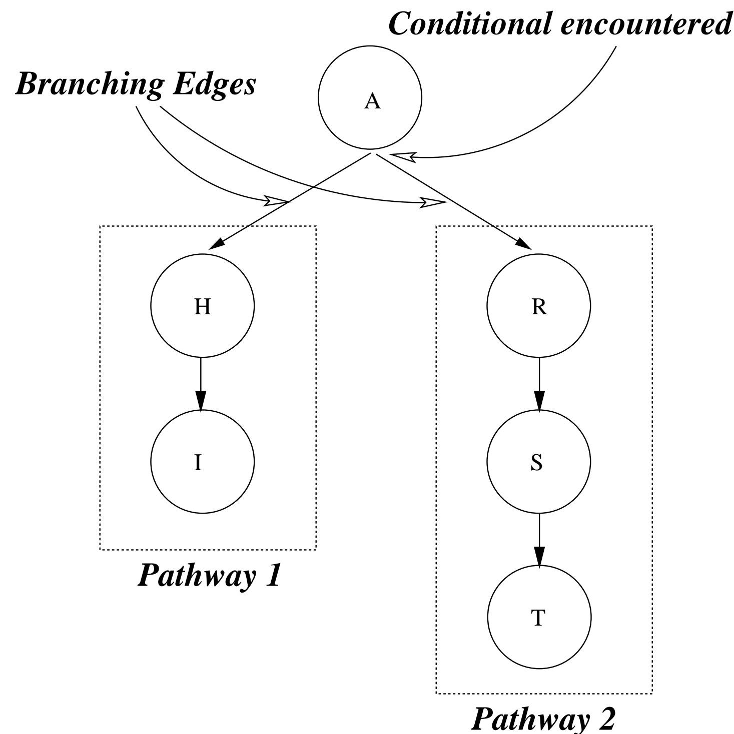

Figure 3.13: Interaction FSA Branching

such special cases involve the presence of conditional, and looping program constructs surrounding the program source that defines a LAE interaction.

In the case of a conditional construct, each distinct outcome pathway is traversed separately, after which the analysis returns to the state at which the conditional construct was encountered. This state becomes the source for the transition which links to the states that represent the alternate conditional execution pathways. This results in a branching of the interaction FSA at this point, as illustrated in Figure 3.13. This example illustrates a conditional construct encountered in the program behaviour control flow after the identification of the LAE interaction ‘A’. The first execution pathway is traversed, during which further LAE interactions ‘H’ and ‘I’ are identified and processed using steps 1-2 of the generation algorithm. At this point the analysis returns to the ‘A’ state and the traversal for the second execution pathway begins, resulting in another transition linking ‘A’ to the new state ‘R’.

Immediately following the complete traversal of a conditional construct, the final states in each outcome control flow path become the source for new transitions to a subsequent state. In the rare

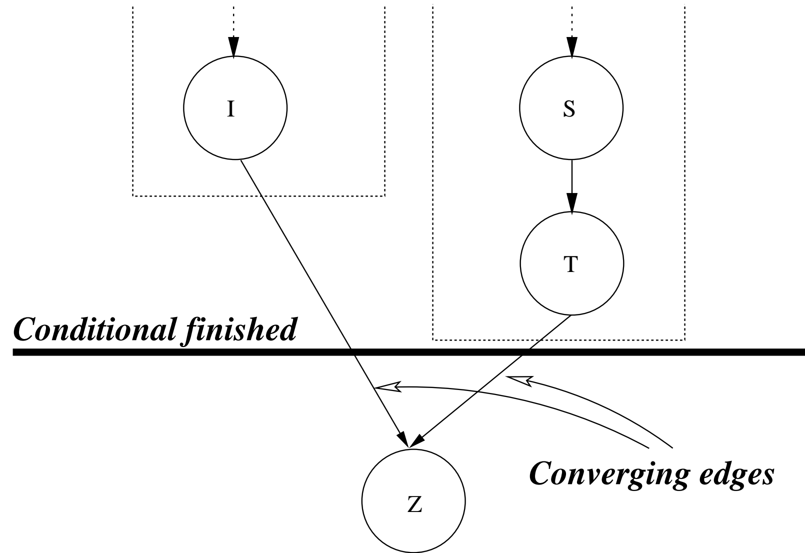

Figure 3.14: Interaction FSA Converging

case where an outcome of a condition was traversed and no LAE interactions were identified, the final state is also considered to be the initial condition state.

The identification of a new LAE interaction after a conditional construct results in converging of the interaction graph at this point as illustrated in Figure 3.14. This example illustrates the completion in analysis of the conditional construct from Figure 3.13. After the conditional construct, another LAE interaction 'Z' is identified. At this point the final states of each outcome path, 'I' and 'T', become the source states for two transitions linking the end of the conditional construct with the next state.

The second special case for the addition of new transitions to the interaction FSA, occurs when a looping construct is encountered. In this scenario the looped execution pathway is analysed, and then a transition is created from the last state in the looped execution pathway, to the first state. This transition captures the nature of the loop construct, which is to perform the same control flow pathway multiple times. This is illustrated in Figure 3.15 which shows an example loop construct encountered after the LAE interaction 'Z'. The looped execution pathway is traversed during which the LAE interactions 'C' and 'V' are identified. The generation algorithm steps 1-2 are performed on these new LAE interactions. At this point the looped execution pathway has been traversed and the special-case looping transition is created, linking the last state 'V' with the first looped state 'C'.

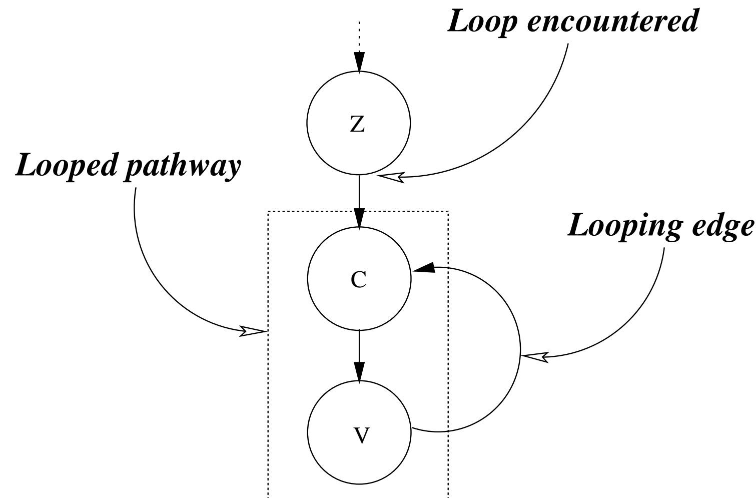

Figure 3.15: Interaction FSA Looping

Following the complete traversal and representation of a looping construct, the final state in the looped pathway becomes the source for transitions to subsequent states.

3.5.4 Summary

The LEAPBM generation algorithm requires the expert identification of LAEs in terms of software library elements, which currently requires human involvement. The stages before (capturing the software library elements) and afterwards (analysing program source for LAE usage) *can* be defined as algorithms which can be implemented as computer software. Implementation of the generation algorithm is beyond the scope of this study.

3.6 Processing Algorithm

The task of processing a LEAPBM model refers to gathering program behaviour observations, typically from a running program, and comparing them with that represented by the model. The procedure for fulfilling this task can be represented and implemented as a computer algorithm. As previously, for simplicity and brevity this algorithm is defined in plain English terms.

The algorithm for processing a LEAPBM model involves comparing the LAE interactions, their constituent software library elements, and the order in which they occur, with those of the observed program behaviour. As these comparisons are performed potentially for each program step, the efficiency of this algorithm significantly contributes to the overall performance of the LEAPBM technique.

The functions enabled by the comparisons between observed and modelled program behaviour, depend upon the application of the model:

In an anomaly-detection system : a matched program behaviour is allowed to execute, and a non-matching behaviour is used to alert a system administrator to the possibility of misuse of the computer system.

In a software development environment : a non-matching behaviour alerts software developers to the incorrect implementation of a program specification.

The comparison algorithm is processed for each software library interaction requested by the program which constitutes part of a LAE interaction. The algorithm determines whether the requested interaction matches any behaviour represented by the model and occurs in several stages:

1. Identifying the allowable LAE interactions.

2. Identifying the interaction's software library elements.

3. Checking the software library element's type and names.

4. Checking the software library element's data values.

3.6.1 Stage 1 - Identifying Allowed LAE Interactions

The first and most significant step in the processing algorithm is to identify the program's LAE interactions which are allowed by the model. How this is achieved depends on whether or not the entire definition of the current LAE interaction has been performed by the program. That is, whether all the software library elements which represent the LAE interaction associated with the current state

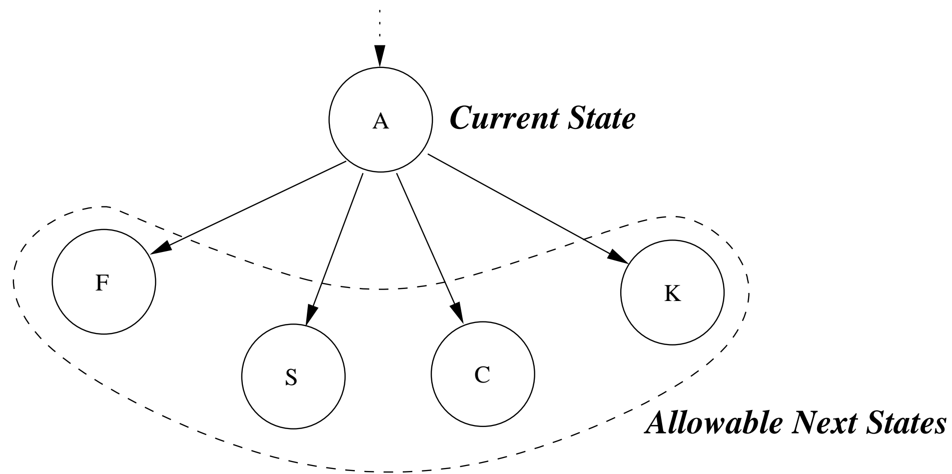

Figure 3.16: Interaction FSA Normal Flow

in the LEAPBM model's FSA, have been appropriately used by the program. If the current state's LAE interaction's definition has not been completed, then the program is not allowed to progress to a subsequent FSA state.

If the current state's requirements have been fulfilled, then the model may continue along the behaviour paths defined by the model's FSA transitions. The transitions from the current state represent these allowed behaviour paths, and identify the states linked to the current state, which represent the next allowed LAE interactions. This is illustrated in Figure 3.16. Additionally, there are special cases for allowable software library LAE interactions:

Unrestricted interactions : which are part of the LAE classes utilised by the LEAPBM must also be checked. Unrestricted interactions are allowed to be performed from any point in a program's behaviour, within the restrictions specified in their definitions.

Unordered blocks : require the checking of other sequences if the current state is the end of a set of states within an unordered block. Each sequence within an unordered block may be performed after any other sequence, within the restrictions specified in their definitions. For example, Figure 3.17 shows a LEAPBM interaction FSA whose current state 'L' is at the end of a sequence

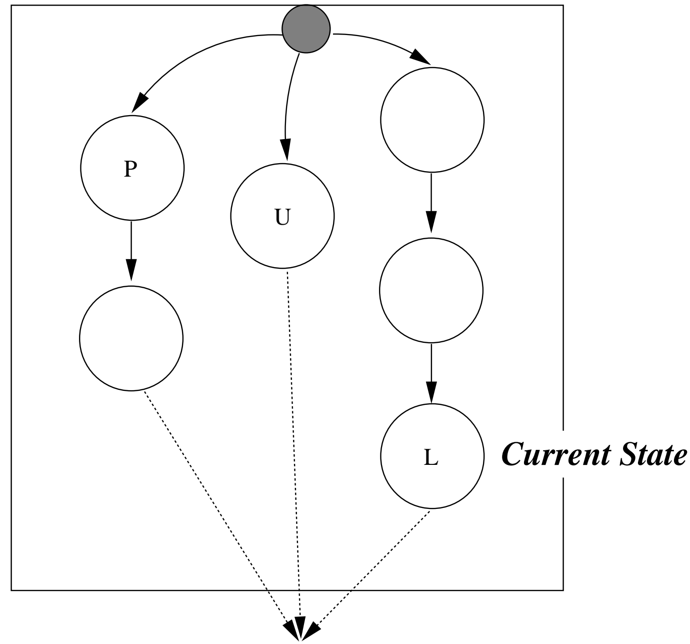

Figure 3.17: Interaction FSA Leaving Unordered Group

in the unordered block. From ‘L’, and in addition to the states it has transitions to, the states ‘P’ and ‘U’ are allowable LAE interactions, as they represent the start of other sequences within the same unordered block.

The following stages 2, 3 and 4 are performed for each LAE interaction identified in this stage. The first LAE interaction which matches the requested program behaviour is adopted by the model and becomes the current state in the LEAPBM model’s FSA.

3.6.2 Stage 2 - Identifying Allowed Software Library Elements

The second stage in the processing algorithm is to identify the software library elements which must be checked against the program request. This is performed using the list of LAE interactions identified in stage 1.

For each LAE interaction, the next software library element in their definition—from the last one processed—represents the allowed software library interaction by the program. In the case that a new

LAE interaction is being examined, this will be the first software library element in the definition of the LAE interaction represented by the FSA state.

The list of potential software library elements is created and utilised in the subsequent stages 3 and 4.

3.6.3 Stage 3 - Checking Type and Name

This stage checks for matches between the type and name of the software library element requested by a program, and those in the list provided by stage 2.

The type of request indicates whether it is a method call or property set interaction. The types represented by each listed software library element are compared with the type of the software library element observed. If these types mismatch, then the current list element is discarded and the next element checked. If the types match, then the names are checked.

The name of the request indicates the specific method being called, or property being set. The name of each software library element (of the right type) in the list of allowable elements is checked with the name of the observed software library element. Each non-matching name element is discarded from the list and the next element is checked. If the names match, then the processing algorithm proceeds to stage 4, otherwise a negative result for the comparison between observed and modelled behaviour is returned.

3.6.4 Stage 4 - Checking Data Values

This stage compares the data values used in the program's observed software library element, with those of each element in the list provided by stage 2. For the software library elements of LAE property assignments, the data value refers to the value being assigned; while for those of LAE method calls, the data value refers to either arguments or return values.

LEAPBM specifies the data values for allowed software library elements as strings. It is the function of specially written plug-in software, termed *checkers*, to correctly interpret this string, and compare it with the data value from the observed program behaviour. In some special cases the data value will represent a link to another LAE instance property value. In this case the actual value of the LAE instance property is substituted in place of the reference before the comparison takes place.

Another special case occurs if the software library element is part of the definition of an unrestricted LAE interaction. No data values are maintained for such interactions, and the comparison is automatically deemed matching.

3.6.5 Summary

The LEAPBM processing algorithm can easily be implemented by computer software. The processing algorithm identifies the program's next allowable interactions with software library elements, from the LEAPBM model FSA, and the LAE object usage details. This set of software library elements is then compared with the observed program behaviour element. If the observed is an element in the list, and with matching data, then it is deemed part of the model, and the model's view of the current point of execution advances appropriately.

The procedure described by this algorithm is performed for each comparison between observed and modelled program behaviour. This will commonly occur repeatedly for most applications of LEAPBM, making this algorithm a crucial component for a LEAPBM application, and influential in determining its time complexity and performance.

3.7 Application Specifications

The LEAPBM technique can be implemented within a specified software application. This application's specifications provide high-level definitions of the functionality required to apply LEAPBM models to the task of detecting anomalies in program behaviour. The functions required of a LEAPBM application are:

1. Loading XML LEAPBM definitions.

2. Maintaining an internal representation of a LEAPBM.

3. Loading and gathering observed program interactions.

4. Processing comparisons between observed and modelled behaviour.

Function 1: Loading XML LEAPBM Definitions

A LEAPBM application must be capable of loading a model from an XML definition. This requires the loading and parsing of an XML-formatted text document, and identification of its element tree's attributes and values. Fortunately, existing software libraries such as the *libxml2* library[9] provide this functionality. After parsing, the program behaviour defined in the XML document is loaded into an internal application structure.

Function 2: Maintaining Internal LEAPBM Representation

A LEAPBM application must maintain the model of a program's behaviour (loaded from an XML file) in an internal structure to ensure efficient access. Efficient access to the model during processing is crucial to the time performance of the LEAPBM application. The XML definition of a LEAPBM is verbose and not well-suited to fast access.

Function 3: Gathering and Loading Observed Program Interactions

A LEAPBM application must specify and employ a format by which observed behaviours can be defined. In the case that the observations of program behaviour are gathered by the LEAPBM application itself[10] the observation format can be simply an internal program definition. In the case that the observations of program behaviour are made by an external software component and provided to the LEAPBM application, this format is open to definition, but must be consistent between the observing software and the LEAPBM application. Although no standard format currently exists, it is hoped that the Internet Engineering Task Force's Intrusion Detection Message Exchange Format (Curry et al., 2004) will be extended to incorporate host-based program behaviours.

Function 4: Perform Comparisons

A LEAPBM application must perform comparisons of individual observed program interactions with a loaded program behaviour model. These comparisons are performed according to the processing

---

[9] *libxml2* is available for download from: *http://xmlsoft.org/downloads.html*.

[10] This case is adopted for this study.

algorithm defined previously in section 3.6. Each observed behaviour is compared with the next allowable interactions defined by the LEAPBM model FSA. A positive result indicates that the observed interaction is part of the model.

3.8 Summary

The Logical Entity Abstracted Program Behaviour Modelling technique provides an innovative approach to representing the functionality of computer programs. This approach enables flexible abstractions for the definition of program behaviour. The terms used to describe behaviour in a LEAPBM model represent abstract concepts, are pseudo-object-oriented, and are specified in terms of software library's interactions. This is a distinct departure from existing PBM techniques which describe behaviour in terms of a fixed set of interactions, commonly low-level operating system function calls.

LEAPBM models can be defined in several formats including modified finite state automata diagrams and the eXtensible Markup Language (XML). These formats are respectively best suited to human, and computer software consumption.

The first time a program employing a new set of abstract concepts is modelled, the abstract concepts used by the program must be expertly identified and specified in LAE classes. Subsequent models utilising the same abstract concepts may reuse this specification and can be generated entirely by computer software.

Processing a LEAPBM involves checking the modelled behaviour (in terms of these abstract concepts) and matching the defined software library interactions with those observed for a program. How the results of these comparisons are handled depends on the specific LEAPBM application.

The following chapters detail LEAPBM's application to anomaly detection and prevention, and the investigation of its relative performance compared with an existing PBM technique.

Chapter 4

INVESTIGATION OUTLINE

Models abstract the complexities of systems to improve understanding, predictions, and development. In the case of computer programs, models are used to abstract often complex source code definitions and allow more simplified investigation and analysis of the program's behaviour.

To recap the discussion in section 2.4, how well a model represents a system can be measured in two ways: accuracy and complexity. These two measures allow comparisons between PBM techniques: that utilise different basis-terms to identify program interactions, and that utilise different techniques to represent the interactions. Accuracy and complexity respectively quantify the degree to which a model correctly represents a program's behaviour, and the simplicity of a model definition.

Accuracy can be measured by quantifying precision and focus errors. A precise model distinguishes program behaviour with similarities in the irrelevant details, and produces fewer false-acceptance errors. A well focussed model equates program behaviour with differences in the irrelevant details, and produces fewer false-alarm errors.

Complexity can be measured by quantifying a model's time and space requirements. A more complex model occupies more space, is harder to understand, and requires more time to make comparisons with. Practical measurements can quantify complexity from working examples, and theoretical analysis can indicate the rate of complexity growth.

This study utilises both practical examples and theoretical analysis to measure accuracy and complexity. The following sections discuss the domain of these practical examples and the specific PBM techniques this study investigates.

.

4.1 Domain: High Level Architecture Distributed Simulation

The domain selected for this investigation is the distributed simulation architecture: the High Level Architecture (HLA). The HLA is a specification which provides intercommunication between sim-

ulation components with potentially heterogeneous implementation characteristics. For example, a simulation written in the C programming language executing on a Sun SPARC computer, can interoperate with a simulation written in the Java programming language executing on a Windows computer.

4.1.1 Background

The HLA's task is to enable the distribution and interoperation of computer program simulations. Simulations capture the functionality of the systems they represent, similar to the function of a model. The distinction between simulation and model comes from temporal awareness: a simulation has an understanding of time and defines changes to a system that occur as time progresses, where a model does not. Simulations represent systems upon which efficient experimentation or testing is unfeasible, whether it be due to cost or time requirements. A simulation is designed to be more easily manipulated and altered than the system is represents, while as closely as possible representing the systems behaviour. For example, simulation is often used in the development of systems for which the production of real prototypes is not feasible, such as aircraft.

Different types of simulations exist. *Analytical simulation*, enable the investigation of a system's performance or behaviour. Analytical simulations are often used in the development of systems, and aim to provide analysis capabilities with less overhead than the production and deployment of a true prototype. This use of simulation is widely applied to various manufacturing and engineering industries.

The second type of simulation is used to provide an environment in which human beings may gain experience and expertise. Such an environment is termed a *virtual simulation*, and is used to provide humans with access to, and training within, an environment which represents a potentially dangerous or otherwise unavailable system. War and combat simulations are employed by the defence industry in order to train personnel; flight simulations are used to train aviators; and the medical industry is beginning to adopt simulation for training doctors and surgeons in dangerous and difficult operations.

In both these types of simulations, accuracy plays a vital role. Accuracy refers to the level of correctness with which a simulation represents the real system: a highly accurate simulation offers a correct representation of system behaviour.

With ongoing advances in performance and processing ability, computer systems have become a

powerful tool for simulation. Computer systems allow quicker analysis of system parameters than previous simulation methods (such as scale models), and also offer highly immersive training environments. Computer simulations are however limited by the following (Wharington et al., 2002):

Computational load : Highly accurate simulations, both training and analytical, require a potentially substantial amount of computational processing power. Training simulations involve detailed real-time graphical processing, while analytical simulations employ potentially complex engineering calculations.

Development cost : Accuracy has a positive correlation with the time and effort required to developing a simulation. The more accurate the simulation, the more time and effort it requires for development.

Limited reuse : The limitation incurred through large development times could be offset by good reuse. Reuse refers to harnessing existing simulations. Unfortunately, poor initial standardisation and widespread ad-hoc development of simulations means the ability to reuse the majority of existing simulations is limited.

Distributed simulation architectures, such as the HLA, attempt to address these limitations. These architectures enable the division of monolithic simulations into smaller portions which interoperate. Each portion can then be executed on its own computer, dispersing the computational load and harnessing greater total computational resources.

In stand-alone computer simulations, there is no guarantee of an existing built-in interface to other software. That is, often simulations are developed as single computer programs which perform all the processing required internally, and are without communication channels for interoperation with other simulation components. This is a significant barrier to increasing reuse of simulations and reducing development time. In contrast, distributed simulation components are all required to communicate with other components. Thus, distributed simulations are guaranteed to have a communication channel which allows other future simulation components to communicate with an existing component.

