Formal Models of Reproduction: from Computer Viruses to Artificial Life

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Matthew Paul Webster

October 2008

Primary Supervisor:	Dr. Grant Malcolm Department of Computer Science University of Liverpool
Secondary Supervisor:	Dr. Alexei Lisitsa Department of Computer Science University of Liverpool
Internal Examiner:	Prof. Michael Fisher Department of Computer Science University of Liverpool
External Examiner:	Prof. Lt. Col. Éric Filiol Virology and Cryptology Laboratory Ecole Supérieure et d'Application des Transmissions

Contents

Abstract **vii**