Distributed simulation architectures, such as the HLA, provide standardised services for this communication and interoperation. Successful interoperation between distributed simulation components allows faster development of new simulations, by enabling the reuse of existing components.

The HLA is the most recent example of a distributed simulation architecture, and is the domain for this study. The Defence Modelling and Simulation Office (DMSO) of the United States Department of Defence developed the HLA standard (DMSO, 1998) which enables interoperability between simulations with heterogeneous operating systems, programming languages, and hardware platforms. HLA was designed to address the shortcomings of previous distributed simulation architectures such as Distributed Interactive Simulation (DIS) (IEEE, 1995), Aggregate Level Simulation Protocol (ALSP) (Wilson and Weatherly, 1994), and Simulation Network (SIMNET) (Miller and Thorpe, 1995).

HLA defines a standard software library interface, to which all simulation components are written. Coordinating and facilitating communication between a group of interoperating HLA-compliant simulation components is the responsibility of the Run-Time Infrastructure (RTI) software. The RTI is comprised of two distinct components:

RTIEXEC : The RTI Executive is responsible for coordinating and delivering communication between LRCs.

LRC : The Local RTI Component is the means for communication between the individual simulation component and the RTI, and is present at each computer.

HLA-compliant simulation components are termed *federates*, while a group of federates cooperating to simulate a system is termed a *federation*. Each federation has an understanding of what is being simulated that is provided by a configuration file termed the *Federation Object Model* (FOM). Each federate's individual view of the simulation is termed its *Simulation Object Model* (SOM). The FOM defines the objects, attributes, interactions and parameters which exist in the federation; from which the SOM defines the subset employed by each federate. The FOM is also required to define the standardised object model used for the management of a federation called the *Management Object Model* (MOM).

All federates inform the RTIEXEC of the FOM portions they produce and consume, which is termed *publication* and *subscription* respectively. The RTI then coordinates communication such that for each object type, only subscribed federates are delivered information updates, and only publishing federates are allowed to produce information updates.

While constituting the most recent and significant effort towards interoperable, high-performance and reusable simulation, the HLA standard is not without imperfections. Numerous deficiencies in the standard have emerged since its release (Wharington et al., 2002):

Poor FOM Agility : The HLA specification allows for future requirements of simulations by enabling flexibility in FOM and SOMs. This provides extensibility, but also introduces other interoperability issues. A federate designed to run in one federation using one object model, will not easily work in another using a different object model. An ideal federate, that will easily operate with distinct FOMs is termed *FOM agile*. FOM agile federates are difficult and expensive to develop.

RTI Interoperability : The specification of the HLA makes no mention of the low-level data formats or protocols for communication between RTI components. This also raises interoperability issues and often means that, despite the standardised program interface, federates can only interoperate if they are using the same implementation of the RTI.

Security : The ease with which the HLA enables connectivity between simulation components, combined with the lack of authentication and access control measures, raises security concerns. The open communication provided by HLA raises issues of proprietary or sensitive information theft, and with the integrity of simulation-dependent development and training procedures. These concerns are exacerbated by the substantial use of HLA within the military and industrial sectors. The HLA standard provides no assurances for security.

Of these deficiencies, HLA security can potentially be improved upon by an anomaly-prevention system. An anomaly-prevention system for the HLA could be used to restrict the behaviour of federates to previously authorised models. Utilising the HLA as a test domain for the LEAPBM technique offers the additional benefit of development towards such a system. A LEAPBM application for the HLA could be developed to act as an anomaly-prevention system, which offers benefits to the HLA community by improving security.

4.1.2 Suitability for Investigation

The HLA provides a good domain for this investigation for a number of reasons:

- Suitable size and complexity.
- Distinct functional components.
- Commonly used in industry.
- Diverse range of system functionality utilisation.
- Simplified control of investigation environment.

The HLA programs used in the following investigations are of suitable size and complexity: they are larger than trivial, and smaller than overwhelming. Additionally, the composite and distributed nature of a HLA federation means that while the federation in total may be quite meaningful and complex, the attention can be easily be placed on smaller and simpler individual federates.

The HLA distributed simulation environment is widely used for simulations: for a number of years all U.S Department of Defence simulations have been required to be HLA-compliant. HLA programs typically make use of a wide range of a computer system's functionality including:

Data access and storage : both long-term on hard disk drives and in short-term memory. Federates use hard disk storage for saving stages of federation execution and configuring new federations, while temporary memory is used for processing simulation functions.

Varied interactivity : with human users. Federates can be both 'human-in-the-loop' in the case of virtual simulations, and entirely procedural in the case of analytical simulations.

Network communication : between HLA federate programs. This communication is facilitated by the HLA's RTI software.

This varied use of a computer system enables thorough and meaningful tests for program behaviour models.

The distinct portions that constitute a HLA federation, and the ability to modify each portion in isolation, simplifies control of the environment used for the following investigations. Figure 4.1 shows the topology of a HLA federation with four federates.

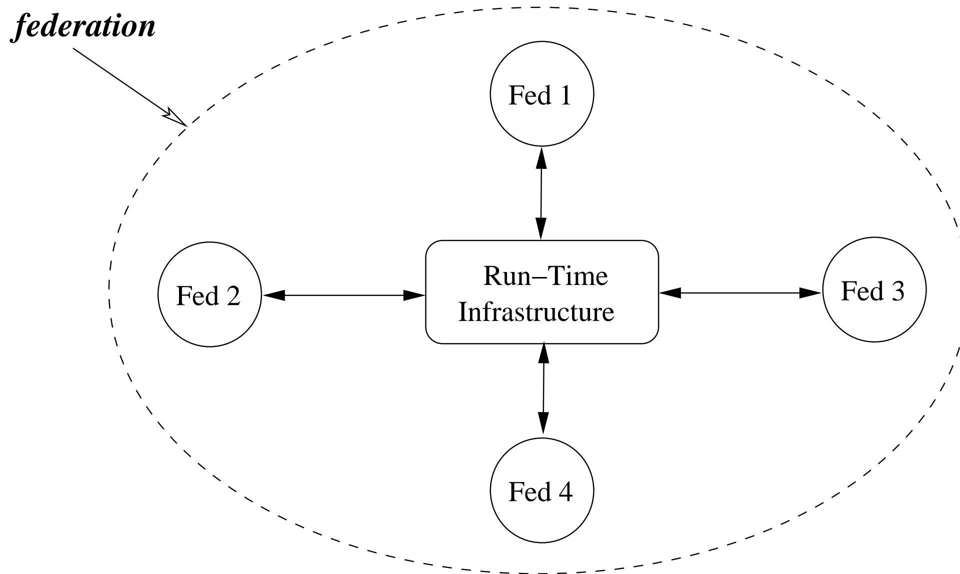

Figure 4.1: High Level Architecture Federation Topology

Each HLA federate can be modified or replaced without any changes required to the remaining federation components, which simplifies control over the investigations' test environment. An alteration to a federate can easily influence the federation's behaviour. Thus, a federation's result should be checked to determine equivalent or distinct behaviour.

4.1.3 Test Environment

The HLA test environment used for this study is composed of two main components:

1. The RTI distribution.

2. The federation.

There are multiple HLA RTI distributions available to suit a range of purposes: from research and development into RTI optimisation, to high-performance simulation. However, currently none are both (Stratton et al., 2004; Givens, 2000):

- Certified by the central HLA authority, the United States Department of Defence Modelling and Simulation Office (DMSO).

- Freely available in a fully-functional state, for research purposes.

In the past DMSO released to the public a distribution of the HLA RTI to promote interest in the technology. This distribution is commonly termed the *RTI-NG*, or *DMSO RTI*, and has been released in a number of versions. The specific version termed *1.3-NGv5* is the basis of this study's test environment.

The HLA federation chosen for this study is called the Air Traffic Operations (ATO) federation. This study focuses primarily on the Aircraft Manager (ACM) federate, which represents the movements and state (including position, speed, direction and mechanical state) of aircraft objects within an international airline service. A full description of the entire ATO federation and its constituent federates is available in Appendix A.1.

Unfortunately, these federates have not undergone conformance testing, and do not have HLA-compliance certification. This is acceptable for several reasons:

1. The ATO federation explores a range of HLA functionality, and constitutes a sizable program, with approximately 20,000 lines of source code.

2. The task of writing a HLA simulation is very time consuming. This task is not within the scope of this study. The ATO federation is taken from an existing source to reduce the time required for development that is peripheral to the goals of this study.

3. In order to achieve HLA compliance certification, a number of steps must be taken Loper et al. (1997); Woldt and Burkhart (1999); Burkhart et al. (1998); Andrews et al. (2006):

   1. Initiation : Federate developer submits a test application and requests information on the testing process, which is then provided by the Certification Agent (CA).

   2. Documentation Testing : Federate developer submits documentation that reflects the object model, and HLA services, employed by the federate. This documentation is checked for incorrect formatting and inconsistencies.

3. Organise Run-time Test : Federate developer and CA swap the files and information necessary to conduct a live test of the federate, and determine a date and time to conduct the test.

4. Live Testing : During a live test of the federate, the CA tests that the federate fulfils its documented object model and interface responsibilities.

This procedure is potentially lengthy and halted as it requires regular correspondence, and is controlled manually. For this reason, obtaining HLA certification for the ATO federation is not viable.

4. The federates in a federation impact and influence each other, thus for an entire federation to be guaranteed HLA compliant, each participating federate must have HLA certification.

5. The tedious HLA compliance procedure is justifiable for large meaningful simulations which will be reused time and again. Even if such industrial-scale simulations were available, they would likely require resources beyond those available for this study, and complicate the testing procedures.

To summarise, this study's HLA test environment consists of the DMSO RTI distribution version 1.3-NGv5, and the Air Transport Operations federation. This RTI was chosen due to its availability, and the federation selected due to its substantial use of HLA functionality, availability, and suitable scale.

4.2 Subject Systems

The subject systems being investigated are: a LEAPBM application architecture for the HLA called *Matis*, and an implementation of the STIDE PBM technique (which utilises system-call basis-terms).

Matis is developed as part of this study to enable investigation of the LEAPBM technique, while STIDE is provided by the Computer Immune Systems research group at the University of New Mexico. The Computer Immune Systems group are also responsible for the initial use of system-call basis-terms for PBM (Forrest et al., 1996; Hofmeyr et al., 1998; Warrender et al., 1999). These two subject systems and their selection for this investigation are discussed in the following sections.

4.2.1 Matis: LEAPBM Application Architecture

The implementation which forms the software artifact through which LEAPBM is investigated in this study is called *Matis*. Matis is an anomaly-prevention system for the domain of HLA distributed simulation, and is implemented in the Linux operating system (Siever et al., 2005). Matis determines whether an interaction requested by a HLA federate conforms to an existing LEAPBM model of federate behaviour. The tasks which Matis is responsible for are:

1. Capturing the federate's behaviour requests.

2. Checking that this requested behaviour conforms to the model.

3. Performing the federate's behaviour in the event that it conforms to the model.

4. Stopping the federate's behaviour in the event that it does not conform.

Development Methodology

The development of Matis was largely influenced by previous work as part of the Distributed Simulation Interposition Library Infrastructure (DSILI) project (Wharington and Andrews, 2002; Andrews et al., 2002a,b). The DSILI project employed interposition to capture and manipulate federate behaviour. The knowledge and expertise gathered as junior lead developer on this project was instrumental in shaping the design of Matis.

The most prominent point of departure from previous work however, is in the manipulation required by Matis. Matis is utilizes the LEAPBM technique to make comparisons of behavioural conformance with a model of federate behaviour, with the resulting manipulation either dropping the behaviour entirely, or performing it unchanged.

The implementation of the LEAPBM technique employs the Boost Graph Library (BGL)[1] in representing the ordered FSA as a directed graph. The decision to utilize the BGL as the basis for the LEAPBM model was influenced by several factors:

---

[1] The Boost Graph Library (BGL) is a software library designed to provide an extensible abstract interface for graph problems, and is available from *http://www.boost.org/libs/graph/*.

Trials : Trial 'from-scratch' implementations of FSA identified the substantial complexity and development required to generate a representation powerful enough to express LEAPBMs.

Simplification of implementation : The BGL abstracts many subtleties of FSA, and provides a simple storage and access facility.

Extensibility : Through use of a templated implementation, the BGL provides extensibility. This extensibility was necessary in order to represent the unrestricted, and blocks of unordered interactions supported by LEAPBM.

The development of HLA LAEs and an example HLA LEAPBM model was performed in conjunction with that of Matis. The development of the HLA LAEs was simplier than anticipated, due to the pseudo-object-oriented nature of the HLA. The HLA, though not providing a object-oriented programmatic interface, does group portions concerning logically related concepts. For example, the interface methods *updateObjectAttributes*, *changeAttributeTransportType*, and *requestClassAttributeValueUpdate* all refer to the attributes of a simulated object.

Thus, each HLA LAE class is a translation of a logically related concept, and LAE methods are a FSA representation of the individual interface methods required to perform the function represented. In most cases these FSA have a single vertex representing a single HLA interface method.

The application of Matis specifically to the HLA enabled iterative development of LEAPBM's LAE requirements. The HLA domain is complex and developing the example HLA LEAPBM model repeatedly raised new requirements of the LEAPBM technique, and that Matis HLA implementation. Each such requirement was analyzed, and the more general form of complication it represented deduced. LEAPBM, and Matis were then extended to incorporate this additional complexity.

This iterative development methodology was repeated until LEAPBM was capable of representing HLA program behaviour, and Matis capable of performing the required tasks.

Capturing Federate Behaviour via Interposition

HLA functionality is provided to federates by the Run-Time Infrastructure (RTI) software library, which enables federates running on different computers to communicate. The DMSO RTI is the software about which the Matis HLA anomaly-prevention system is designed. The technique which

enables Matis to fulfil its first responsibility is called *interposition*. Interposition allows transparent access to the interactions between a program and a software library through the following steps (Jones, 1993):

1. Creating a software library with an identical interface to the one being interposed. The interface of a software library is defined in the programming header files that accompany it.

2. Ensuring that the calls made by the program go to the interposing library. This is achieved in different ways for different operating systems.

Capturing a program's software library interactions as they occur, via interposition, requires no modifications to programs' source code. This means that interposition also allows the capture of behaviour of programs distributed in binary form.

In the case of the *Matis* HLA anomaly-prevention system the interposition steps are fulfiled as follows:

Step 1 : A software library is created which provides the same interface as the DMSO RTI software library. The interface of the DMSO RTI software library, in addition to being defined in the associated header file, is discussed in detail in the documentation accompanying its release (DMSO, 2002).

Step 2 : The Matis software library is given preferential treatment through configuration of the Linux operating system dynamic linker program (Free Software Foundation, 2005). The dynamic linker program (*ld*) matches and links the software libraries requested by a program with those available on the computer system at the beginning of a program's execution.

Successful interposition of the Matis software library in place of the DMSO RTI software library allows Matis to intercept a program's requests for HLA behaviour and fulfils task 1 of a HLA anomaly-detection system. Once a program's behaviour request has been captured it is checked for conformance with the LEAPBM behaviour model.

Checking Requested Behaviour Conformance

Each time a federate program interacts with a HLA distributed simulation, it makes a method call on the interposed DMSO RTI software library. These calls are captured as described above. The LEAPBM model, with which Matis is configured, is then used to check the conformance of the federate's requested behaviour.

This conformance is achieved as per the LEAPBM processing algorithm described in section 3.6. The result of this algorithm is a boolean value indicating conformance or non-conformance.

Performing Conforming Behaviour

If the captured federate request conforms to Matis's LEAPBM model, the behaviour is performed on behalf of the federate. Performing behaviour in the interposed DMSO RTI software library is more complicated than simply performing the appropriate interaction with the HLA software library method or property. Due to the interposition of this library, such an interaction would be directed by the dynamic linker to Matis's interposing software library, as occurred with the initial captured interaction. This results in a circular reference.

To avoid this, the dynamic link loader software is used to determine how to directly access the method or property from the interposed DMSO RTI software library. Once this has been determined, the federate's behaviour request can be performed by direct use of the DMSO RTI software library functions.

The successful interaction with the interposed DMSO RTI software library allows Matis to perform the federate behaviours which conform to the LEAPBM model, and fulfils task 3.

Blocking Non-conforming Behaviour

If the captured federate program behaviour request does not conform to the model then the behaviour is blocked and prevented from being performed. This task is a relatively simple matter of ignoring the behaviour, assuming the configuration of the dynamic loader software can not be circumvented or bypassed. The dynamic loader configuration can potentially ensure all interactions with the DMSO RTI software library are directed to the interposing Matis library, however if this configuration can be bypassed the program can get direct access to the DMSO RTI software library and perform non-

conforming behaviour.

Ensuring that the dynamic loader configuration cannot be bypassed, requires additional operating system configuration, which is beyond the scope of this study.

Anomaly Reporting

When a federate program has finished executing, the Matis architecture provides a report of its anomalous behaviour. Each program requested behaviour that is part of the program's interaction graph that is determined to be anomalous is counted and given as a percentage of the total number of calls requested. In addition, the number of anomalies is reported at the end of a federate's execution.

This report is necessary to this investigation as it quantifies any accuracy errors made by the LEAPBM application.

HLA Logical Abstract Entities

Logical Abstract Entities (LAEs) which represent HLA abstract concepts, are required to allow LEAPBM models for federates to be generated. The LEAPBM technique is designed to provide flexibility in the specification of LAEs, in order to capture a potentially wide range of abstract concepts. A side-effect of this flexibility is an infinite range of possible LAEs that can be specified for a given software library.

This infinite range is a result of the limitless number of potential LAE classes and class elements (method and properties), and allowable multiple reference to software library elements. A software library with methods and function calls totalling k, can have $\sum_{i=1}^{\infty} k!$ distinct LAE method calls. However, it is likely that of these potential LAE classes and class elements, only a much smaller number properly and logically collect software library terms to represent abstract concepts.

For the HLA software library, as for all software libraries, there will be a number of proper and logical specifications, which define LAEs to represent abstract HLA concepts. It is expected that the choice of the LAE classes' specifications within this set, will have an insignificant effect on a HLA program's LEAPBM model's performance. Thorough investigation of this expectation would require the automated generation of a number of LAE sets using HLA software library elements, and measurement of the relative performances of models using them as basis-terms. Such an investigation is beyond the scope of this study.

The specification of the HLA LAEs used in the following investigations draws from expert understanding of the HLA distributed simulation domain. The HLA LAEs were specified to reflect the different abstract concepts upon which HLA federates operate. These LAEs are:

Federation : A group of interoperating programs simulating a system. This is important as a LAE as it provides context for a federate's execution.

Object class : A particular type of object of concern to a simulation, is necessary as it is used to determine a federate's participation in the simulation.

Attribute : A particular object variable, is a LAE as it is used to refine a federate's participation, and defines the accuracy of its simulation.

Object instance : A specific object representing an individual entity in the simulated system, is important as it provides a representation of a federate's responsibility.

Interaction class : A particular type of instantaneous message sent between federate's, is important as it is used to determine a federate's participation.

Parameter : A particular interaction message's variable, is a LAE as it refines a federate's participation, and defines the accuracy of its simulation.

Synchronization point : A specific instant in a simulation time, is a LAE as it define's how a federate cooperates in the management aspect of distributed simulation.

These LAEs are represented in a format based on UML class-diagrams, shown in Figure 4.2.

This format discards some information, most noticeably the mappings to software library elements for each LAE method and property. It is designed only to provide an overall picture of the available LAEs and their contents.

Summary

The architecture which applies the LEAPBM technique in this study is an anomaly-prevention system for HLA distributed simulation, called Matis. Matis is implemented as per the specifications for a

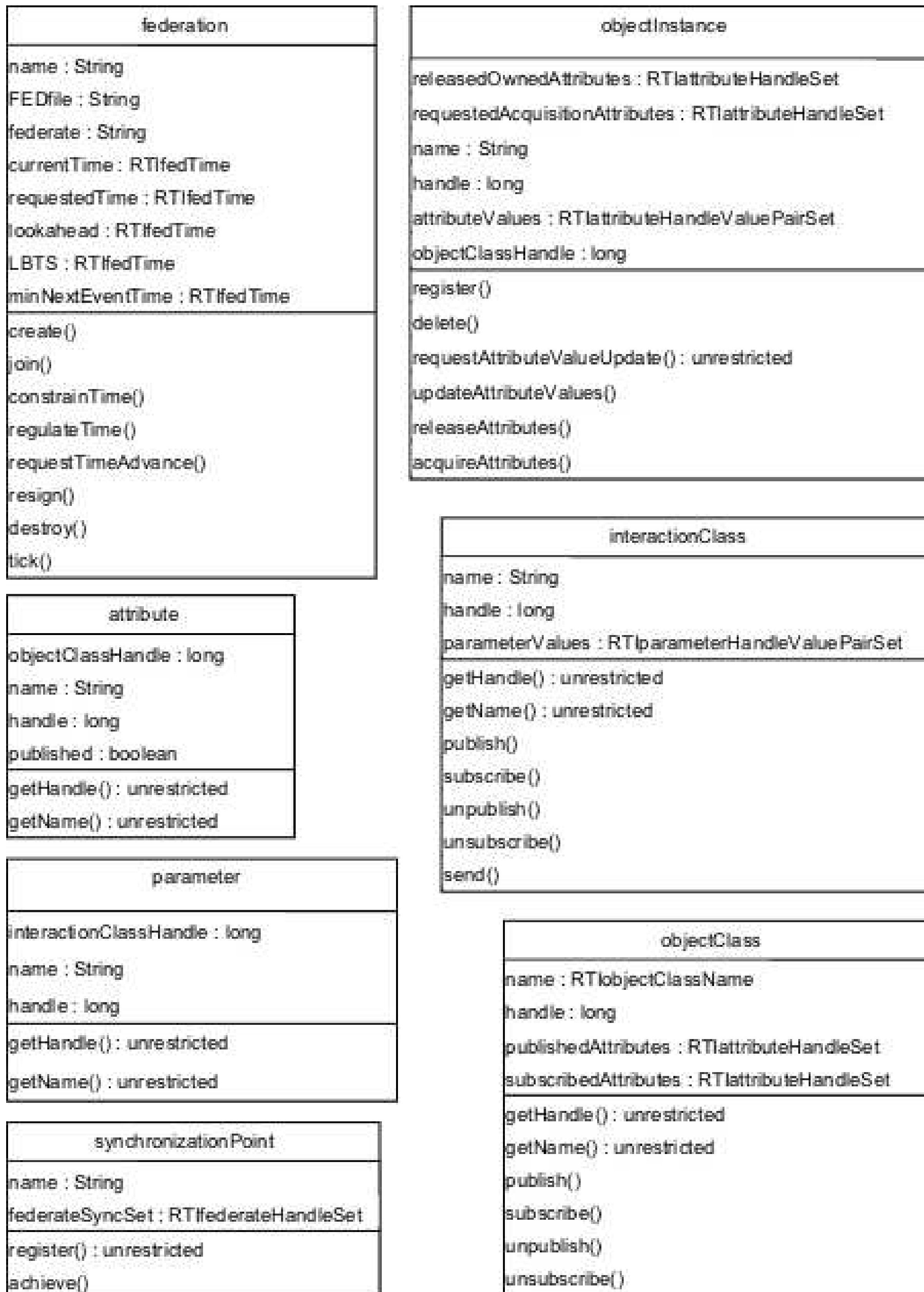

Figure 4.2: High Level Architecture Logical Abstract Entities

LEAPBM application defined in section 3.7, it understands the XML definition of LEAPBM models, and provides comparisons between observed and modelled program behaviour.

The Matis application has the following features:

- Transparent interposition for federate programs.
- XML model format consumption.
- Anomaly detection and prevention.

Matis utilises interposition to transparently observe and control the behaviour of federate programs. This behaviour is checked for conformance to a model of behaviour: conforming behaviour is performed by Matis on behalf of the federate, while non-conforming behaviour is blocked.

4.2.2 STIDE: Existing Modelling Technique

The Sequence TIme-Delay Embedding (STIDE) technique for PBM was derived from previous work by Forrest et al. (1994, 1997) and D'Haeseleer et al. (1996) on applying immunological techniques to computer security. They discovered that short sequences of system-calls serve as a good basis for distinguishing between program behaviours. The resulting STIDE technique uses these short sequences of system-calls, organised as finite state automata, to model a program's behaviour.

STIDE models are generated by training based on the set of observed system-calls made by a program. Each such STIDE model can be described by a series of finite automata. To generate these finite automata from a sequence of observed system-calls, a window of a size k is slid across the sequences, creating a series of unique sequences of length k. Take the illustrative example provided by Hofmeyr et al. (1998): given the observed sequence of system-calls:

open, read, mmap, mmap, open, read, mmap

A window where $k = 3$ slid across this sequence will generate the k length sequences:

open, read, mmap

read, mmap, mmap

mmap, mmap, open

mmap, open, read

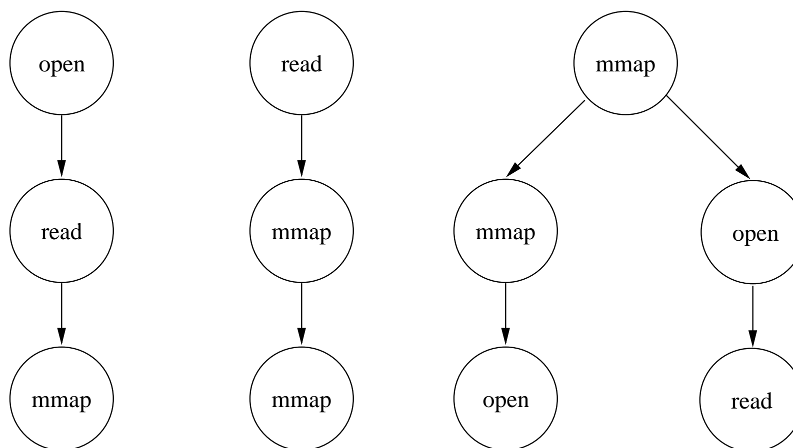

Figure 4.3: Example STIDE Finite Automata: Length 3

These series of sequences of length 3 are then modelled as finite state automata, or trees, to enable efficient processing for each distinct initial system-call (Hofmeyr et al., 1998). This is illustrated in Figure 4.3.

The particular k value chosen has been noted to have an impact on the ability of STIDE to identify anomalous behaviours. A value of 6 was deemed the minimum value necessary to capture commonly occurring anomalies in privileged UNIX programs, though the STIDE development group also use $k = 10$ in certain experiments (Hofmeyr et al., 1998). Both k values are tested in the course of this study.

Suitability for Investigation

STIDE is a good choice as the existing PBM technique to evaluate the LEAPBM technique proposed by this study for a number of reasons:

1. Both STIDE and LEAPBM employ finite automata-based techniques to model program behaviour.

2. STIDE is one of the only mature PBM technique freely available.

3. STIDE is a commonly used existing PBM technique for evaluating new proposals.

The use of finite state automata to represent a program's behaviour by both STIDE and LEAPBM potentially simplifies comparisons of models generated by the two techniques. While not all semantics of the finite automata are shared between STIDE and LEAPBM, both techniques focus on the program interactions and associate each individual interaction with an automaton state. This distinguishes the LEAPBM and STIDE techniques from the majority of other PBM techniques that employ FSA to model program behaviour, which most commonly associate program interactions with FSA transitions. This similarity between LEAPBM and STIDE models further simplifies comparisons concerning attributes of two FSA, such as the number of states and their average-branching factor.

STIDE is one of the only PBM technique that is both mature enough to be applied to a fully functional anomaly-detection system[2] and freely available[3].

Other more sophisticated techniques for the modelling of program behaviour have been developed by several research teams. Two common characteristics of these proposals are their inclusion of arguments to system-calls, and the use of static analysis to overcome training limitations. These techniques and the reasons they were not used in this study are:

Sekar et al. (2001b) have developed a technique called Model Carrying Code (MCC) which employs static analysis, and an extended finite state automaton (EFSA) structure which maintains state variables and system-call arguments (Sekar et al., 2001a; Venkatakrishnan et al., 2002; Sekar et al., 2002). The MCC technique has been demonstrated to date, via integration into the Redhat Package Manager software to produce *RPMshield* (Venkatakrishnan et al., 2002).

While this software demonstrates MCC to be effective in ensuring security against malicious mobile code, it unfortunately does not appear to constitute an anomaly-detection system that is suitable for modelling HLA distributed simulation federate programs.

Wagner (2000) propose a statically analysed technique which also maintains a simple representation

---

[2] The process Homeostasis (pH) Linux kernel extension is freely available from World Wide Web: *http://www.scs.carleton.ca/ soma/pH/*.

[3] Available via download from the Computer Immune Systems Group at the University of New Mexico from World Wide Web:*http://www.cs.unm.edu/ immsec/systemcalls.htm*.

of the arguments to system-calls (Wagner, 2000; Wagner and Dean, 2001). Personal correspondence from Dr. Wagner notes the immaturity of his proposal making it unsuitable for comparison.

Giffin et al. (2002) have developed a technique based on a Dyck model which maintains system-call arguments, and additionally, automatically generates models of the software libraries used by a program (Giffin et al., 2004, 2005). Results of this technique are in previous publications, report positive progress towards solving the problem of compromising precision and complexity. The publication of results indicates that a software implementation has been developed. Although no specific details are given, it seems likely this implementation forms a part of the *Wisconsin Safety Analyzer* software (World Wide Web: *http://www.cs.wisc.edu/wisa/*).

This software is unfortunately not freely available, potentially due to its use of commercial software: *GrammaTech CodeSurfer*^{TM}, and *DataRescue IDA Pro*^{TM}.

The STIDE technique has been used as the basis for comparisons of numerous PBM proposed techniques in previous research (Lee et al., 1997; Debar et al., 1998; Ghosh et al., 1999; Eskin, 2000; Sekar et al., 2001a; Maxion and Tan, 2002; Tan et al., 2002a,b). Evaluating the LEAPBM technique with STIDE potentially provides a common baseline from which predictions can be made about the relative performance compared with other techniques proposed in previous research. These previously proposed techniques are not suitable for comparison as they are uniformly unavailable in a implementation suitable for the task of HLA anomaly-detection.

4.3 Investigations

The goal for PBM techniques, including those employed by an anomaly-prevention system, is to provide the highest possible accuracy while maintaining usable performance. The following investigations utilize the investigation environment outlined in this chapter to examine the comparative accuracy and complexity of the LEAPBM and existing STIDE PBM techniques. This environment is the HLA distributed simulation domain, running a simple air traffic operations simulation, and concerns the Matis software artifact developed during this study, and the existing STIDE PBM implementation.

The first two investigations concern the comparative model accuracy; the first concerning precision, and the second concerning focus. These two investigations aim to compare the two opposite, and complimentary factors of PBM accuracy. The LEAPBM technique is expected to be more accurate, and exhibit both improved precision and focus. This expectation results from both its flexibility, and ability to tailor models to the level of abstraction that best captures a program's defining behaviour.

The final investigation compares the complexities of Matis and STIDE HLA models. The factors of complexity investigated are size, speed, human comprehension, and generation. The LEAPBM and STIDE techniques are expected to provide comparably small and fast program behaviour models, while the human comprehension of LEAPBM is expected to be an improvement over STIDE.

Chapter 5

MODEL PRECISION INVESTIGATION

The accuracy of a program behaviour model depends, in part, on its ability to distinguish programs with different behaviour despite irrelevant similarities. This is termed *precision*. A precise model requires that enough details are maintained for each program interaction such that no other interaction, no matter how similar, is considered equivalent. This chapter investigates the precision accuracy of the PBM techniques STIDE and LEAPBM.

Precision can be quantified by two measures: false-acceptance errors, and average-branching factor. False-acceptance errors occur when a functionally distinct (but similarly implemented) interaction is mistakenly identified as conforming to a model. The average-branching factor metric quantifies the degree of freedom an attacker has in avoiding detection in a computer security application. The precision measure of accuracy is also detailed in section 2.4.1.

In this chapter, the cause of model precision problems in the existing PBM technique STIDE is investigated, and the solution provided by LEAPBM is detailed. This solution is then demonstrated by the *Matis* architecture, on a series of HLA distributed simulation federate programs. Finally, the effect of the LEAPBM technique's solution on the problem of precision is discussed, and suggestions are made for future extensions to the solution to cover other types of programs.

5.1 Task Description

Ensuring that programs on a computer system perform only legal and safe behaviour improves security (Sekar et al., 2001b). This is the aim of anomaly-prevention systems, which make comparisons between observed program behaviour and models of verified behaviour.

Performance and precision accuracy are conflicting requirements of PBM techniques: a precise technique requires the representation of the details of each program interaction, which increases its complexity. An increase in complexity results in increased space and time requirements and consequently reduced performance.

Anomaly-prevention systems often operate simultaneously with the programs they monitor[1]. This scenario requires high performance PBM techniques, to ensure minimal latency of program functionality. By deduction, the PBM techniques used in existing anomaly-prevention systems may be considered high-performance.

The majority of PBM techniques used in existing anomaly-prevention systems are defined in terms of a program's interactions with the operating system kernel. The system kernel provides a single layer of abstraction to direct hardware access, and is accessible by a program via function calls termed *system-calls*. The important behaviour of a program results in system-calls. Thus, defining behaviour in these terms allow potentially excellent model precision. In addition, the set of available system-calls is mostly generic between a large number of operating systems, which enables such techniques to model a wide range of software.

In the interest of simplicity, and the performance improvements it provides, STIDE does not make full use of the information of a program's system-call interactions. The growth of the STIDE database with relation to the number of system-call interactions generated by a program is shown to converge for several programs, such as the rather large *sendmail* program. Sendmail executions, the total of which involved over 1,500,000 system-calls, were able to be stored in a database of approximately 1500 elements (Forrest et al., 1996). STIDE's view of an individual system-call program interaction is limited to identifying which particular system-call was invoked. All values surrounding this system-call, such as return values and argument values, are ignored. The reasoning behind the approach adopted by STIDE is supported by observations by Sekar and Uppuluri (1999) who note that maintaining additional information, such as variable values, can significantly increase the size of models.

The reduction in the details of system-calls represented in STIDE limits its precision, but facilitates real-time performance. Precision is limited as interactions performing distinct behaviour become potentially indistinguishable. For example, if the arguments to a *write* system-call are ignored the PBM will be unable to distinguish between writing to a network socket for communication, and writing to the system password file. The reduction in details represented by the STIDE PBM technique is applied in a generic way: a given set of details are disregarded for each system-call made. No concern or importance is given to any particular system-calls, whose details might be crucial to the

---

[1] Anomaly-prevention systems which operate across relatively low speed network connections—such as that proposed by Giffin et al. (2002)—have less demanding performance requirements.

model precision.

The inflexible and unoptimised disregard of program behaviour details potentially reduces model precision more than is necessary to ensure performance requirements are met. The task for LEAPBM is to provide more precise program behaviour models while maintaining performance.

5.2 Flexible Model Basis

The systems in which PBM techniques are employed, such as anomaly detection and prevention systems, demand high performance and therefore, low computational complexity. Although the level of acceptable complexity is rising as the power of processing devices increases, increasing precision while maintaining complexity will provide an overall improvement. The task of improving precision while maintaining performance is fulfilled by LEAPBM, as a result of its flexibility in model basis-terms.

The basis-terms of a PBM is the set of interactions made by the program, which are chosen to indicate its behaviour. The basis-terms of the STIDE PBM technique is the set of interactions a program can make with the system kernel. Contrastingly, the basis of LEAPBM models is flexible, and can refer to the set of interactions a program can make with *any* software library. A program whose definition is mostly in terms of abstract software library interactions can be modelled in terms of them, while still allowing a program whose behaviour primarily involves direct interaction with system-calls to be modelled by them.

Modern programs often do not interact directly with the system-library, but rather with more abstract software libraries (Fre, 2006). Providing flexibility in the model's basis, allows LEAPBM to represent modern programs in appropriately abstract terms. The degree to which LEAPBM fulfils the task description depends upon the LAE classes chosen for the basis of a particular program's model, and their correct specification in terms of software library elements.

5.2.1 Potential Choices

The software libraries utilised by a program, either directly or via intermediate software libraries, can reflect the abstract concept choices for basis-terms of the program's LEAPBM model. For example, HLA federate programs make direct calls to HLA software libraries which then mediate calls

to CORBA software libraries. This mediation of calls continues through different software libraries, typically decreasing in abstraction, until the system kernel is called and results in use of the computer system hardware.

The system kernel library, which provides the set of system-calls, is the least abstract software library on a given computer system and forms the root of the tree from which all other software libraries stem. That is, all calls on other software library functions typically result in a system-call. Combinations and aggregations of system-calls are logically encapsulated, into more meaningful and abstract functions. Software libraries are groups of such abstract functions. Powerful programs can be more quickly developed by utilising existing software libraries and the abstract behaviour they provide.

The number of system-library calls made by modern programs is much larger than the number of more abstract software library calls performed as shown in Table 5.1. The investigation into simple GUI and HLA programs' abstract and system-call library usage is detailed in Appendices C.1 and C.2.

Program	***Abstract Library Calls***	***System-Library Calls***
Simplest GTK Program	4	590
HelloWorld HLA Federate	332	24918

Table 5.1: Comparative System and Abstract Library Call Usage

The flexible basis for LEAPBM models allows the abstraction of behaviour (and system-calls) that is present in software libraries, to be utilised to simplify a model. Appropriately choosing software libraries for the basis-terms of a LEAPBM model can result in a reduction in the number of program interactions which must be captured. This potentially has the following effects:

- The model's computational complexity is reduced.
- More information can be stored for each program interaction without substantially increasing computational complexity.
- The complexity of the decision-making algorithms is increased to make use of the additional information.

- More detailed information about each program interaction enables a more precise modelling technique.

A full investigation of LEAPBM's effects on model complexity is undertaken in Chapter 7.

5.2.2 Selection Considerations

The selection of abstract concepts to use as the basis-terms for a program's LEAPBM model is important. Modern programs often employ numerous software libraries, resulting in more choices for this selection. Mediated software library calls (those made by direct program software library calls) are typically more numerous again.

The software libraries whose behaviour abstractions best match those of concern to the program, should be selected as the basis for the LEAPBM model. For example, the primary concern of HLA federates is the abstract concepts of federations, simulated objects, their data and time. These concepts are defined in the HLA software library, which is therefore an appropriate selection of basis for a HLA federate LEAPBM model.

While often the selection of LEAPBM basis-terms will follow the program's direct software library calls, this requires careful consideration in all cases for the following reasons:

- Programs can make direct use of multiple software libraries which may not all represent the program's abstract concepts.

- Software libraries, just like programs, have different versions with potentially vastly different behaviour.

- Software libraries used by a program are potentially defined in terms of other libraries whose behaviour is also used directly by the program. The simplicity of utilising the single, lower-level software library as the basis for a model must be weighed against the additional abstraction provided by the more powerful, high-level software library.

- Programs can internally encapsulate calls to distinct software libraries into abstractions, though this commonly occurs externally in software libraries. The custom internal abstractions can

potentially make the most suitable basis for a LEAPBM, however the definition of which will refer to multiple distinct software libraries.

The improvements to precision that careful selection of LEAPBM basis-terms provides, are demonstrated in the following section.

5.3 Demonstrations

The improvements in model precision by enabling the flexible selection of a model's basis are demonstrated against the STIDE technique. Anomalous HLA federate behaviours are compared against models of normal behaviour to test the precision of STIDE and LEAPBM techniques. Anomalous federate behaviour is created by modifications to the functionality of a particular HLA program.

For this experiment, the ATO test federation's ACM federate was modified to produce three functionally distinct versions with the following modifications:

ACM Without Time Regulation : termed *ACMWTR*, this ACM version does not perform the important 'time regulation' HLA distributed simulation behaviour. This means that the federate's events are not sent in a time-ordered fashion, with unpredictable effects on causality and reduction in simulation repeatability.

ACM With Additional Subscription : termed *ACMWAS*, this ACM version requests additional information from the federation on all the simulated objects. Such behaviour could be considered an attack on the confidentiality of the simulation, and could be used in an attempt to steal sensitive or propriety information about the systems being simulated (Andrews et al., 2002a).

ACM With Distinct Data Model : termed *ACMWDDM*, this ACM version performs the same simulation for a Bus, as an ACM federate does for an Aeroplane. This ACM federate's behaviour was modified to employ an object model for a Bus simulation which included renaming multiple Aeroplane specific attributes, and altering the simulation logic to reflect a Bus's behaviour.

These three modified ACM versions represent different modifications to its HLA-related behaviour: ACMWTR has an interaction removed, ACMWAS has additional interactions, and ACMWDDM has similar interactions with different argument values.

PBM Technique	*Comparison*	*Execution Behaviour*
LEAPBM	1	ACM
	2	ACMWTR
	3	ACMWAS
	4	ACMWDDM
STIDE: Sequence Length 6	5	ACM
	6	ACMWTR
	7	ACMWAS
	8	ACMWDDM
STIDE: Sequence Length 10	9	ACM
	10	ACMWTR
	11	ACMWAS
	12	ACMWDDM

Table 5.2: Precision: Observations of ACM Model Comparisons

To demonstrate the improvements in precision LEAPBM offers over the STIDE technique, each of these federates is compared with the model for the ACM federate. The design of the investigation is more fully discussed below.

Design

It is hypothesised, that the flexible nature of the LEAPBM technique will allow it to more precisely differentiate between the ACM federate, and modified ACM versions, than the STIDE system-call technique. In order to test this hypothesis, models of the ACM federate were generated for both LEAPBM and STIDE. These models are not included here for brevity, but are present in Figures 7.1 and 7.2 respectively. Comparisons of each federate version's execution behaviour with the ACM model, are then observed for each technique as shown in Table 5.2. It is expected that the LEAPBM technique will exhibit better precision and correctly distinguish between the similar, yet functionally distinct versions of the ACM federate.

Observations 1, 5 and 9 are control observations which compare the same model and execution behaviours to ensure the PBM technique is functioning correctly. Each of these observations should result in 0 anomalies identified.

Execution	Create Federation	Sync Point Achieve Order
1	Yes	First
2	Yes	Second
3	Yes	Third
4	No	First
5	No	Second
6	No	Third

Table 5.3: STIDE ACM Model Generation Combinations

Method

The investigation comparisons specified in Table 5.2 were performed on the computer detailed in Appendix B, using the ATO HLA federation detailed in Appendix A.1.

The STIDE model for ACM was generated (trained) based on the run-time analysis of multiple, consecutive executions of the ACM federate within the ATO federation until. This training was performed until the model converged. To ensure a working range of behaviour was used to generate the model, these multiple executions varied in terms of:

- Whether the ACM federate created the federation execution: yes or no.
- The order in which the ACM federate achieved the first synchronisation point: first, second or third.

These are the only two inputs into the federate's execution, and multiple federate executions for each combination of values, as shown in Table 5.3 were used in the generation of the STIDE model.

Figure 5.1 shows the growth of the number of 6 length and 10 length sequences in the STIDE models in terms of the number of system-calls executed by the ACM federate during this model generation.

The LEAPBM models were generated manually using static analysis, based on the source code of the ACM federates. The methods required for the model generation of the LEAPBM and STIDE PBM techniques are distinct, with different usability characteristics. Refer to Chapter 7 for a qualitative analysis of these techniques.

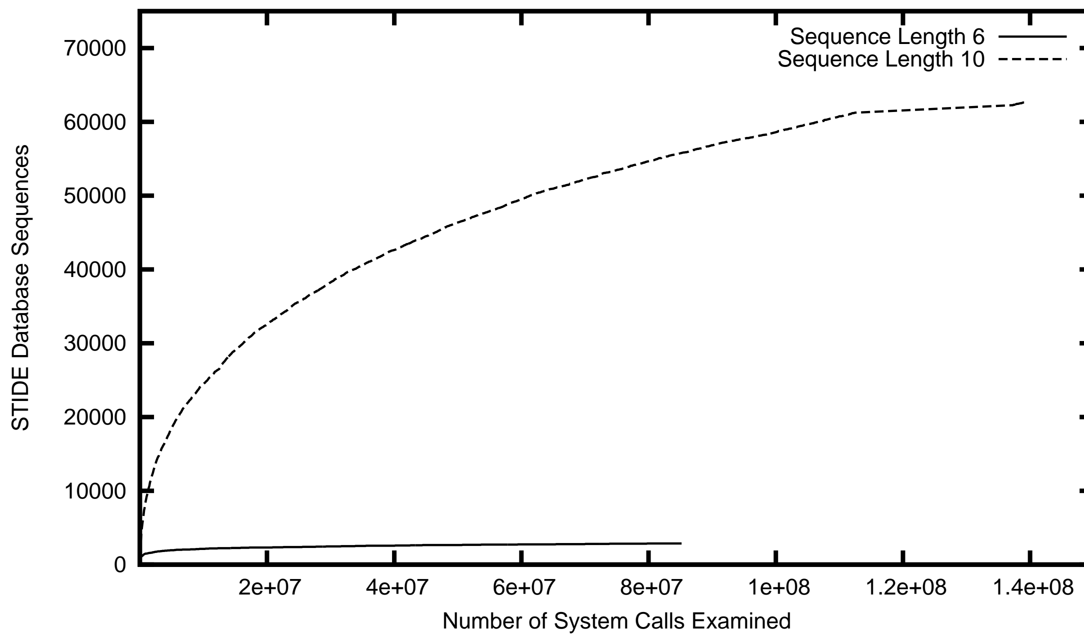

Figure 5.1: STIDE Model Generation Convergence

Results

The results observed for the comparisons listed in Table 5.2 are shown in Table 5.4.

From Table 5.4, comparisons 1, 5 and 9 correctly show practically 0% of anomalous behaviour identified during the comparison between a federate and its model for both the LEAPBM and STIDE PBM techniques. Comparisons 2-4, 6-8 and 10-12 highlight the increased precision accuracy the LEAPBM technique provides over STIDE: 84.84% versus 6.5% and 0.06%, 94.62% versus 0% and 0.04%, and 99.18% versus 0% and 0.04%. The LEAPBM technique identified a higher percentage of anomalous behaviour, for each version of the ACM federate with distinct HLA functionality.

These comparisons show the relative poor precision accuracy offered the STIDE technique for the test Air Transport Operations HLA federation. These result supports the hypothesis that the LEAPBM technique is able to more precisely differentiate between the similarly implemented, yet functionally distinct HLA federate programs more effectively that STIDE.

Technique	Comparison	Behaviour	Anomalous %
LEAPBM	1	ACM	***0.00***
	2	ACMWTR	***84.84***
	3	ACMWAS	***94.62***
	4	ACMWDDM	***99.18***
STIDE: Sequence Length 6	5	ACM	***0.00***
	6	ACMWTR	***6.49***
	7	ACMWAS	***0.00***
	8	ACMWDDM	***0.00***
STIDE: Sequence Length 10	9	ACM	***0.03***
	10	ACMWTR	***0.06***
	11	ACMWAS	***0.04***
	12	ACMWDDM	***0.04***

Table 5.4: Precision: ACM Comparison Results

5.3.1 Further Investigation: STIDE Differentiation of HLA Federates

The very low percentage of anomalous behaviour identified by the STIDE technique for distinct ACM federate behaviours, as reported in Table 5.4, raises the question: Given a STIDE model trained on a HLA federate which implements a reasonable portion of possible HLA functionality, can the STIDE technique identify *any* different HLA federates' behaviours?

In order to answer this question, the model for the ACM federate, which implements portions of 5 of the 6 sections of HLA functionality[2], was compared with three completely unrelated and vastly different test federates:

1. The *FAQ* test federate utilised in the DSILI project (Wharington and Andrews, 2002).

2. A modified version of the *FAQ* test federate implemented to be as simple as possible.

3. The *HelloWorld* test federate provided with the DMSO RTI distribution.

Each of these federates' behaviour is substantially different from the ACM, and does not include interoperation with other federates. The FAQ federate creates a federation, joins, subscribes to an ob-

[2] The ACM model represents use of 'Federation Management', 'Declaration Management', 'Object Management', and 'Time Management', but not 'Ownership Management' or 'Data Distribution Management'.

Technique	Comparison	Behaviour	Anomalous %
LEAPBM	1	FAQ DSILI	***100.00***
	2	FAQ Low-Level	***100.00***
	3	HelloWorld	***100.00***
STIDE: Sequence Length 6	4	FAQ DSILI	***1.01***
	5	FAQ Low-Level	***1.07***
	6	HelloWorld	***0.13***

Table 5.5: Precision: STIDE HLA Federate Differentiation Results

ject class, advances time 10 times, then resigns and destroys the federation. The HelloWorld federate is slightly more complicated, in addition to the behaviour of the FAQ federate, it registers an object, and updates its attribute values as it advances time. Even the most complicated test federate, HelloWorld, is distinct from the ACM federates in numerous ways: publishing interactions, discovering objects, reflecting attribute values, sending interactions, and receiving interactions. The FAQ federate is further distinct from both: it does not publish an object class nor register objects.

Each of these federates' executions were compared using the STIDE model with sequence length of 6, as in the previous section this showed better precision than sequence length 10. The same comparisons were also performed using LEAPBM to confirm its ability to identify the large functional differences. The results are shown in Table 5.5.

As shown in Table 5.5, the STIDE PBM technique identifies the FAQ federate, whose functionality is vastly different from ACM, as only performing approximately 1% anomalous behaviour when compared with the model of the ACM federate. From these results, it would appear unlikely that the STIDE technique can precisely differentiate any HLA federate programs, from a model of a reasonable portion of HLA behaviour.

5.3.2 Average-Branching Factor

The average-branching factor metric, defined by Wagner (2000), is designed to provide an intuitive view of a PBM model's precision by highlighting the functional options available to an attacker during a program's execution. This is indicated by the amount of branching[3] that occurs during a finite

[3] Branching occurs when a single state has multiple transitions which lead to distinct states.

state automaton. The branching factor of a state X is the number of distinct states linked to X, by transitions whose source is X. The average-branching factor for a finite state automaton is then the sum of branching factors for all states divided by the number of states.

The average-branching factor can be easily compared for both the STIDE and LEAPBM techniques for the models of ACM behaviour. The STIDE technique calculates the average-branching factor internally and maintains its value with its models. For the LEAPBM technique the average-branching factor was calculated as per the definition described above. The results are shown in Table 5.6.

Technique	**Details**	**Avg. Branching Factor**
LEAPBM	Including unrestricted interactions	***12.31***
	Excluding unrestricted interactions	***1.31***
STIDE	Sequence Length 6	***1.73***
	Sequence Length 10	***1.50***

Table 5.6: Precision: Average-Branching Factor Results

Table 5.6 shows two average-branching factors for the LEAPBM technique. These factors either include, or exclude the set of potential order unrestricted interactions for each state in the interaction FSA. Order unrestricted interactions most typically constitute unimportant and unrestricted program behaviour[4], and thus, have no impact on the important (security related) program behaviour the average-branching factor is designed to represent. For this reason the average-branching factor for LEAPBM should exclude unrestricted interactions. Thus, these results indicates that the LEAPBM average-branching factor of 1.31 compares favourably with the two STIDE values of 1.73 and 1.50 for sequence lengths 6 and 10 respectively.

5.4 Discussion

The results provided in this investigation indicate that, when applied to High Level Architecture distributed simulation federate programs, the LEAPBM technique provides greater precision accuracy than the STIDE technique. For the comparisons between normal ACM federate functionality, and

[4] In this case they constitute helper HLA interface methods that have no impact on a simulation.

three modified ACM federate with anomalous behaviour, the STIDE technique identified an average *1.1%* anomalous behaviour. For the same comparisons the LEAPBM technique identified an average *92.88%* anomalous behaviour.

Additionally, using the average-branching factor metric, the LEAPBM technique's ACM model value of 1.31 constitutes an average improvement of *23.28%* over the STIDE technique's values of 1.73 and 1.50.

Only once did STIDE identify a substantial percentage of anomalies: using a sequence length of 6, and comparing a modified version of the ACM which is not time regulated resulted in *6.49%* anomalous behaviour identified (comparison 6). This apparently uncharacteristic precision of STIDE for this comparison is potentially explained, by the impact time regulation has on federate behaviour.

The simulation events of a federate that is not time-regulated are not delivered in time-order. Therefore, the causality of these events in terms of simulation time is not guaranteed, and simulation repeatability suffers. In order for a non-regulated federate behaviour to be consistent between multiple executions (such as the ACMWTR federate in the prior demonstrations) the execution ordering and timing of the federate must be controlled. In this investigation however, the execution ordering is purposefully varied to cover a range of acceptable execution orderings, as shown in Table 5.3.

The altered execution ordering and timing of the ACMWTR federate during the demonstrations effects the order in which the simulation progresses. This alters the behaviour of the ACMWTR federate and results in STIDE identifying an uncharacteristic high percentage of anomalous behaviour.

The improved precision accuracy of the LEAPBM technique, with appropriate LAE identification and when applied to HLA distributed simulation, indicates better identification of anomalous federate behaviours. This results in a lower rate of false-acceptance errors in a system that utilises the LEAPBM technique, and has potentially positive implications for industries using the HLA distributed simulation:

- The arduous verification procedure for HLA federates can be augmented by constant run-time checking to provide quality assurance.
- The potential for information theft is reduced.
- The integrity of simulated training and developments is improved.

Further investigation into the ability of STIDE to identify anomalies in any HLA federate's behaviour, given a model of reasonable HLA functionality, highlights the poor precision performance of STIDE within the HLA distributed simulation domain.

It appears reasonable to state that STIDE has difficulty identifying anomalous HLA behaviour. This is due to the unchanging definition of the HLA software libraries between federates, and the fact that the differences in their use, while obvious at the application-level, become highly obfuscated in the resulting low-level system-call sequences. As it is these sequences upon which STIDE bases its model, the resulting poor performance is not unexpected.

5.4.1 Limitations

This investigation has quantified the comparative precision accuracies of STIDE and LEAPBM in the identification of anomalous HLA federate behaviour, for the example ACM federate. Importantly, the LEAPBM models used in this investigation were created with appropriate LAEs to represent HLA concepts. The identification of appropriate LAEs is crucial to the performance of the LEAPBM technique, but cannot be guaranteed as it relies on the domain-specific expertise of the modeler. This is an important limitation to these results.

While HLA simulations can be considered distributed applications, the results of this investigation may not be echoed in other distributed applications. The HLA is relatively distinct amongst distributed applications in that it has no standardised communications format. Other distributed applications are potentially better modelled using other, network-based techniques.

The federates used in this investigation, while implementing a substantial portion of the available HLA functionality, are not of an industrial quality, and have not received official certification for HLA compliance. This was unavoidable given the prohibitive effort and cost involved in obtaining such compliance, as discussed in section 4.1.3. As a result, these results cannot be generalised to all HLA federates without extended investigation.

5.4.2 Extensions

Several extensions would greatly benefit this investigation:

- Investigation using certified, industrial HLA federates. This extension would require either

access to existing certified federates, or sufficient time and resources to develop and compliance new federates.

- Further investigation of the limits of the STIDE technique to HLA PBM. Specifically, investigating whether a STIDE model of a program's HLA functionality S, will match with any other HLA program utilising a subset of S.

These extensions would provide useful further insights into the precision accuracies of LEAPBM and STIDE, when applied to HLA distributed simulation federate programs.

Chapter 6

MODEL FOCUS INVESTIGATION

The accuracy of a program behaviour model depends in part, on its ability to correctly equate equivalent program behaviour with different implementations. This is termed *focus*. A well focussed model requires that the specification of a program interaction is general enough that any behaviour which produces an equivalent result will be equatable, no matter how different in programmatic definition. This chapter investigates the focus accuracy of the STIDE and LEAPBM PBM techniques.

Focus accuracy can be quantified by the frequency of false-alarm errors. These errors occur when a functionally identical interaction with definition differences is mistakenly identified as anomalous. This measure is also detailed in section 2.4.1.

In this chapter, the cause of model focus problems in the existing STIDE PBM technique is examined. The solution provided by LEAPBM is detailed and its implementation is explained, and then demonstrated. This demonstration is performed by the *Matis* architecture, on a series of HLA distributed simulation federate programs. Finally suggestions are made for extending LEAPBM's solution further to cover other types of programs.

6.1 Task Description

Ensuring that programs on a computer system perform only legal and safe behaviour improves security (Sekar et al., 2001b). This is the aim of anomaly-prevention systems, which make comparisons between observed program behaviour and models of verified safe behaviour.

The focus accuracy of a PBM technique depends on its ability to provide general definitions of behaviour, which cover the potentially numerous ways a given behaviour may occur in terms of program interactions. For low-level programs interactions, such definitions can occupy substantial space, and increase model complexity which reduces performance.

As previously discussed, poor performing PBM techniques result in increased latency for systems that apply them. Performance is important to anomaly-prevention systems as they often function

simultaneously with the programs they control. Thus, PBM techniques with poor performance are likely unusable in anomaly-prevention applications.

The basis-terms of the PBM technique used in the existing *pH* anomaly-prevention system are a program's interactions with the system kernel via system-calls. Typically for system-call-based PBM techniques, a substantial program behaviour requires multiple system-calls, as individual system-calls often perform atomic and unsubstantial functionality. There are potentially many groups and configurations of system-calls for achieving any given behaviour. For example, writing the string 'Hello World' to the screen could be achieved by a single call such as *write(0, "Hello World", 11);* , or by any series of calls individually printing parts of the string in sequence, such as *write(0, "He", 2); write(0, "llo W", 5); write(0, "orld", 4);*.

A well focussed system-call-based PBM technique requires an understanding of every potential group of system-calls which perform a program behaviour. This requirement increases the technique's complexity, as it ideally requires all distinct groups of system-calls to be represented for each program behaviour. This quickly becomes impractical for modern programs, which perform thousands of system-calls per second. The resulting increase in model complexity would result in a consequential loss of performance, and usability.

To maintain the performance required of anomaly prevention applied PBM techniques, not all functionally identical system-call sequences are guaranteed to be represented by the model. This incomplete definition of program behaviour limits the focus accuracy of system-call-based PBM techniques. The task for LEAPBM is to provide better focussed program behaviour models while maintaining performance.

6.2 Multiple Path Encapsulation

Current system-call-based PBM techniques, most often, do not maintain all potential combinations of interactions which represent a program behaviour, due to the performance requirements. The performance impact of maintaining these combinations is a result of the large number of such interactions made during a program's execution, and the potentially large number of combinations of system-calls which are equatable (Sekar et al., 1999). Some recently developed PBM techniques address this issue by optimising the maintenance of system-call interaction combinations about more abstract *events*

(Lin et al., 1998; Lin J., 1998; Ning et al., 2001). This maintains performance while allowing some combinations to be represented, but does not guarantee the total set of equivalent combinations is included. Generating and representing the complete set of functionally equivalent system-call sequences for a specific event is potentially intractable:

- Calls with no effective functionality can be added,
- Unimportant ordered calls can be reordered.
- Single calls can be divided into multiple equivalent calls.

The focus accuracy of system-call-based PBM techniques suffers as a result of not representing some or all equivalent interaction combinations. The task of improving focus accuracy while maintaining performance is fulfilled by LEAPBM as a result of its abstraction of low-level functionality.

LEAPBM is not forced to maintain the low-level interactions of a program. Instead, LEAPBM allows its basis-terms to be tailored to directly represent appropriately abstract behaviours. This abstraction can encapsulate many low-level interactions and thus, encapsulate their equivalent groups and combinations. The degree to which LEAPBM fulfils its task depends upon how well this encapsulation via abstraction results in the indirect representation of programs' equivalent low-level interactions.

6.2.1 Encapsulation via Abstraction

The LEAPBM technique enables the encapsulation of groups and combinations of system-call interactions by abstraction. A LEAPBM model's ideal basis, is the abstract concepts understood by a program that can be defined in terms of the software libraries used by the program. The LEAPBM technique maintains the complete set of equivalent abstract interaction combinations for a program's behaviour in order to provide focus accuracy, while remaining suitably simple. However, the larger the set of equivalent interaction combinations, the more complex and less usable a model becomes.

The number of possible combinations that result in equivalent behaviour depends, in part, on the number of individual interactions which compose a program's abstract behaviour. As demonstrated in the precision investigation (Chapter 5), LEAPBM reduces the number of program interactions which

define behaviour by using more abstract interactions as a model basis. The potential focus accuracy of the LEAPBM technique is thus potentially improved due to the encapsulation of equivalent low-level interaction combinations via abstraction.

The number of equivalent interaction combinations is also influenced by:

- The degree to which combinations of interactions can be *composed* into a single interaction with equivalent behaviour.
- The *divisibility* of single interactions into combinations of interactions with equivalent behaviour.
- The degree to which the *order* of interactions is important to the behavioural equivalence.

These effects on the number of abstract behaviour interactions represented by LEAPBM further determine its potential focus accuracy.

Interaction Composition

The degree to which a combination of program interactions can be composed, to form a single equivalent program interaction, is expected to be minimal. Such a composition would simplify the program's source code definition, and would likely have been identified during the design phase of the software's development process.

The unlikelihood of the composition of abstract interactions reduces the number of equivalent-behaviour interaction combinations for a LEAPBM model, and potentially improves its focus accuracy.

Interaction Divisibility

The behaviour of an interaction most often depends on the data it utilises, which take the form of specified parameters and program variables. The same interaction can perform different functions dependent on this data. An interaction is divisible if this data can be divided into smaller constituent portions, and used by other interactions to affect the same behaviour.

The degree to which abstract data is divisible is determined in part by the presence and importance of information inherent in the structure of the data. Structured data is commonly employed by modern object-oriented programs. Structured data is divisible down to a series binary values by a process commonly known as *serialising* or *marshalling* (Jordan and Russell, 2003). During serialising information about any structure of the data is potentially lost, such as the endianness of the data (Wikipedia, 2006c). The constituent serialised values of structured data are not necessarily equivalent to the data itself, due to this potential lost of format and structure information. The importance of this information loss determines if data with structure is divisible.

Information lost during the serialising of structured data can be considered unimportant if there exist other interactions which appropriately consume portions of the serialised data, to produce the original behaviour. These interactions complement the lost information through intrinsic understanding of the serialised data's structure and meaning. The presence of such interactions is determined by the specific abstract behaviours involved (provided by a software library) and cannot be predicted.

The divisibility of abstract interactions is potentially reduced over that of system-call interactions, due to the more common use of structured data and the resulting additional, but not necessarily met, requirements for divisibility. System-call interactions use only unstructured primitive values, with no inherent structure, which results in more likely divisibility.

The decrease in the divisibility of abstract interactions reduces the number of potential multiple definitions of behaviour required in a LEAPBM. This reduction allows for increased focus accuracy.

Order of Interactions

The importance of the order of individual interactions in constituting identical behaviour varies from critical to none, as previously introduced in sections 2.3.3 and 3.3.2. This variation affects the number of interaction combinations of equivalent behaviour, which must be maintained by a well focussed PBM technique. In any situation when the importance of the order of individual interactions is not critical, it is probable that multiple combinations of interactions will be required to represent them. Program behaviours in which the order of interaction is of critical importance, cannot be equivalently reordered. Representing critical ordered program behaviours requires only a single combination of interactions.

Moderate importance occurs when the order of a set of interactions is unimportant within a subgroup of program interactions. For example, a HLA federate can be considered to have identical behaviour, irrespective of the order it declares its object subscription and publication intents.

When the order of interactions is of no importance, it means interactions can occur at any point and have identical effects on program behaviour. For example, a HLA federate can legally request the RTI's integer handles for simulation objects at any point in its execution, subsequent to joining a federation execution.

The requirement to capture this varying degree of order importance often necessitates modifications to PBM techniques. The LEAPBM technique exemplifies this.

6.2.2 Order Importance Modifications

The abstract encapsulation of low-level program interactions in the LEAPBM technique reduces the need for multiple behaviour definitions, but does not remove it entirely. The abstract program interactions maintained by LEAPBM models, can occur in situations where their order is of moderate or no importance. In these situations multiple definitions are required to represent the abstract interaction combinations that result in a particular behaviour. These multiple definitions must be as concise as possible, in order to minimise their impact on a model's performance and usability.

LEAPBM models define sequence combinations of interactions as finite state automata. This means that the same interaction can be defined once as a state, with different sequences employing this interaction represented by various transitions to and from it. Take for example, the combinations of interactions: ABC, ACB, and ABB. The definition of these three combinations that would otherwise require 9 states only requires 3 states, and 5 transitions. This results in a likely saving of model complexity, as a smaller representative FSA is possible.

This is in contrast with the STIDE technique which redefines states. STIDE's redefinition of states simplifies the processing algorithm, and is not overly expensive as each state require only 1 byte of storage. A LEAPBM state requires substantially more than 1 byte for storage, as it stores more details of the interaction it represents.

Moderately Important Order

In some cases the order of program interactions is only moderately important. These situations occur when a block of interactions occur at a point in a program's behaviour, but the ordering of sequences within it is irrelevant to the resulting behaviour. LEAPBM provides a specific definition for this situation. Instead of all potential pathways to and from each sequence being explicitly defined, the interaction sequences are logically grouped together. Every such sequence within the block which has not been performed as required can then be performed before or after every other interaction. This reduces the number of pathways represented and the required LEAPBM FSA transitions.

Let n_G be the number of sequences in such a block, and p_{un} and p_{op} be the minimum number of pathways required for unoptimised and optimised models respectively. Then $p_{un} = n_G.(n_G + 2)$, because each interaction sequence has a pathway to each other interaction including itself, at least one pathway to and one pathway from an interaction external to the each sequence[1]. Contrastingly, $p_{op} = 2$ (for LEAPBM models), because only one pathway is required to the block, and one pathway from the block.

Unimportant Order

In some cases, the order of a program's interactions is irrelevant to the defining behaviour. These situations occur when interactions exist that can occur at any point in a program's execution. While it is likely that the majority of such interactions will have no impact on the behaviour of a program, in some cases they do. LEAPBM provides the 'unrestricted' flag for such interactions and maintains them separately to FSA-based structure. LEAPBM defines no pathways to such interactions, but checks for these calls during processing, failing the existence of explicit FSA transition pathways. This reduces the number of pathways defined in the model, but adds additional complexity to the model's processing algorithm.

Adopt the previous definition of p_{un} and p_{op}, let n_T be the total number of interactions defined by a model, and let n_U be the number of interactions whose order is unimportant. Then $p_{un} = n_T.n_U$, as each interaction in the model must have a pathway to each order-unrestricted interaction, and vice

[1] This is also the number of pathways which must be checked by the model processing algorithm in both optimised and unoptimised models.

versa. However, $p_{op} = 0$ (for LEAPBM models), because no explicit pathways to such interactions are required.

Let c_I be the increase in decisions required by the processing algorithm for stages in a LEAPBM model, and adopt the previous definition of n_U. Then $c_I \leq n_U$, because a decision regarding order-unimportant interactions is made only if explicit pathways from the current state are exhausted, and only until a matching interaction is found. Thus $c_I < p_{un}$ and the overall result is a reduction in complexity.

6.3 Demonstrations

This investigation demonstrates the comparative focus accuracy of the existing STIDE technique and the LEAPBM technique. In order to test the focus accuracy of STIDE and LEAPBM techniques, a particular federate is modified to perform its function in a number of different ways. These modifications are performed both on the order of unimportant interactions and interaction divisibility.

For this experiment, the ATO test federation's ACM was modified in three distinct ways:

ACM With Individual Updates : This modification to the ACM federate, which results in a single federate version termed *ACMWIU*, is an example of interaction divisibility. The ACM federate is responsible for updating 13 attribute values of aircraft object instances, which it normally achieves by a single *RTI::updateAttributeValues* call. The ACMWIU version updates the attribute values with 13 distinct *RTI::updateAttributeValues* calls. This has no effect on the overall functionality of the federate, but does change its interactions.

ACM Publication-Subscription-Regulation-Constrained : This modification to the ACM federate results in four federates, each of which performs two groups of unimportant interactions in different orders. These interactions are the federate's declaration of publication and subscription intents, and the federate's enabling of time regulation and constraint. This results in four federate versions named after the order in which these interactions occur:

[PSCR]: Publication, subscription, constrained, regulation.

[PSRC]: Publication, subscription, regulation, constrained.

[SPCR]: Subscription, publication, constrained, regulation.

[SPRC]: Subscription, publication, regulation, constrained.

ACM Handle Fetching Every time : This modification to the ACM federate, which results in a single federate version termed *ACMHFE*, is an example of unimportant interaction ordering. The integer handles for object classes, attributes and interactions are determined by the RTI at runtime. A federate queries the RTI for the value of these handles, which can subsequently either be stored and accessed locally (as occurs in most federates), or discarded and re-queried when next they are needed. This ACM version re-queries the RTI each time it requires a handle for an object class, attribute, or interaction class.

These three modifications are made on functionally unimportant implementation specifics (the ordering of unimportant interactions and the divisibility of interactions based on data) and do not affect federate behaviour. To demonstrate the comparative focus accuracy of the LEAPBM and STIDE techniques, each of these federates is compared with the model for the ACM federate. The design of this investigation is further discussed below.

Design

It is hypothesised, that the flexible nature of the LEAPBM technique will allow it to more correctly focus on the equivalent ACM federate versions than the STIDE technique. In order to test this hypothesis, models of the ACM federate are generated for both STIDE and LEAPBM. The comparisons of each federate version's execution behaviour, with the ACM model, are then observed for each technique as shown in Table 6.1. It is expected that the LEAPBM technique will exhibit better focus accuracy, and correctly equate the functionally identical ACM federate versions.

Observations 1, 8 and 15 are control observations, which compare the same model and execution behaviours to ensure the PBM technique is functioning correctly. Each of these observations should result in 0 anomalies identified.

Technique	***Comparison***	***Execution Behaviour***
LEAPBM	1	ACM
	2	ACMWIU
	3	ACMPSCR
	4	ACMPSRC
	5	ACMSPCR
	6	ACMSPRC
	7	ACMHFE
STIDE: Sequence Length 6	8	ACM
	9	ACMWIU
	10	ACMPSCR
	11	ACMPSRC
	12	ACMSPCR
	13	ACMSPRC
	14	ACMHFE
STIDE: Sequence Length 10	15	ACM
	16	ACMWIU
	17	ACMPSCR
	18	ACMPSRC
	19	ACMSPCR
	20	ACMSPRC
	21	ACMHFE

Table 6.1: Focus: Observations of ACM Model Comparisons

Method

The investigatory observations specified in Table 6.1 are performed on the computer detailed in Appendix B, using the ATO HLA federation detailed in Appendix A.1. The LEAPBM and STIDE models generated for the precision investigation are reused for this investigation. For more information on their generation, refer to section 5.3. The LEAPBM and STIDE models themselves are present in Figures 7.1 and 7.2 respectively.

Results

The results observed for the comparisons listed in Table 6.1 as shown below in Table 6.2.

From Table 6.2, comparisons 1, 8 and 15 correctly show 0 anomalies detected during the comparison between a federate and its model for both the LEAPBM and STIDE PBM techniques. Compar-

Technique	Comparison	Behaviour	Anomalous %
LEAPBM	1	ACM	***0.00***
	2	ACMWIU	***0.00***
	3	ACMPSCR	***0.00***
	4	ACMPSRC	***0.00***
	5	ACMSPCR	***0.00***
	6	ACMSPRC	***0.00***
	7	ACMHFE	***0.00***
STIDE: Sequence Length 6	8	ACM	***0.00***
	9	ACMWIU	***0.01***
	10	ACMPSCR	***0.01***
	11	ACMPSRC	***0.01***
	12	ACMSPCR	***0.01***
	13	ACMSPRC	***0.01***
	14	ACMHFE	***0.01***
STIDE: Sequence Length 10	15	ACM	***0.00***
	16	ACMWIU	***0.04***
	17	ACMPSCR	***0.03***
	18	ACMPSRC	***0.02***
	19	ACMSPCR	***0.03***
	20	ACMSPRC	***0.03***
	21	ACMHFE	***0.03***

Table 6.2: Focus: ACM Comparison Results

isons 2-7, 9-14 and 16-21 indicate that the focus accuracy of both the LEAPBM and STIDE techniques is virtually identical. Both techniques correctly identified a low percentage of anomalous behaviour, for modified ACM federate versions with identical defining behaviour.

It should be noted that the LEAPBM technique identified exactly 0% in all cases, while there was some variance in the STIDE technique's results, with around 3 times as much incorrect identification of anomalous behaviour for STIDE of sequence length 10, as compared with the more commonly used sequence length of 6.

These results do *not* support the hypothesis that the LEAPBM technique is able to better focus on defining HLA federate behaviour and ignore implementation differences, than the STIDE technique. They show that both the STIDE and LEAPBM technique's perform well at equating equivalent HLA federate behaviours with different implementations.

6.4 Discussion

The results provided in the previous section indicate that, when applied to simple HLA distributed simulation federate programs, both the LEAPBM and STIDE techniques provides excellent focus accuracy. These results do *not* indicate however, that the LEAPBM technique provides substantially better focus accuracy than the STIDE technique.

The excellent focus accuracy of both the LEAPBM (with appropriate LAE identification) and STIDE techniques, when applied to HLA distributed simulation, can potentially result in a low false-alarm rate for applications.

The focus accuracy of the LEAPBM technique, and its use of a FSA structure to represent the sequence and order of program interactions, supports the claim by Wagner and Dean that such models produce no false-alarms (Wagner and Dean, 2001).

The unexpected focus accuracy of the STIDE technique is logical when taking into consideration the results of the prior precision investigation. The demonstrated inability of STIDE to distinguish between distinct HLA federate behaviours, could also cause excellent focus accuracy results. Modifications to the federates which do not cause functional distinctions are likely to be less severe than those that do. Thus, as modifications that do cause functional distinctions are not identified as anomalous (as shown Chapter 5), it is reasonable that less severe modifications are similarly not identified.

6.4.1 Limitations

This investigation has quantified the comparative focus accuracies of STIDE and LEAPBM in the identification of equivalent HLA behaviour, for the example ACM federate. This investigation employs the same federates and test environment as the previous precision investigation, and is bound by the same limitations. These limitations are described in section 5.4.1.

6.4.2 Extensions

This focus investigation could be further extended using certified HLA federates, to confirm that the trend of high focus accuracy for both STIDE and LEAPBM techniques applies to HLA federates in general. As discussed in section 4.1.3, the use of certified HLA federates in this investigation was not possible due to the prohibitive effort and cost required.

An additional investigation could also examine the non-HLA functionality of the federate, and its potential impact on the focus accuracy of these techniques when modelling HLA behaviour. For example, two versions of a certified HLA federate could be created that load and save initialisation data in different ways. This difference would have no effect on the HLA functionality of the federate, therefore the execution behaviours of both versions should match a single model of program behaviour.

It is possible however that the STIDE technique may incorrectly identify anomalies between two such federate versions, due to potential differences between the usage of the system-calls necessary to load and save data.

Chapter 7

MODEL COMPLEXITY INVESTIGATION

The complexity of a PBM technique is an important factor in determining its usability. A technique's complexity is indicated by the computational space required for storage, and the computational processing required to both generate, and make comparisons with, a model. A modelling technique that is overly complex becomes unusable by requiring excessive computational storage space or computational processing time. This chapter investigates the complexities of the STIDE and LEAPBM PBM techniques.

In this chapter, the levels of complexity for the existing PBM technique and the LEAPBM technique are quantified. The results indicate whether the increased accuracy of the LEAPBM technique[1] occurs at the expense of increased complexity, and decreased performance. Comparisons between the complexities of the two techniques are demonstrated both theoretically and practically for models of HLA federate behaviour.

7.1 Task Description

Typical anomaly-prevention systems operate simultaneously with the programs they monitor, and LEAPBM and STIDE are no exceptions. A PBM technique can become unusable if its complexity significantly affects the performance of the application employing it. The complexity and performance of both the STIDE and LEAPBM techniques is the topic of this chapter.

The measurement of the complexity of a PBM technique depends on the application which utilises it. All existing work applying the STIDE technique utilises computer software as the sole consumer of models, due perhaps to the large number states for the finite automaton[2] and the little meaning inherent in each state (Lunt, 1993).

---

[1] As demonstrated in Chapter 5.

[2] A sample HLA federate execution produced a finite automaton of over 2000 states as detailed in Appendix C.2.

A driving motivation of this study is to make program behaviour models more easily understandable by a human, in order to augment the decision making processes in the fields that employ PBM, such as computer security. Thus, in this investigation the complexity of a PBM technique is investigated for both software and human consumers.

As discussed in Chapter 2, computational complexity can be practically measured in two ways: the time taken to perform functions, and the storage space required for them. These measurements can be taken respectively by: the time taken by a PBM application, and the storage space occupied by a PBM model definition. Additional theoretical asymptotic analysis can be performed to augment these practical examples, and give a view of the potential growth of complexity as the programs represented grow.

The human interpreted complexity of a PBM model is subjective and much more difficult to quantify than computational complexity. A brief discussion of the human comprehension for LEAPBM and STIDE provides an intuitive comparison.

Chapters 5 showed improved accuracy for the LEAPBM technique compared to the existing STIDE PBM technique. An increase in accuracy can often lead to an increase in complexity and consequential reduction in usability. While the existing system-call based STIDE technique is of small enough computational complexity to be applied to an anomaly-prevention application, its human comprehensibility is unknown. The task for the LEAPBM technique is to ensure its computational complexity is low enough to be usable, while attempting to improve the ease with which a human can understand a program's behaviour from its model.

7.2 Small and Fast Models

The STIDE system-call-based PBM technique has been shown to be practically usable for real-time detection of anomalous program behaviour (Somayaji and Forrest, 2000, 2002). This practical implementation demonstrates STIDE's acceptable computational complexity in terms of both storage space, and processing time. Although based on low-level system library function calls that occur in large quantities, STIDE produces smaller models of program behaviour by defining fixed-length sequences, and discarding all arguments to the system-calls.

The LEAPBM technique enables the arguments of abstract interactions to be represented, which

would logically increase its complexity due to more information storage and processing required. However, by allowing a model to take advantage of program abstractions, the number of program interactions a LEAPBM technique represents can be much smaller. This reduction in the number of program interactions modelled depends upon the appropriate selection of LAEs as the basis-terms in which program behaviour is represented, and the inherent complexity of the program.

The reduction in the number of program interactions, enabled by LEAPBM's flexible definition of basis-terms, is expected to *decrease* the required processing time and storage space[3]. However, the storage of interaction details, such as argument values, is expected to *increase* the processing time and storage space. Thus, STIDE performs simple processing for a high number of program interactions, while LEAPBM performs more complex processing for a typically smaller number of interactions. This trade off is expected to result in similar overall computational time complexity.

An additional factor which primarily concerns the comparative space computational complexity between STIDE and LEAPBM, is the definition of LEAPBM's LAE basis-terms. This has no equivalent in the STIDE technique as it uses a static map, provided by the operating system, to distinguish between interactions. A LEAPBM model defines LAE class basis-terms, and how they occur in terms of software library elements. This additional requirement increases the computational space complexity of LEAPBM models. However, a set of LAE class definitions can be reused between models, and thus, the increase in size complexity by including the LAE definitions, is shared by all the LEAPBM models that use it. In the domain of HLA distributed simulation, many thousands of LEAPBM models can make use of the set of HLA LAEs. Thus, $\lim_{x\to\infty} f(x) = 0$, where $f(x)$ is the increased space complexity as a result of HLA LAEs, and x is the number of HLA LEAPBM models.

Thus, STIDE maintains small amounts of information for a large number of generic program interactions, while LEAPBM stores more information for a smaller number of interactions. This trade off is also expected to result in similar overall computational space complexity.

7.3 Human Comprehension

An expected benefit of the LEAPBM technique is an increase in ease with which a model of program's behaviour can be comprehended by a human. Previous computer security PBM techniques have not

---

[3] Specifically in security applications where each program interaction is compared with the model.

aimed at making a model of a program's behaviour humanly comprehensible.

It is proposed that enabling such comprehension could provide useful benefits to fields that apply PBM techniques. In the case of computer security, human comprehension could enable a human being to augment existing processes for examining the security impact of a program, such as anomaly-prevention (Necula, 1997; Necula and Lee, 1998; Sekar et al., 2001b). Enabling the human verification of program security could make this process more transparent and alleviate dependence upon software testing tools.

It is expected that the reduction in the number of program interactions in a LEAPBM model can enable a human to gather an overall understanding of the program's behaviour. It is also expected that the addition of details to a LEAPBM model will further enable human comprehension by providing sufficient details to distinguish and understand individual program interactions. This expectation will be briefly discussed for several models in the following demonstrations.

7.4 Model Generation

Models of program behaviour are typically generated either: manually by a human, or automatically by a computer program. The trend among recent techniques has been towards automated development in the interest of usability: automated model generation simplifies the requirements of PBM applications and make them more usable.

STIDE model generation is automated. An execution of a federate is examined using a modified version of the Linux *strace* utility which produces a STIDE formatted input file, which is then converted to an internal database format by STIDE. However, the generation of STIDE models is more complicated for HLA federate programs, than for the typical stand-alone UNIX programs it has previously been utilised for, such as *sendmail* and *lpr*. Due to the distributed nature of a HLA federation, every other HLA federate must also be executed in a controlled manner for each training execution.

The generation of LEAPBM models is a hybrid of both automatic and manual: the vast majority of the LEAPBM generation can be performed automatically, but the identification of new LAEs, and any optimisations or generalisations to the model, require expert human input and are therefore performed manually. Complete HLA functionality requires the definition of 10 LAEs, of which 7 are defined for these investigations. Refer to Figure 4.2 for the definition of these LAEs.

It is expected that the additional complication to the automated generation of STIDE models, caused by the distributed nature of the HLA, will increase its complexity to a level comparable to that of LEAPBM's semi-automated model generation procedure.

7.5 Demonstrations

This investigation demonstrates the comparative complexity of the existing STIDE technique and the proposed LEAPBM technique. These demonstrations examine practical results gathered from the application of the LEAPBM and STIDE techniques in the previous precision and focus investigations. These investigations utilised multiple versions of the Aircraft Manager (ACM) federate, which is part of the Air Traffic Operations (ATO) federation. For more information on this federate refer to the previous investigations in sections 5.3 and 6.3, and the full ATO federation description in Appendix A.1.

The demonstrations also examine the theoretical asymptotic complexity, human comprehension complexity and model generation complexity of the LEAPBM and STIDE techniques.

Design

It is hypothesised, that the reduction in the number of program interactions in a LEAPBM model will offset the more detailed information that is recorded for each interaction, and result in complexity that is comparable to the STIDE model of the same program. In order to test this hypothesis, the processing time and storage space utilised by the LEAPBM and STIDE techniques for the previous precision and focus investigations are compared. The use of the STIDE and LEAPBM techniques in these investigations is shown in Tables 6.1 and 5.2.

It is expected that the LEAPBM and STIDE techniques will exhibit approximately equivalent complexity, and not substantially affect the run-time of a program or occupy substantial storage space. It is also expected that the LEAPBM technique will exhibit complexity growth that is comparable with STIDE (and other) PBM techniques.

It is also expected that LEAPBM models are more easily human comprehensible than STIDE models. Thorough and correct measurement of human comprehensibility of program behaviour models is beyond the scope of this study, however an explanation is provided of the improvements intuitively expected for the LEAPBM technique.

Finally, it is expected that model generation of HLA federate programs for the LEAPBM technique is approximately as complex, and usable, as that for the STIDE technique. This hypothesis will be tested by the brief qualitative discussion, of experiences with generating the ACM federate models for the two PBM techniques.

Method

All observations conducted for this, and the previous precision and focus investigations were performed on the computer detailed in Appendix B, using the ATO HLA federation detailed in Appendix A.1.

The STIDE and Matis applications include program functions to identify anomalous behaviour, and other program constructs required to support this identification: the maintenance of the model in memory, the loading of the model from a file, and capturing and returning program behaviours. Only the functions responsible for identifying anomalous behaviour are measured in this investigation. For example, the Matis implementation of the LEAPBM processing algorithm outlined in section 3.6. The asymptotic analysis of the complexity of the two PBM techniques, also focuses solely on these functions as their effect is most pronounced during program model growth.

The measurement of the algorithms for anomalous behaviour identification, was performed by summing the number of CPU cycles which elapsed during the program function, and controlling all other processes executing on the computer. For more details on the technique used to measure the number of elapsed CPU cycles, refer to Appendix A.3.

The models used for this investigation were generated for the precision investigation, by the procedure described in section 5.3. These generation procedures are also qualitatively compared in the following results.

Results

The results observed for the comparisons listed in Tables 5.2 and 6.1 are shown below in Table 7.1.

From Table 7.1, the LEAPBM detection times were longer than both the STIDE with sequence length 6, and STIDE with sequence length 10. However all three techniques are insubstantial in terms

	LEAPBM	STIDE 6	STIDE 10
Behaviour	*Total Detection Times* (sec)		
ACM	0.28	0.06	0.09
ACMWTR	0.17	0.07	0.09
ACMWAS	0.17	0.06	0.09
ACMWDDM	0.29	0.07	0.08
ACMWIU	0.60	0.07	0.10
ACMPSCR	0.18	0.07	0.09
ACMPSRC	0.25	0.07	0.10
ACMSPCR	0.20	0.07	0.10
ACMSPRC	0.20	0.07	0.10
ACMHFE	60.35	0.07	0.09
	Avg. Increase in Execution Time (%)[4]		
ACM Federate	0.0053	0.0014	0.0019

Table 7.1: Complexity: Detection Time of ACM Behaviour

of the percentage of total ACM federate execution time.

One interesting result is the LEAPBM's detection time for 'ACM Handle Fetching Every-time', which is substantially larger than all the other federate versions. After review of the LEAPBM processing algorithm, this was discovered to be a result of the distinction between order unrestricted interactions, and those interactions which form part of the internal LEAPBM FSA. The LEAPBM processing algorithm checks the interaction FSA first and then, if the observed interaction does not match, checks the order unrestricted interactions. For the intentional but atypical increased number of unrestricted (HLA handle fetching) interactions in ACMHFE, checking the interaction FSA before the unrestricted interactions incurs a heavy penalty.

This is acceptable for two reasons:

1. The ACMHFE federate is unusually modified, specifically for the purposes of this investigation. These modifications would not occur in a functional HLA simulation, as they needlessly reduce performance. This point is highlighted by examples from the DSMO RTI programmers guide DMSO (2002) and emphasised by performance considerations in the DMSO Federation Checklists DMSO (1999).

[4] The ACMHFE results not included in average % of execution time result.

2. The majority of HLA federate interactions made, beyond initial configuration, are not unrestricted. While it would be possible to check for matches with unrestricted interactions before the FSA structure, this would have a greater impact on the performance of most federates during simulation.

Disregarding the ACMHFE federate detection times, these results show that the tested STIDE techniques and the LEAPBM technique add less than 0.006% of the ACM federate program's execution time to detect anomalies. This supports the hypothesis that the LEAPBM technique exhibits approximately equivalent complexity to the STIDE technique, and insignificantly affect the run-time of a program.

As additional verification of the reasonable time complexity for LEAPBM models, Feng et al. (2004) report average system-call verification times for the more recent Dyck model for several programs: *htzipd*, *gzip*, and *cat*. On average, the Dyck model took 300, 415, and 289 microseconds to verify each system-call by these programs respectively. Comparatively, the LEAPBM model for the ACM federate takes on average 816 microseconds to check each HLA software library call. Although the LEAPBM checks appear to take substantially longer for each call, the flexible abstraction it enables means the number of calls that need to be checked is greatly reduced. This reduction factor is anywhere from 75:1, as demonstrated in Appendix C.2, to 538:1, as observed during generation of STIDE ACM model. Taking this reduction into account, these results indicate that compared to the most recent development in system-call PBM techniques, the LEAPBM technique provides commensurate time complexity throughout the processing of an entire program.

The results for the storage space comparison for the STIDE and LEAPBM techniques were generated with the use of a standard compression utility. The model storage formats of the STIDE and LEAPBM techniques are distinct and have different goals. The LEAPBM technique's XML storage of models is designed to enable future compatibility and promote data format interoperability. The sizes of the models for these two techniques are therefore not directly comparable. To determine comparable, format-independent storage requirements, each model was compressed using Lempel-Ziv coding[5]. Table 7.2 shows the size of the models for the ACM federate for both STIDE and LEAPBM.

From Table 7.2, the storage space required for the LEAPBM technique, excluding the LAE class

[5] As implemented in the GNU *gzip* utility.

Technique	**Gzipped Size** (bytes)
LEAPBM	***5722***
LEAPBM: Including LAE Classes	***14158***
STIDE: Sequence Length 6	***5916***
STIDE: Sequence Length 10	***128665***

Table 7.2: Complexity: Compressed Size of ACM Models

definitions as per previous discussion, is slightly smaller than the STIDE with sequence length 6 model. The largest model is the STIDE with sequence length 10 model. These results also show that even if the HLA LAE class definitions are included in the LEAPBM model size, it is still 9 times smaller than a STIDE model with sequence length 10.

In terms of the amount of storage available on a typical computer system however, all the models are insubstantial. These results support the hypothesis that the LEAPBM technique exhibits approximately equivalent complexity to the STIDE technique, and that both occupy insubstantial storage space.

Theoretical analysis of the LEAPBM technique's processing algorithm, indicates that the complexity of processing models which employ the same LAE basis terms has $O(i{\times}p)$, using the following definitions: let i be the number of LAE interactions made by the program, and let p be the total number of program interactions with LAE properties. This is a linear big-O value, indicating that they growth in complexity of LEAPBM processing is proportionate to the growth of the model.

The big-O value for system-call-based FSA models[6]: $O(n)$, reported by Wagner and Dean (2001), where n is the program's number of system-call interactions. The theoretical complexity of the STIDE technique's processing algorithm is also of $O(n)$ order, where n is the number of nodes in the FSA.

Thus, both LEAPBM and STIDE processing algorithms have linear orders of complexity.

[6] Also called 'callgraph' models.

7.5.1 *Human Comprehensibility*

In terms of human comprehensibility, a program's model is considered in its entirety, given the likelihood a human would have little difficulty comprehending individual program interactions[7]. It is envisioned that human comprehension of an entire program behaviour model could also allow users to check the correct behaviour of computer software.

Human checking of computer software behaviour is an exciting possibility. It could allow harnessing a human being's intelligence, and pattern matching abilities, to augmenting existing computer security processes, such as the identification of anomalous program behaviour. This ability has not been proposed before, and would constitute an innovative contribution to the field of computer security.

It is expected that the LEAPBM technique will offer improvements in human comprehensibility, due to the smaller sized models expected as a result of enabling the flexible abstraction of program interactions. The extent of this reduction in size is dependent on the appropriate level of abstraction by the model's basis-terms.

To demonstrate that the LEAPBM technique (with appropriately abstracted basis-terms) results in smaller models than the STIDE technique, both technique's models for the ACM federate were automatically translated into FSA diagrams. These diagrams are shown in Figures 7.1 and 7.2.

Figure 7.2 for the STIDE technique, is zoomed out significantly to illustrate the overall complexity of the model. The STIDE model (sequence length 6) is comprised in total of 6706 states, which occur in 45 distinct automata. The LEAPBM model is comprised of 49 states, which occur in 18 logically distinct blocks. For the example ACM federate investigated in this study, the sheer number of states in the STIDE model is intuitively expected to be prohibitive to human comprehensibility, while the LEAPBM model is comparatively much simpler.

Additionally, the LEAPBM model identifies the LAE instance and the name of the program interaction which constitute each FSA state. Appropriate naming of the LAE classes during the model generation can result in informative names for the resulting LEAPBM model interaction states, as illustrated in Figure 7.1. Contrastingly, the STIDE technique names each FSA state according to the

---

[7] Ease of human understanding individual program interactions is probable as it is necessary for the program's development.

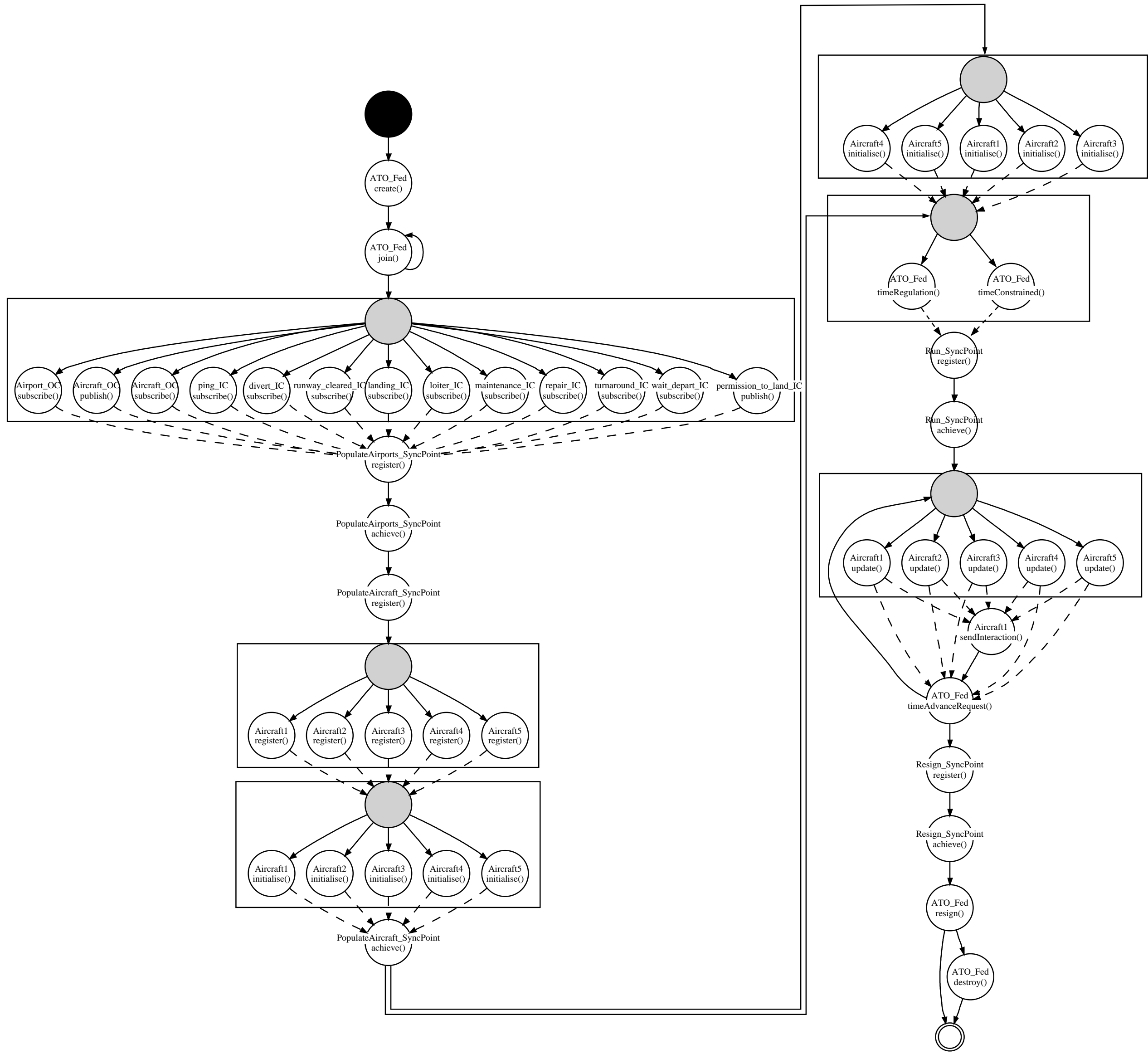

Figure 7.1: LEAPBM ACM Finite State Automaton

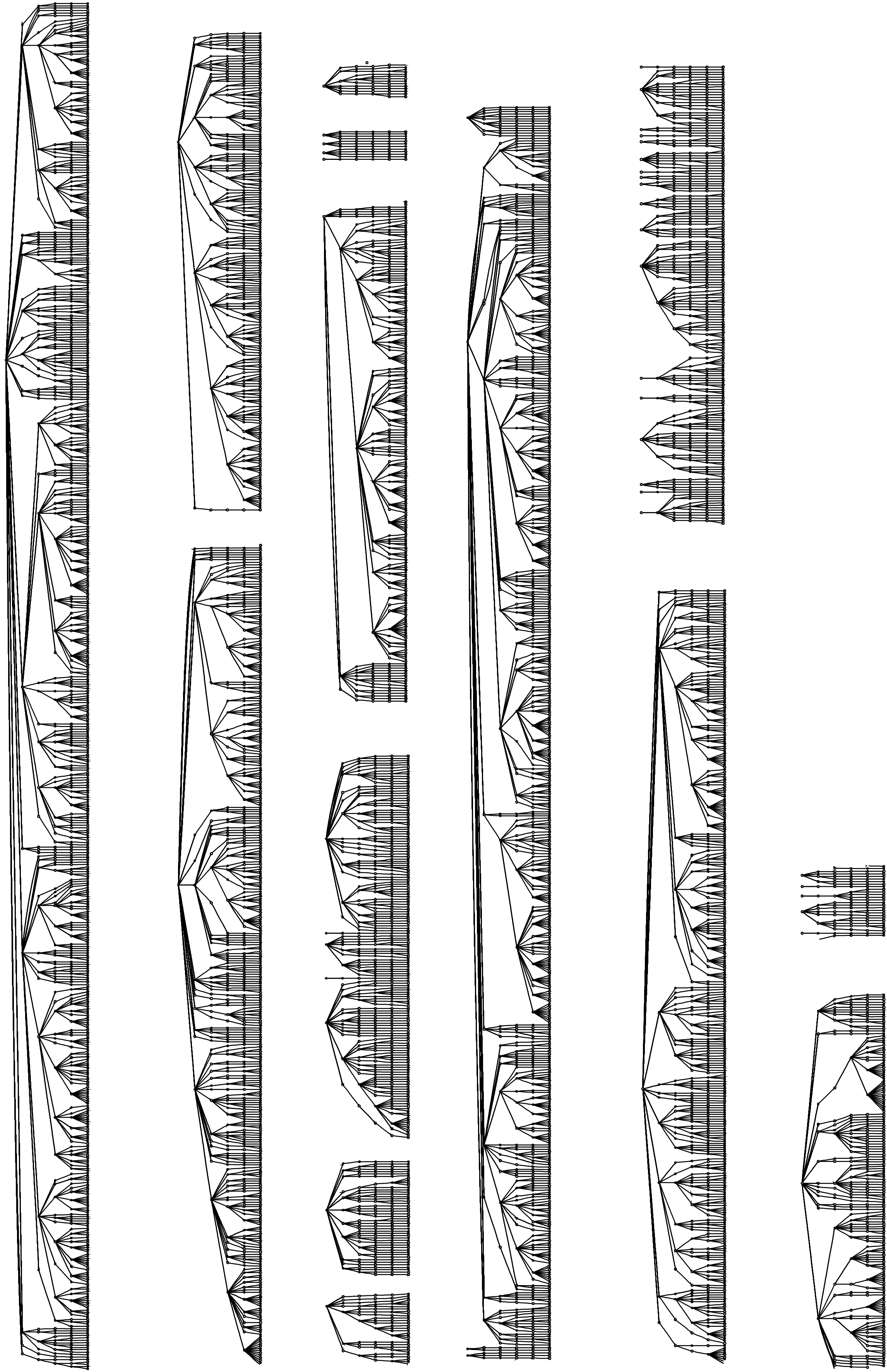

Figure 7.2: STIDE Sequence Length 6 ACM Finite State Automata

internal operating system number assigned to each system-call interaction, providing little indication of the behaviour STIDE models represent.

The substantially smaller number of interactions represented in a LEAPBM model, combined with the more informative naming of interactions, intuitively suggests that the LEAPBM technique enables more easily human comprehensible models than the STIDE technique. Conclusive investigation of this hypothesis is beyond the scope of this study.

7.5.2 Model Generation

Comparisons between the complexities of the model generation procedures for the STIDE and LEAPBM techniques, are unfortunately not straightforward given the difference in automation between the two. The STIDE technique's generation procedure is automatic but occurs at run-time, necessitating the manual manipulation of the run-time environment. The LEAPBM technique's generation procedure, is semi-automated (and currently unimplemented) and occurs prior to run-time, based on static analysis of the program definition (source code).

STIDE Generation

A STIDE model of the ACM federate's behaviour is generated for the precision and focus investigations. This involves a number of distinct run-time execution traces, as shown in Table 5.3. For each distinct execution trace the following steps are required:

1. Start the HLA Run-Time Infrastructure program.

2. Wait for the HLA Run-Time Infrastructure program to initialise.

3. If the ACM federate was to create the federation execution start it, otherwise start the Air Traffic Control federate.

4. Wait for the federation execution to be created.

5. Start the remainder of the federate programs.

6. Wait until all federate programs have initialised themselves.

7. Begin the federates simulating in the appropriate order.

These steps are encapsulated in a Perl script program, which enables them to be executed repeatably and automatically. However, development of this script to control the executing timing of multiple programs is time consuming. Additionally, it is required to be executed numerous times before model convergence is obtained: approximately 102 times for sequence length 6, and 140 for sequence length 10. For this study these executions each take over 8 hours on the laptop computer detailed in Appendix B.

The ATO federation is small enough that it can be run on a single computer. However the limitation of single computer power, and its driving motivation for distributed applications (such as the HLA), means this is unlikely to always be the case. Training a STIDE model for a HLA federate that requires additional computers is expected to be more complicated.

LEAPBM Generation

The LEAPBM model generation procedure, as detailed in section 3.5, requires manual effort by the model developer to identify the LAEs to use as basis-terms in defining a program's behaviour. Importantly, this manual task is only required each time new LAEs are identified. For example, the HLA LAEs identified and defined during generation of the ACM LEAPBM model could be used to automatically generate LEAPBMs for other HLA federate programs.

It is potentially cumbersome that the LEAPBM technique requires manual effort in some cases. However, this manual effort has parallels to the manual decisions which must be made with the STIDE technique. That is, the STIDE technique requires the user choose the sequence length specification. This choice, though much simpler than defining LAEs, also has an effect on the performance and success of the STIDE technique.

A positive potential result of more intuitive human comprehensibility of the LEAPBM technique, is that it allows both human verification and optimisation of the model after it has been generated. Human verification provides an additional level of assurance regarding the correctness of the model. While human optimisation of a LEAPBM model can potentially involve the generalisation of model attributes, such as the interaction ordering and LAE usage data values, to be more representative of those potentially occurring in future program versions.

Comparison

Although the STIDE technique is automated, for the HLA distributed simulation test domain of these investigations, it necessitates additional effort to control the execution sequencing of the other simulation components. During this study, the development of a STIDE model for the ACM federate took approximately 60 man hours, including the development of a script to control execution sequencing, and the amount of run-time execution required to reach model convergence.

The LEAPBM technique's generation algorithm has not yet been implemented in the Matis architecture, thus requiring completely manual definition of the HLA software library elements, the HLA logical abstract entities, and the ACM federate behaviour. Given reasonable expertise in the HLA distributed simulation test domain and access to source code of the ACM federate, this procedure took approximately 65 man hours, during the course of this study.

Of these 65 hours however, only approximately 10 were spent developing the HLA LAEs. This would indicate that had the LEAPBM technique's generation algorithm's automated steps been implemented in Matis, the time required to develop the LEAPBM model of the ACM federate would have been only 10 man hours. Additionally, models for subsequent federates being automatically generated with no manual effort required.

This brief qualitative comparison of the experiences with generating both STIDE and LEAPBM models for the ACM federate supports the hypothesis that the generation of HLA federate programs for the two techniques are of approximately equal complexity, and usability.

7.6 Discussion

It appears from these demonstrations that the LEAPBM technique does not place a greater burden on computational resources, than does the STIDE technique. These demonstrations also intuitively suggest improvements to human comprehensibility for the LEAPBM technique. Each of these aspects affect the overall usability of a PBM technique.

The practical time complexities of the two techniques are both less than 0.006% of the program's execution time, while the models for the techniques can occupy less than 6Kb of storage space[8]. This indicates approximately equivalent practical complexity. The fact that the complexities of both

[8] Using the most common STIDE sequence length value, which is 6.

the STIDE and LEAPBM techniques' is of $O(n)$ order, is reasonably expected to result in usable LEAPBM applications, such as exist for STIDE (Somayaji and Forrest, 2002).

The generation procedure employed for the LEAPBM technique in this investigation was not automated, as the algorithm defined by the LEAPBM specification has not been implemented. Even with the additional tedious manual tasks required, the total time taken to produce a LEAPBM model is comparable with that required by the STIDE technique, although the STIDE technique was not designed for examining programs that interact with multiple distributed software components. Additionally, control of geographically disparate programs would further complicate the STIDE generation procedure.

An initial intuitive comparison suggests that the LEAPBM technique is more comprehensible by a human than the STIDE technique. This is partially indicated by comparisons of the size of the models: the LEAPBM FSA contains 49 elements in 18 logical sub-groups, compared with the STIDE FSA which has 6706 elements within 45 logical sub-groups.

7.6.1 Limitations

This investigation has examined the comparative complexities of STIDE and LEAPBM in practice, generation, and human comprehension. The practical results in this investigation were gathered using the same federates and test environment as the previous precision and focus investigations, and are bound by the same limitations. These limitations are described in section 5.4.1.

7.6.2 Extensions

Similar to the previous precision and focus investigations, this investigation could be suitably extended using certified HLA federates to confirm that the approximate equivalence of the STIDE and LEAPBM techniques' complexities, applies to HLA federates in general. As discussed in section 4.1.3, the use of certified HLA federates in this investigation was not possible due to the prohibitive effort and cost required.

This investigation provides some intuitive indications of the human comprehensibility of the STIDE and LEAPBM techniques, based on personal experience. A more thorough empirical study is necessary before any conclusive statement about their relationship can be made.

Further investigation of the LEAPBM generation algorithm's complexity using a computer software implementation, would provide useful information and enable a more indicative comparison. During this investigation, the generation algorithm had not been implemented, and model generation was performed manually.

Chapter 8

CONCLUSION

This study proposes a new concept for the modelling of program behaviour: flexible abstraction. Previous efforts concentrate on fixed concepts at varying degrees of abstraction, such as the audit data provided by an operating system, or the program function calls to the operating system kernel. The flexible abstraction technique proposed by this study is termed LEAPBM.

This study investigated the performance of this proposed technique compared with the existing, freely available, STIDE technique. The STIDE technique bases its program behaviour models on operating system kernel function calls and its models are defined as finite state automata. However, STIDE is distinct from the LEAPBM technique in that it discards system-call arguments and utilises run-time analysis for model generation.

The investigations conducted for this study examine the behaviour of High Level Architecture distributed simulation programs; a domain distinct from those previously investigated. This domain provides programs that are of suitable size, complexity and ranging functionality. The specific HLA simulation which was examined is called the Air Transport Operations federation, and was developed for educational purposes by the Australian Defence Science & Technology Organisation.

8.1 Contributions

The LEAPBM technique achieves the states goals of this study, and in doing so makes several contributions. These goals are, improved accuracy while maintaining usable performance, improved human comprehensibility, and the application of PBM to novel types of programs.

8.1.1 Accuracy

This study's investigatory demonstrations indicate that the proposed flexible abstraction LEAPBM technique can more precisely distinguish between similar program behaviours than the STIDE technique. This is an important ability for PBM techniques, and facilitates an important performance

characteristic: the reduction of false-acceptance errors in their applications. For example, a more precise PBM technique employed in an anomaly-prevention system can better identify potential security attacks, and therefore has better performance. Further investigation of the ability of STIDE to model and compare HLA federate behaviour showed its poor performance at identifying anomalous HLA functionality. The excellent precision performance of the proposed LEAPBM technique suggests its suitability for correctly modelling more abstract and powerful programs than has been performed in the past.

A high level of precision can however, be falsely exhibited by a technique which poorly focuses on the functional equivalences of program behaviours. That is, if different program behaviours that are similar in appearance are correctly considered distinct, the danger is that *all* program behaviours will be considered distinct, even those that are functionally equivalent. The results of this study's investigation on focus performance, indicate that the LEAPBM technique correctly equates functionally identical program behaviours with differing definitions.

This study's results indicate that the precision for the STIDE technique's modelling of HLA federates is low. STIDE reports good focus performance, but exceedingly poor precision, which indicates STIDE is poorly suited to distinguishing HLA federate behaviours with similar appearances but distinct functionality.

The improved precision and excellent focus performance of the LEAPBM technique, compared to the existing STIDE technique, constitutes an improvement in accuracy.

Usable Performance

The supplemental goal of maintained complexity and performance for the LEAPBM technique is also successfully demonstrated. This demonstration covered both practical examples of the tested ACM federate, and theoretical asymptotic analysis of complexity growth. The practical and asymptotic complexities of the LEAPBM technique compared with those of the STIDE technique, achieve the supplemental goal of maintained complexity and performance.

8.1.2 Human Comprehension

The results from this study indicate that the LEAPBM technique's models for HLA programs contain fewer elements. The intuitive and preliminary discussion of the comparative human comprehensibility of STIDE and LEAPBM models suggests that LEAPBM offers improvements. However, further investigation is required to conclusively confirm or deny this preliminary indication. The complex issues involved in accurately measuring the subjective metric of human comprehensibility, and examining models for a range of HLA federate programs, are beyond the scope of this study.

This result is intuitively expected to provide the second benefit (improved human comprehensibility) from this study's research goal.

8.1.3 Modelling Novel Program Types

The second goal of this study was to investigate the modelling of new types of computer programs. Prior research focuses strongly on only two types of programs: privileged computer server processes, or Internet-borne mobile code. HLA simulation programs have aspects in common with prior research, such as the privileged execution on powerful interconnected computers, and the inclusion and understanding of multiple interoperating computers. HLA programs however, constitute a distinct type of program compared with those examined by prior research: they are more recent developments and employ more abstract and powerful software libraries than typical server processes. They are also more substantial than the relatively small pieces of mobile code that get passed around the Internet. This use of new types of programs in this study's investigations achieves the second goal of this study.

8.2 Limitations

This study has several limitations, which stem from the specific details of the investigations undertaken.

The unavailability or unsuitability of more recent, and potentially superior, PBM techniques (as reviewed in section 4.2.2) to compare with the proposed LEAPBM technique is unfortunate. Although the STIDE technique to which LEAPBM was compared might offer a common baseline for comparison with more recent techniques, it is impossible to know with any certainty how LEAPBM compares to more recent PBM techniques.

The lack of HLA certification for the ACM federate utilized in the investigations, limits the degree of confidence with which the outcomes of this study can be expected to apply to other HLA federates. However, this was unavoidable given the lack of access to certified HLA federates, and the high cost involved in gaining certification.

The definition of the HLA domain-specific logical abstract entities, is limited by the definer's knowledge and understanding of the domain. This raises the possibility of poor LEAPBM performance in certain environments, and necessitates expert knowledge for optimum performance. This could prove an obstable for future corroboration of the results presented in this study.

The approach to flexible abstraction detailed in this study, utilises a program's interactions with software library elements as terms to define its behaviour. It is unlikely that this is the only conceivable approach to flexible abstraction, and thus the demonstrated positive results cannot be said to apply to all forms of flexible abstraction.

Despite these limitations, this study does have some potential implications. The following section discusses these implications and outlines directions for future ongoing research.

8.3 Implications and Future Directions

The results of this study have a number of implications. Flexible abstraction for program behaviour modelling has implications for computing fields that employ program behaviour models. The demonstrated improvement in precision over an existing PBM technique implies the potential for improved accuracy in PBM application fields, such as computer security. The precision accuracy of PBM in computer security applications determines the amount of potentially dangerous program behaviour that is mistakenly executed.

The use of LEAPBM in general computer security applications is dependent on addressing any *circumvention* issues. Circumvention refers to the possibility that a LEAPBM application is bypassed, resulting in uncontrolled access to the resources it protects. In the expected typical case, that a LEAPBM model captures a program's behaviour through its use of abstract software libraries, the possibility of circumvention exists in a program directly utilising lower-level libraries[1].

A potential solution to this problem could employ existing techniques such as *sandboxing* (Gold-

[1] Such as the operating system kernel library.

berg et al., 1996), to control the execution environment of a computer program and intercept all external function calls. Another possible solution is a low-level implementation which monitors the dynamic loading of software libraries for application's execution, and restricts those excluded from the LEAPBM definition of a program's behaviour. Either of these solutions are likely to necessitate the additional requirement of LEAPBM models to represent *all* of a program's external function calls. These solutions could be a focus of future work on the application of the LEAPBM to the field of computer security.

The LEAPBM technique also has possible application in the area of HLA security. The *Matis* architecture, which implements the LEAPBM technique, was developed during the course of this study, and is freely available for download from *http://dsl.ballarat.edu.au*. Matis has been demonstrated to successfully identify anomalous federate behaviours that can indicate attempts to steal information. This result is described in the precision investigation in Chapter 5 and specifically concerns the ACM With-Additional-Subscriptions federate (ACMWAS). The ACMWAS federate was designed to subscribe to all federation object classes and gather as much information on their simulation as possible. Given the pervasive use of the HLA in military applications, and by military contracting companies, this technique could potentially be used to deduce information about sensitive systems being simulated. Matis's demonstrated ability to identify and control this technique has positive implications for these uses of HLA simulation.

The Matis architecture provides a basis upon which future developments could build to provide a more mature HLA security system. It is likely that such a system would be distributed in itself, and present on each computer involved in a HLA federation simulation. This distributed nature would require the strict adherence to secure programming techniques, for any future system based on the Matis architecture. The requirements of such a HLA security system have been previously explored (Andrews and Stratton, 2002).

An additional application of the Matis architecture is in the compliance testing of HLA federate programs. Current compliance testing procedures require dedicated testing by a Defence Modelling Simulation Office appointed official Certification Agent. This dedicated and specific testing is time consuming. The Matis architecture however is aware of all aspects of a federate program's HLA behaviour, thus it would appear possible that the Matis architecture could be used to provide on-going

real-time verification of HLA federate behaviour's compliance with HLA specifications. Recent work by the author on this topic has resulted in a paper to appear in the European Simulation Interoperability Workshop (Andrews et al., 2006).

An additional implication of this study concerns the human comprehensibility of program behaviour models. A preliminary comparison between the LEAPBM technique and the existing STIDE technique, intuitively indicates a possible improvement in the human comprehensibility of HLA federate program behaviour models. This has important implications for the application of the LEAPBM technique to both computer security and software development. If the intuitive results suggested by this study prove accurate, and a LEAPBM model for a program's behaviour can be reasonably comprehended by a human, it may be possible to incorporate an element of human analysis into both computer security and software testing.

Considering a system such as that proposed by Sekar et al. (2001b), wherein a program carries with it a model of its behaviour, a human could augment and potentially verify the automated comparison of the program's behaviour with the computer system security policy. While a human user cannot reasonably be expected to verify the security conformance of every program they execute on a computer, providing another option and an additional check for security intrusions can only offer improvements.

Considering the already mature application of PBM to software development, in its current specification a LEAPBM model is unlikely to be able to assist developers in planning internal software behaviour. However, providing a developer with additional insight into the behaviour of a program can only be of assistance in debugging, development and verification. Future research could also investigate the potential for LEAPBM to encapsulate a program's internal logical abstract entities, in addition to the external ones present in high-level software libraries. Such internal LAEs could potentially mirror some aspects of software development planning models, and be applicable to them.

Future research is needed to more conclusively determine the comparative human comprehensibility of LEAPBM models, and to investigate their use in enabling human analysis to augment existing computer security and software development processes.

The LEAPBM technique itself has examined a type of program that is neither a privileged processes running on a server system, nor a piece of mobile code being passed around the Internet. Such

programs have not previously received substantial attention in previous research. This investigation thus has implications for the use of PBM in more common computer programs. While this investigation has exclusively examined LEAPBM applied to HLA distributed simulation programs, future research could study its application to other types of programs. These investigations could examine the LEAPBM technique applied to new previously unexamined programs, as well as privileged server processes, mobile Internet-borne programs.

The LEAPBM technique itself could be the basis of interesting future developments and research. A comparison between the LEAPBM implementation and more recently developed system-call-based PBM techniques, such as those proposed by Giffin et al. (2002, 2004, 2005), and Sekar et al. (2001a,b, 2002), would provide further interesting insights into the relative accuracy and complexity of the LEAPBM technique.

Additional extensions and developments to the LEAPBM technique, and an application architecture such as Matis, could also be the basis for ongoing research. The LEAPBM generation algorithm could be implemented, potentially along with a graphical application to simplify the necessary manual definition of abstract basis-terms. This would then enable more direct and conclusive comparisons between the generation complexity and usability of LEAPBM and other techniques. The LEAPBM technique currently focuses specifically on the C++ programming language, but is capable of understanding the majority of concepts applicable to a wider range of languages. An interesting future development would be the investigation of extensions to LEAPBM to cover other programming languages.

Finally, the LEAPBM technique borrows concepts, such as the composition of both data and behaviour, from the object-oriented programming paradigm. This raises the possibility of further extensions to the LEAPBM technique to include the remaining object-oriented concepts, the most significant of which is inheritance. Such extensions could provide inheritance between LAE classes and further improve the LEAPBM technique's performance.

To summarise, the LEAPBM technique's improved accuracy has implications for many applications that employ program behaviour models. It is also hoped that this study's investigation of flexible abstract will encourage ongoing research in this new and exciting direction. All of the data, programs and configurations files utilised and presented in this study are available for download, from

the University of Ballarat Distributed Simulation Laboratory website at *http://dsl.ballarat.edu.au*.

Chapter 9

REFERENCES

S. Ambler. *The Object Primer: Agile Model Driven Development with UML 2.0*, chapter 11: Dynamic Object Design. Cambridge University Press, 2004.

D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Detecting Unusual Program Behavior using the Statistical Component of the Nextgeneration Intrusion Detection Expert System (NIDES). Technical report, SRI International, Computer Science Laboratory, Menlo Park, CA, USA, May 1995.

D. Anderson and G. McNeil. Artificial Neural Networks Technology. Technical report, US DoD Data and Analysis Center for Software, Rome, NY, USA, August 1992. World Wide Web: http://www.dacs.dtic.mil/techs/neural/neural

J. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Hanscom AFB, Air Force Systems Command, Electronic Systems Division, Bedford, MA, USA, October 1972.

J. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co, Box 42, Fort Washington, PA, USA, April 1980. Technical Report Contract 79F26400.

D. Andrews, P.Smith, D. Stratton, and J. Wharington. HLA Security through Real-Time Compliance Testing. In *Proceedings of the 2006 European Simulation Interoperability Workshop*, Stockholm, Sweden, June 2006. Simulation Interoperability Standards Organisation. Workshop paper: 06E-SIW-036.

D. Andrews and D. Stratton. Requirements and Feasibility of a Security Architecture for the Higher Level Architecture. Honours Thesis, 2002. School of Information Technology and Mathematical Sciences, University of Ballarat.

D. Andrews, D. Stratton, and J. Wharington. SecProxy - A Security Architecture Proposal for the HLA. In *Proceedings of the 2002 Fall Simulation Interoperability Workshop Proceedings*. Simulation Interoperability Standards Organisation, 2002a. Workshop paper: 02F-SIW-113.

D. Andrews, J. Wharington, and D. Stratton. SecProxy - A Proposed Security Architecture for the HLA. In *Proceedings of the 2002 Simulation Technology and Training Conference Proceedings*, pages 45–50. Simulation Industry Association of Australia, SimTect 2002 Organising and Technical Committee, May 2002b.

M. Auguston. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 1995 Automated and Algorithmic Debugging Conference*, pages 277–291, Saint Malo, France, May 1995.

S. Axelsson. Research in Intrusion Detection Systems: A Survey. Technical Report TR 98-17, Chalmers University of Technology, Department of Computer Engineering, S-412 96 Gothenburg, Sweden, December 1998. Revised August 19, 1999.

A. Bartussek and D. Parnas. Using Traces to Write Abstract Specifications for Software Modules. Technical report, University of North Carolina, Chapel Hill, NC, USA, 1977. UNC Rep. TR 77-012.

P. Bates and J. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3:255–264, 1983.

J. Beck and D. Eichmann. Program and Interface Slicing for Reverse Engineering. In *Proceedings of the 15th IEEE International Conference on Software Engineering*, Baltimore, MD, USA, May 1993. Institute of Electrical and Electronics Engineers, IEEE Society Press.

D. Bell and L. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Hanscom AFB, Air Force Systems Command, Electronic Systems Division, Bedford, MA, USA, 1975.

W. Bennett. *Visualizing Software: A Graphical Notation for Analysis, Design and Discussion*, chapter 1, page 1. Marcel Dekker, April 2004.

M. Bishop. A Model of Security Monitoring. In *Proceedings of the 5th Annual Computer Security Applications Conference*, Tucson, AZ, USA, December 1989.

P. Braspenning, F. Thuijsman, and A. Weijters. *Artificial Neural Networks*, chapter 1. Springer-Verlag, Berlin, Germany, 1995.

P. Bruza and T. Van der Weide. The Semantics of Data Flow Diagrams. In *Proceedings of the 1989 International Conference on Management of Data*, 1989.

L. L. Burkhart, A. Old, M. Loper, and M. B. Woldt. The Federate Test Sequence Explained. In *Proceedings of the 1998 Fall Simulation Interoperability Workshop Proceedings*. Simulation Interoperability Standards Organisation, 1998. Workshop paper: 98F-SIW-152.

A. Cansian, E. Moreira, A. Carvalho, and J. Jr. Network Intrusion Detection using Neural Networks. In *Proceedings of the 1997 International Conference on Computational Intelligence and Multimedia Applications*, pages 276–280, Gold Coast, QLD, Australia, February 1997.

J. Casad. *Sams Teach Yourself Tcp/Ip in 24 Hours*, chapter V: Hour 19: What Hackers Do, pages 324–325. Sams Publishing, 3rd edition, 2004.

H. Chang, S. Wu, and Y. Jou. Real-time Protocol Analysis for Detecting Link-state Routing Protocol Attacks. *ACM Transactions on Information and System Security*, 4(1):1–36, February 2001.

H. Chen and D. Wagner. MOPS : an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, 2002. ACM Press.

N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

M. Christerson, I. Jacobson, and L. Constantine. Object-Oriented Software Engineering - A Use Case Driven Approach. Technical report, Object Management Group, USA, 1992.

CISCO. NetRanger Intrusion Detection System. Technical Information, CISCO Systems, April 1999.

F. Cohen. 50 Ways to Defeat Your Intrusion Detection System. World Wide Web: http://all.net/journal/netsec/1997-12.html, 1998. Accessed at 14:49 April 8 2004.

W. Cohen. Fast Effective Rule Induction In Machine Learning. In *Proceedings of the Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, USA, June 1995. Morgan Kaufmann.

D. Curry, H. Debar, and B. Feinstein. The Intrusion Detection Message Exchange Format. Technical report, IETF Intrusion Detection Exchange Format Working Group, 2004.

T. W. Curry. Profiling and Tracing Dynamic Library Usage Via Interposition. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 268–278, Boston, MA, 1994.

H. Debar, M. Dacier, M. Nassehi, and A. Wespi. Fixed vs. Variable-Length Patterns for Detecting Suspicious Process Behavior. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, September 1998. Springer-Verlag.

D. E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13 (2):222–232, February 1987.

D. E. Denning and P. G. Neumann. Requirements and Model for IDES - A Real-Time Intrusion Detection Expert System. Technical Report 83F83-01-00, SRI International, Computer Science Laboratory, Menlo Park, CA, USA, 1985.

Department of Defence. Trusted Computer System Evaluation Criteria. Technical Report DoD 500.28-STD, United States Department of Defence, USA, 1985.

P. D'Haeseleer, S. Forrest, and P. Helman. An Immunological Approach to Change Detection: Algorithms, Analysis, and Implications. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 110–119, May 1996.

DMSO. *High Level Architecture Interface Specification*. United States Defence Modelling Simulation Office, Alexandria, VI, USA, April 1998.

DMSO. Federation Development and Execution Process Checklists. Technical report, United States Defence Modelling Simulation Office, Alexandria, VI, USA, December 1999.

DMSO. *High Level Architecture Run-Time Infrastructure - RTI 1.3-Next Generation Programmer's Guide*. United States Defence Modelling Simulation Office, USA, February 2002.

P. Domingos. Unifying Instance-Based and Rule-Based Induction. *Machine Learning*, 24(2), August 1996.

J. Dugelay, J. Junqua, C. Kotropoulos, R. Kuhn, F. Perronnin, and I. Pitas. Recent Advances in Biometric Person Authentication. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 4, pages IV4060–IV4063. IEEE Computer Society Press, 2002.

D. Endler. Intrusion Detection Applying Machine Learning to Solaris Audit Data. In *Proceedings of the 14th Annual Computer Security Applications Conference*, Scottsdale, AZ, USA, December 1998. IEEE Computer Society.

E. Eskin. Anomaly Detection over Noisy Data using Learned Probability Distributions. In *Proceedings of the 17th International Conference on Machine Learning*, Stanford, CA, USA, June 2000. Stanford University.

E. Eskin, W. Lee, and S. Stolfo. Modeling System Calls for Intrusion Detection with Dynamic Window Sizes. In *Proceedings of the 2001 DARPA Information Survivability Conference and Exposition*, volume 2, pages 165–175, Anaheim, California, June 2001. Institute of Electrical and Electronics Engineers, IEEE Computer Society.

E. Eskin, M. Miller, Z. Zhong, G. Yi, W. Lee, and S. Stolfo. Adaptive Model Generation for Intrusion Detection Systems. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, Athens, Greece, 2000. ACM Press.

H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, March 2004.

S. Forrest, S. Hofmeyr, and A. Somayaji. Computer Immunology. *Communications of the ACM*, 40 (10):88–96, 1997.

S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press, May 1996.

S. Forrest, A. Perelson, L. Allen, and R. Cherukuni. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press, 1994.

J. Foster, V. Osipov, N. Bhalla, and N. Heinen. *Buffer Overflow Attacks : Detect, Exploit, Prevent*, chapter 1. Syngress Publishing, 2005.

J. Frank. Artificial Intelligence and Intrusion Detection: Current and Future Directions. In *Proceedings of the 17th National Computer Security Conference*, pages 22–33, Washington, DC, USA, 1994. National Institute of Standards and Technology.

Free Software Foundation. *Linux Manual: ld*, July 2005. Linux Manual.

The GNU C Library. Free Software Foundation, Boston, MA, USA, April 2006. World Wide Web: http://www.gnu.org/software/libc/manual/.

A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0, March 1996. Internet Draft, Netscape Communications.

D. Frincke, D. Tobin, and Y. Ho. Planning, Petri Nets, and Intrusion Detection. In *Proceedings of the 21st National Information Systems Security*, pages 346–361, Arlington, VI, USA, October 1998. National Institute of Standards and Technology.

V. Ganapathy, S. Seshia, S. Jha, T. Reps, and R. Bryant. Automatic Discovery of API-Level Vulnerabilities. Technical Report UW-CS-TR-1512, University of Wisconsin-Madison Computer Sciences, July 2004.

S. Garfinkel, G. Spafford, and A. Schwartz. *Practical Unix & Internet Security*. O'Reilly & Associates, Sebastopol, CA, USA, 1996.

A. Ghosh, C. Michael, and M. Schatz. A Real-Time Intrusion Detection System Based on Learning Program Behavior. In *Proceedings of the 3rd International Workshop on the Recent Advances in Intrusion Detection*, Toulouse, France, October 2000.

A. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., USA, August 1999.

A. Ghosh, A. Schwartzbard, and M. Schatz. Learning Program Behavior Profiles for Intrusion Detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Santa Clara, CA, USA, April 1999.

A. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the 14th Annual Computer Security Applications Conference*, Scottsdale, AZ, USA, December 1998. IEEE Computer Society.

J. Giffin, D. Dagon, S. Jha, W. Lee, and B. Miller. Environment-Sensitive Intrusion Detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, USA, September 2005.

J. Giffin, S. Jha, and B. Miller. Detecting Manipulated Remote Call Streams. In *Proceedings of the 11th USENIX Security Symposium Proceedings*, San Francisco, CA, USA, August 2002. USENIX Association.

J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2004. The Internet Society.

B. Givens. Positions For and Against an Open-Source RTI. In *Proceedings of the 2000 Spring Simulation Interoperability Workshop*. Simulation Interoperability Standards Organisation, 2000. Workshop number: 00S-SIW-013.

I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium Proceedings*, San Jose, CA, USA, 1996.

M. Gonzalez, A. Serra, X. Martorell, J. Oliver, E. Ayguade, J. Labarta, and N. Navarro. Applying Interposition Techniques for Performance Analysis of OpenMP Parallel Applications. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, page 235, Cancun, Mexico, May 2000. Institute of Electrical and Electronics Engineers, IEEE Computer Society.

S. Grover. Buffer Overflow Attacks and Their Countermeasures. *Linux Journal*, March 2003.

L. R. Halme and R. K. Bauer. AINT Misbehaving: A Taxonomy of Anti-Intrusion Techniques. In *Proceedings of the 18th National Information Systems Security Conference*, pages 163–172, Baltimore, MD, USA, October 1995. National Institute of Standards and Technology/National Computer Security Centre.

M. Handley and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 10th USENIX Security Symposium Proceedings*, page 115131, Washington, D.C., USA, August 2001.

S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 2002 International Conference on Software Engineering Proceedings*, Orlando, FL, USA, May 2002.

C. Heitmeyer and J. McLean. Abstract Requirements Specification: A New Approach and Its Application. *IEEE Transactions on Software Engineering*, 9(5):580–589, September 1983.

B. Henderson Sellers and J. Edwards. The Object-Oriented Systems Life Cycle. *Communications of the ACM*, 33(9):142–159, September 1990.

B. Henry. Tech Target (SMB): Programming Language Generations. World Wide Web: http://whatis.techtarget.com/definition/0 sid9_gci211502 00.html, February 2001. Accessed at 14:47 April 8 2004.

P. Herrmann, L. Weibusch, and H. Krumm. State-based Security Policy Enforcement in Component-based E-Commerce Applications. In *Proceedings of the 2nd IFIP Conference on E-Commerce, E-Business and E-Government*, Lisbon, Portugal, October 2002. Kluwer Academic Publisher.

S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

G. Holzmann and M. Smith. Software Model Checking Extracting Verification Models. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Digital Rights Management*, page 141, Philadelphia, PA, USA, November 2001. Springer Berlin / Heidelberg.

IEEE. IEEE Standard for Distributed Interactive Simulation – Application Protocols. Technical report, Institute of Electrical and Electronics Engineers, New York, NY, USA, 1995. IEEE Std 1278.1-1995.

ISS. *Introduction to RealSecure 3.0*. Internet Security Systems, Atlanta, GA, USA, January 1999.

K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the 6th Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2000. Internet Society.

H. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Annual Report A010, SRI International, Computer Science Laboratory, Menlo Park, CA, USA, March 1994.

B. M. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th Association for Computing Machinery Symposium on Operating Systems Principle Proceedings*, pages 80–93, New York, NY, USA, 1993.

D. Jordan and C. Russell. *Java Data Objects*, chapter 1, page 1. O'Reilly Media, April 2003.

Y.F. Jou, F. Gong, C. Sargor, X. Wu, S.F. Wu, H.C. Chang, and F. Wang. Design and Implementation of a Scalable Intrusion Detection System for the Protection of Network Infrastructure. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume 2, page 1069, Hilton Head, SC, USA, January 2000.

W. Ju and Y. Vardi. Profiling UNIX Users And Processes Based on Rarity of Occurrence Statistics with Applications to Computer Intrusion Detection. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, Davis, CA, USA, October 2001.

P. Karger and R. Schell. Multics Security Evaluation: Vulnerability Analysis. Technical Report ESD-TR-74-193, Hanscom AFB, Air Force Systems Command, Electronic Systems Division, Bedford, MA, USA, June 1974.

A. Kent, J. Williams, and K. Kent. *Encyclopedia of Computer Science and Technology: Volume 25 - Supplement 10*, chapter 3.2, page 110. Marcel Dekker, October 1991.

C. Ko. *Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach*. PhD thesis, University of California: Davis, September 1996.

C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications*, pages 134–144, December 1994.

L. Kohout, A. Yasinsac, and E. McDuffie. Activity Profiles for Intrusion Detection. In *Proceedings of the 2002 North American Fuzzy Information Processing Society-Fuzzy Logic and the Internet*, pages 463–468, New Orleans, LA, USA, June 2002.

A. Kosoresow and S. Hofmeyr. Intrusion Detection via System Call Traces. *IEEE Software*, 14(5): 35–42, September 1997.

C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In D. Gollmann E. Snekkenes, editor, *Proceedings of the 8th European Symposium on Research in Computer Security*, pages 326–343, Gjvik, Norway, October 2003. Springer-Verlag.

J. Kuhn. Research toward Intrusion Detection through the Automated Abstraction of Audit Data. In *Proceedings of the 9th National Computer Security Conference*, pages 204–208. National Institute of Standards and Technology, September 1986.

B. A. Kuperman and E. Spafford. Generation of Application Level Audit Data via Library Interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, West Lafayette, Indiana 47907-1398, October 1999.

W. Lucyshyn L. A. Gordon, M. P. Loeb and R. Richardson. 2004 CSI/FBI Computer Crime and Security Survey. World Wide Web: http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2004.pdf, 2004. Accessed at 12:23 December 14 2006.

W. Lucyshyn L. A. Gordon, M. P. Loeb and R. Richardson. 2005 CSI/FBI Computer Crime and Security Survey. World Wide Web: http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2005.pdf, 2005. Accessed at 12:24 December 14 2006.

W. Lucyshyn L. A. Gordon, M. P. Loeb and R. Richardson. 2006 CSI/FBI Computer Crime and Security Survey. World Wide Web: http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2006.pdf, 2006. Accessed at 12:25 December 14 2006.

T. Lane. Hidden Markov Models for Human/Computer Interface. In *Proceedings of the 1999 IJCAI Workshop on Learning About Users*, Stockholm, Sweden, July 1999.

T. Lane and C. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 150–158, San Francisco, CA, United States, 1998. ACM Press.

T. Lane and C. E. Brodley. Sequence Matching and Learning in Anomaly Detection for Computer Security. In *Proceedings of the 1997 AAAI Workshop on AI Methods in Fraud and Risk Management*, pages 43–49. American Association for Artificial Intelligence, AAAI Press, July 1997.

J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation*, La Jolla, CA, USA, June 1995.

W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, uSA, January 1998.

W. Lee, S. Stolfo, and P. Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of the AAAI97 Workshop on AI Methods in Fraud and Risk Management*, pages 50–56. American Association for Artificial Intelligence, AAAI Press, July 1997.

W. Lee, S. Stolfo, and K. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 120–132, Berkeley, California, United States of America, May 1999. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

T. Li. Behavioral Clustering and Statistical Intrusion Detection. Master's thesis, Florida State University, College of Arts and Sciences, Department of Computer Science, 1997.

J. Lin, X. Wang, and S. Jajodia. Abstraction-Based Misuse Detection: High-Level Specifications and Adaptable Strategies. In *Proceedings of the 11th Computer Security Foundations Workshop Proceedings*, pages 190–201, Rockport, MA, USA, June 1998.

Lin J. *Abstraction-Based Misuse Detection: High-Level Specifications and Adaptable Strategies*. PhD thesis, George Mason University, Fairfax, VA, USA, December 1998.

Linux. *Linux Manual: syscalls*, January 2002. Linux Manual.

P. Liu. Architectures for Intrusion Tolerant Database Systems. In *Proceedings of the 2002 Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2002.

M. Loper, T. McLean, M. Horst, and K. Crawford. The High Level Architecture Federate Conformance Testing Process. In *Proceedings of the 1997 Fall Simulation Interoperability Workshop Proceedings*. Simulation Interoperability Standards Organisation, 1997. Workshop paper: 97F-SIW-062.

E. Lundin and E. Jonsson. Some Practical and Fundamental Problems with Anomaly Detection. In *Proceedings of the 4th Nordic Workshop on Secure IT Systems*, 1999.

T. Lunt. Automated Audit Trail Analysis and Intrusion Detection: A Survey. In *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, USA, 1988. National Institute of Standards and Technology.

T. Lunt. Detecing Intruders in Computer Systems. In *Proceedings of the 1993 Conference on Auditing and Computer Technology*, 1993.

R. Martin. UML Tutorial: Collaboration Diagrams. Engineering Notebook, November 1997. World Wide Web: http://www.objectmentor.com/publications/umlCollaborationDiagrams.pdf.

R. Martin. UML Tutorial: Complex Transitions. Engineering Notebook C++ Report., September 1998a. World Wide Web: http://www.objectmentor.com/publications/cplxtrns.pdf.

R. Martin. UML Tutorial: Finite State Machines. Engineering Notebook C++ Report., June 1998b. World Wide Web: http://www.objectmentor.com/publications/UMLFSM.PDF.

R. Martin. UML Tutorial: Sequence Diagrams. Engineering Notebook., April 1998c. World Wide Web: http://www.objectmentor.com/publications/UMLSequenceDiagrams.pdf.

R. Martin. UML Use Case Diagrams. Engineering Notebook C++ Report, November 1998d. World Wide Web: http://www.objectmentor.com/publications/usecases.pdf.

J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*, chapter 1. Sun Microsystems Press, 2001.

R. Maxion and K. Tan. Anomaly Detection in Embedded Systems. *IEEE Transactions on Computers*, 51(2):108–120, February 2002.

J. McLean. A Formal Method for the Abstract Specification of Software. *Journal of the ACM (JACM)*, 31(3):600–627, 1984.

D. Miller and J. Thorpe. SIMNET: The Advent of Simulator Networking. *Proceedings of the IEEE*, 83(8):1114–1123, August 1995.

Ltd. Mind Tools. Decision Tree Analysis - Choosing Between Options by Projecting Likely Outcomes. World Wide Web: http://www.mindtools.com/pages/article/newTED_04.htm, February 2006. Accessed at 18:46 February 22 2006.

S. Murthy. A Tool for Static Model Extraction from Linux Programs. Technical report, Computer Science Department, Stony Brook University, Stony Brook, NY 11794, January 2003.

M. J. B. M. Nasir. A Journey Through Programming Language Generations. *Surveys and Presentations in Information Systems Engineering (SURPRISE)*, 2, 1996.

NCSC. A Guide to Understanding Audit in Trusted Systems. Technical Report NCSC-TG-001, National Computer Security Centre, Fort George G. Meade, MD, USA, June 1988.

G. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

P. Neumann and P. Porras. Experience with EMERALD to date. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring Proceedings*, pages 73–80, Santa Clara, CA, USA, April 1999.

P. Ning, S. Jajodia, and X. Wang. Abstraction-based Intrusion Detection in Distributed Environments. *ACM Transactions on Information and System Security (TISSEC)*, 4(4):407–452, 2001.

Object Management Group. Unified Modelling Language: Superstructure. Technical report, USA, August 2005.

T. Olovsson. A Structured Approach to Computer Security. Technical report, Chalmers University of Technology, Department of Computer Engineering, S-412 96 Gothenburg, Sweden, 1992. Technical Report No 122.

V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, USA, January 1998.

J. Picciotto. The Design of an Effective Auditing Subsystem. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press, April 1987.

J. Nievola R. Santos and A. Freitas. Extracting Comprehensible Rules from Neural Networks via Genetic Algorithms. In *Proceedings of the 2000 IEEE Symposium on Combinations of Evolutionary*

Computation and Neural Networks, pages 130–139, San Antonio, TX, USA, May 2000. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

L. Raccoon. Fifty Years of Progress in Software Engineering. *ACM SIGSOFT Software Engineering Notes*, 22(1), January 1997.

J. Refsnes. Introduction to DTD. World Wide Web: http://www.xmlfiles.com/dtd/dtd_intro.asp, February 2006. Accessed at 19:29 February 22 2006.

R. Richardson. 2003 CSI/FBI Computer Crime and Security Survey. World Wide Web: http://www.gocsi.com/forms/fbi/pdf.html, 2003. Accessed at 17:51 August 14 2003.

D. Ritchie. *The UNIX Programmer's Manual: On the Security of UNIX*, June 1977.

S. Ross. *UNIX System Security Tools*. McGraw-Hill Companies, 1999.

R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys (CSUR)*, 28(1):241–243, March 1996. ISSN: 0360-0300.

SANS-Institute. The Twenty Most Critical Internet Security Vulnerabilities (Updated) The Experts Consensus. World Wide Web: http://www.sans.org/top20/, November 2005. Accessed at 20:17 February 20th 2006.

R.R. Schell, P. J. Downey, and G. J. Popek. Preliminary Notes on the Design of Secure Military Computer Systems. Technical Report MCI-73-1, Hanscom AFB, Air Force Systems Command, Electronic Systems Division, Bedford, MA, USA, January 1973.

B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model Checking An Entire Linux Distribution for Security Violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, AZ, USA, December 2005. IEEE Computer Society.

K. Seefeld. *Java Programming Fundamentals*, chapter 8, page 115. Charles River Media, August 2002.

R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and*

Privacy, pages 144–155, Oakland, CA, USA, May 2001a. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

R. Sekar, T. Bowen, and M. Segal. On Preventing Intrusions by Process Behavior Monitoring. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, April 1999.

R. Sekar, Y. Cai, and M. Segal. A Specification-Based Approach for Building Survivable Systems. In *Proceedings of the 21st National Information Systems Security Conference Proceedings*, Arlington, VI, USA, October 1998.

R. Sekar, A. Gupta, J. Frullo, and T. Shanbha. Specification-Based Anomaly Detection: A New Approach for Detecting Network Intrusions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security Proceedings*, pages 265–274, Washington, DC, USA, 2002. ACM Press. ISBN: 1-58113-612-9.

R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model-Carrying Code (MCC): A New Paradigm for Mobile Code Security. In *Proceedings of the 2001 New Security Paradigms Workshop Proceedings*, Cloudcroft, NM, USA, September 2001b.

R. Sekar and P. Uppuluri. Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications. In *Proceedings of the 8th USENIX Security Symposium Proceedings*, pages 63–78, Washington, DC, USA, August 1999.

R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Boston Landing, NY, USA, October 2003.

SGI. IRIX 6.5 Technical Brief. Technical report, Silicon Graphics, Mountain View, CA, USA, 1998.

J. Siegel. Introduction to OMG's Unified Modeling Language (UML). World Wide Web: http://www.omg.org/gettingstarted/what_is_uml.htm, July 2005. Accessed at 20:15 January 5 2006.

E. Siever, S. Figgins, A. Weber, R. Love, and A. Robbins. *Linux in a Nutshell*, chapter 1. O'Rielly Media, Sebastopol, CA, USA, 5th edition, 2005.

R. Sladkey. *Linux Manual: strace*, 1995. Linux Manual.

M. Sobell. *Unix System 5: A Practical Guide*, chapter 1. Addison Wesley, September 1994.

A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium*, page 185198, Denver, C.L., USA, August 2000.

A. B. Somayaji and S. Forrest. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 1 University of New Mexico, Albuquerque, New Mexico, United States of America, July 2002.

J. Sterlicchi. Developers Building in Protection. Newspaper: The Australian 25/05/2005, May 2004. IT Business Special Report: Security, page 2.

D. Stratton, S. Parr, and J. Miller. Developing an Open-Source RTI Community. In *Proceedings of the 2004 Spring Simulation Interoperability Workshop*. Simulation Interoperability Standards Organisation, 2004.

B. Stroustrup. *The C++ Programming Language*, chapter 1. Addison-Wesley, Reading, MA, USA, 1986.

Sun Microsystems. *Solaris 7 Software Developer Collection: System Interface Guide*, chapter 2 - Java Programming. Sun Microsystems Press, 1998.

Sun Microsystems. *Trusted Solaris 7 Reference Manual: Man Pages (2): System Calls*, chapter System Calls. Sun Microsystems Press, 1999.

Sun Microsystems. SunSHIELD Basic Security Module Guide. Palo Alto, CA, USA, 2000.

K. Tan, K. Killourhy, and R. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, pages 54–73, Zurich, Switzerland, October 2002a.

K. Tan, J. McHugh, and K. Killourhy. Hiding Intrusions: From the Abnormal to the Normal and Beyond. In F. Petitcolas, editor, *Proceedings of the 5th International Workshop on Information Hiding*, pages 1–17. Springer-Verlag, October 2002b.

H. S. Teng, K. Chen, and S. Lu. Adaptive Real-time Anomaly Detection Using Inductively Generated Sequential Patterns. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 278–284, Oakland, CA, USA, May 1990. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

V.N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An Approach for Secure Software Installation. In *Proceedings of the 16th USENIX Large Installation System Administration Conference*, Philadelphia, PA, USA, November 2002.

D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, University of California: Berkley, Berkley, CA, USA, 2000.

D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2001. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

D. Wagner and P. Soto. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security Proceedings*, pages 255–264, Washington, DC, USA, 2002. ACM Press.

C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Berkeley, California, United States of America, May 1999. Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press.

P. Watters. *Solaris 8 Administrator's Guide*, chapter 1, page 1. O'Reilly, 2002.

Webopedia. Kernel Definition. World Wide Web: http://www.webopedia.com/TERM/k/kernel.html, 2003. Accessed at 12:00 July 4 2003.

M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, May 1981.

M. Weiss. *Data Structures and Algorithm Analysis in Java*, chapter 2. Addison-Wesley, 1999.

A. Wespi, M. Dacier, and H. Debar. An Intrusion-Detection System Based on the Teiresias Pattern Discovery Algorithm. In K. Pedersen U. E. Gattiker, P. Pedersen, editor, *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Annual Conference*, Aalborg, Denmark, February 1999. European Institute for Computer Anti-Virus Research.

B. R. Wetmore. Paradigms for the Reduction of Audit Trails. Master's thesis, University of California: Davis, 1993.

J. Wharington and D. Andrews. A Guide to the DSILI Software Core. Technical report, DSTO Platforms Sciences Laboratory, Defence Science Technology Organisation, Maribyrnong, Victoria, Australia, 2002.

J. Wharington, A. Travers, D. Stratton, and P. Smith. Introduction to High Level Architecture Distributed Simulation. Course notes, January 2002.

J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Robziar, and J. Stern. Design for Multics Security Enhancements. Technical Report ESD-TR-74-176, Hanscom AFB, Air Force Systems Command, Electronic Systems Division, Bedford, MA, USA, 1974.

Wikipedia. Abstraction. World Wide Web: http://en.wikipedia.org/wiki/Abstraction, February 2006a. Accessed at 15:42 February 7 2006.

Wikipedia. Compiler. World Wide Web: http://en.wikipedia.org/wiki/Compiler, February 2006b. Accessed at 20:01 Februrary 21 2006.

Wikipedia. Endianness. World Wide Web: http://en.wikipedia.org/wiki/Endianness, February 2006c. Accessed at 22:20 February 22 2006.

A. Wilson and R. Weatherly. The Aggregate Level Simulation Protocol: an Evolving System. In *Proceedings of the 26th Winter Simulation Conference*, pages 781–787, Orlando, FL, USA, 1994. Society for Computer Simulation International.

M. B. Woldt and L. L. Burkhart. HLA Federate Compliance Testing: Keys to a Successful Test. In *Proceedings of the 1999 Spring Simulation Interoperability Workshop Conference Proceedings*. Simulation Interoperability Standards Organisation, 1999. Workshop paper: 99S-SIW-200.

L. Xun. A Linux Executable Editing Library. Master's thesis, National University of Singapore, Singapore, 1999.

Appendix A

UTILISED SOFTWARE

A.1 Air Transport Operations Federation

The Air Transport Operations (ATO) federation is designed to simply simulate the behaviour of aircraft. The ATO was written by Dr. John Wharington, Dr. Anthony Travers, and Dr. Lorenz Drack for use in the *HLA Short Course*; developed by Maritime Platforms Division of the Defence Science & Technology Organisation of Australia; and taught by the University of Ballarat in Victoria, Australia.

The ATO federation simulates the international flights of aircraft by an international airline service. This federation is composed primarily of three federates:

1. Aircraft Manager
2. Air Traffic Controller
3. Fleet Manager

These federates simulate different portions of aircraft flight including, the physical aircraft object characteristics, the airports at which aircraft land, and the planning of aircraft destinations.

The following sections detail what objects are simulated by the ATO federation, and the simulation responsibilities of each federate.

Object Model

The ATO federation's object model is a simple hierarchy, consisting of five object classes: Position, Aircraft, Airport, Storm and Runway. This hierarchy is shown in the Figure A.1.

The attributes of the Position object class are designed to capture the location of an object instance in a flat Earth space. They defined in the Position class and then inherited by the Aircraft, Airport and Storm child classes.

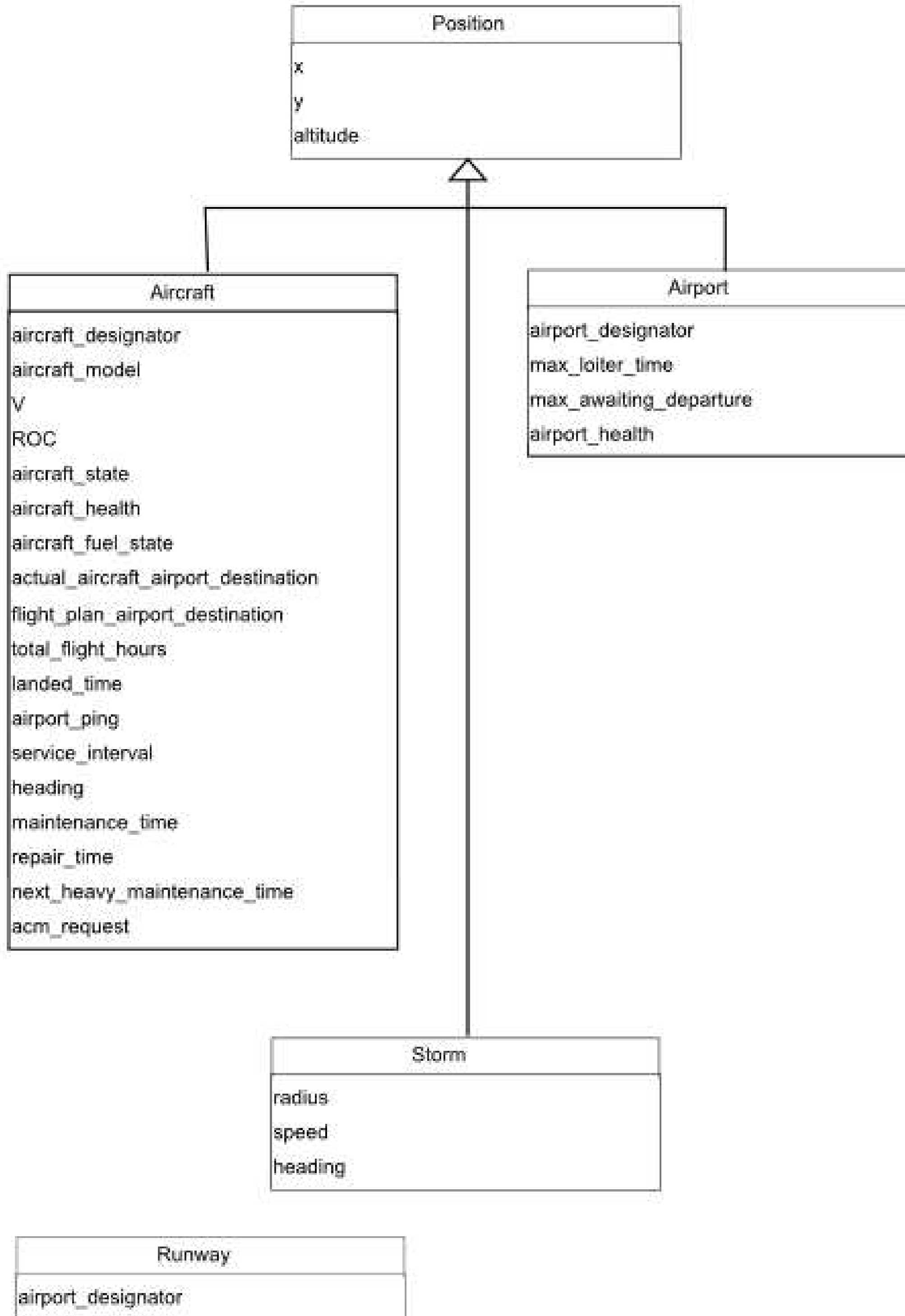

Figure A.1: Air Transport Operations Federation Object Model Hierarchy

The Aircraft, Airport and Storm object classes represent simulation objects which exist in the flat Earth space. Aircraft fly between Airports with fixed locations, while Storms exist and drift between random locations.

Aircraft Manager Federate

The Aircraft Manager (ACM) federate is responsible for simulating the 'Aircraft' federation objects. The ACM federate represents the Aircraft movements and state: position, speed, direction and mechanical state.

Air Traffic Controller Federate

The Air Traffic Controller (ATC) federate is responsible for simulating the 'Airport' federation objects. ATC represents the Airport's state and their interactions with 'Aircraft' objects during take-off and landing.

Fleet Manager Federate

The Fleet Manager (FM) federate is responsible for several aspects of 'Aircraft' federation objects: maintenance, refueling, and flight path planning.

A.2 STIDE

The Sequence TIme-Delay Embedding (STIDE) PBM technique was developed by the Computer Immune Systems research group at the University of New Mexico, who are also responsible for the initial use of system-call basis-terms for PBM (Forrest et al., 1996; Hofmeyr et al., 1998; Warrender et al., 1999). The STIDE software is available for download from *http://www.cs.unm.edu/ immsec/-software/*. In addition to this software, some additional work was required to get STIDE to perform its functionality in real-time. This real-time processing was necessary to ensure compatible comparisons with Matis, which also executes in real-time.

A.2.1 Real-Time Processing Modifications

The real-time processing modifications made to STIDE, primarily concern the ways in which its input data is gathered. The following steps were performed:

1. Version 4.5.9 of the *strace* utility was downloaded.

2. *strace* was modified to produce an output compatible with STIDE's input data format; multiple lines each with a process identifier, and the internal operating system number identifier for the system-call used.

3. An operating system FIFO file object (First-In-First-Out) is created.

   ```
   dave@vortex $ mkfifo acmrun.dat
   ```

4. Real-time output from the strace utility monitoring the executing federate is redirected to the FIFO file.

   ```
   dave@vortex $ sstrace -o acmrun.dat acmfederate/acm
   ```

5. STIDE is started and reads its input lines from the FIFO file as strace captures them directly from the executing federate.

   ```
   dave@vortex $ stide -d acm-seq10.db < acmrun.dat
   ```

A.3 CPU Cycle Measurement

The CPU cycle measurement is performed using the Read Time Stamp Counter (RDTSC) function of Intel Pentium processors, which returns the current CPU cycles since power up. This function is accessible directly from C++:

```
unsigned long long int x;
__asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
```

Measuring the RDTSC value at two points and subtracting the difference provides a highly accurate indication of a program or function's processor requirements.

Appendix B

LAPTOP COMPUTER

This appendix details the hardware used during the investigation performed by this study. The laptop computer is an Acer TravelMate 4100 with:

- Intel Corporation Pentium M Processor. 1.73GHz and 2048KB Cache.
- 512MB Random Access Memory.
- 80GB Hard Disk Drive.

B.1 Operating System Configuration

The Operating System on the computer used during this study was the Gentoo Linux distribution, with kernel version 2.6.12-r4 #10. The following packages are installed on this computer: sys-apps/slocate, net-wireless/ipw2200-firmware, media-libs/win32codecs, media-sound/alsa-utils, sys-process/vixie-cron, sys-libs/glibc, dev-db/postgresql, dev-db/phppgadmin, media-video/gxine, app-text/xpdf, www-client/lynx, sys-kernel/linux-headers, media-gfx/sodipodi, dev-libs/boost, app-arch/rpm, dev-java/sun-jdk, dev-libs/xerces-c, dev-tcltk/expect, app-emacs/nxml-mode, sci-visualization/gnuplot, app-emacs/php-mode, media-gfx/xfig, app-editors/vim, media-fonts/corefonts, net-misc/telnet-bsd, net-irc/xchat-systray, net-wireless/wepdecrypt, app-arch/sharutils, net-wireless/wpa_supplicant, net-wireless/ipw2200, media-libs/xine-lib, sys-boot/grub, www-misc/gurlchecker, net-wireless/aircrack, net-dialup/rp-pppoe, net-www/apache, media-sound/xmms, media-gfx/gimp, x11-base/xorg-x11, net-misc/dhcpcd, gnome-base/gnome, net-ftp/ftp, net-wireless/wireless-tools, app-office/openoffice-bin, dev-tex/latex2html, dev-util/strace, app-arch/unrar, app-admin/syslog-ng, app-editors/nano, x11-misc/xscreensaver, media-gfx/graphviz, dev-util/cvsd, sys-kernel/gentoo-sources, media-fonts/sharefonts, app-text/tetex, media-video/xine-ui, net-im/gaim, net-wireless/wepattack, x11-drivers/ati-drivers, dev-php/mod_php, net-

irc/xchat, net-p2p/azureus-bin, net-firewall/iptables, app-text/dos2unix, app-text/rpl, dev-libs/libxml2, dev-php/php, sys-libs/libstdc++-v3. www-client/epiphany, x11-drivers/synaptics and dev-util/argouml.

Appendix C

ADDITIONAL DEMONSTRATIONS

C.1 Simple GTK Program System Call Usage

The most basic program that can be built Gimp Tool Kit (GTK)[1] graphical user interface software library, as per the tutorial available at *http://www.gtk.org/tutorial/c58.html* was implemented on the computer system described in Appendices B and B.1.

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]){

    GtkWidget *window;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_show  (window);

    gtk_main ();

    return 0;

}
```

This program source was compiled using the command line specified in the same tutorial:

```
dave@vortex $ gcc base.c -o base `pkg-config --cflags --libs gtk+-2.0`
```

This complied program was then executed using strace as follows:

```
dave@vortex $ strace -c ./base
```

[1] The Gimp Tool Kit (GTK) is a software library designed for creating graphical user interfaces, and is available from *http://www.gtk.org*.

This execution resulted in the following system-call summary:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 43.83    0.004230         111        38           write
 21.47    0.002072          11       184         2 read
  6.37    0.000615          32        19           ioctl
  4.30    0.000415         138         3           socket
  4.22    0.000407           4        91           mmap2
  3.74    0.000361           7        50           open
  3.65    0.000352          39         9           writev
  2.59    0.000250         125         2           select
  2.49    0.000240          11        21         1 poll
  1.24    0.000120           2        52           close
  1.12    0.000108           2        47           fstat64
  0.93    0.000090           5        18        13 access
  0.91    0.000088          88         1           execve
  0.87    0.000084          28         3         2 connect
  0.63    0.000061           5        12           munmap
  0.41    0.000040          20         2           readv
  0.27    0.000026           7         4         2 lstat64
  0.22    0.000021           3         8           uname
  0.21    0.000020           7         3           mprotect
  0.20    0.000019           2         9           fcntl64
  0.11    0.000011           3         4           brk
  0.05    0.000005           3         2           _llseek
  0.05    0.000005           5         1           stat64
  0.04    0.000004           1         3           getpid
  0.02    0.000002           2         1           getrlimit
  0.02    0.000002           2         1           getuid32
  0.02    0.000002           2         1           getresuid32
  0.02    0.000002           2         1           getresgid32
------ ----------- ----------- --------- --------- ----------------
100.00    0.009652                   590        20 total
```

As can be easy seen, the 4 GTK library calls made in the original source code (*gtk_init*, *gtk_window_new*, *gtk_widget_show*, *gtk_main*) resulted in 590 system library calls.

C.2 Demonstration: Simple HLA Federate System Call Usage

A very simple HLA federate application is supplied with the US Department of Defence Modelling and Simulation Office HLA Run-Time Infrastructure called ‘helloWorld’.

This program was compiled on the computer system described in Appendices B and B.1 as per the Makefile provided.

This compiled program was executed a number of times using the Matis LEAPBM application modified to include a simple counter of calls made by the program:

```
dave@vortex $ matisrti --run ./helloWorld Australia 1
```

These executions resulted in an average of 332 HLA library function calls.

The helloWorld federate was then executed a number of times using strace as follows:

```
dave@vortex $ strace -c ./helloWorld Australia 1
```

These executions resulted in an average 24918 of system-library function calls.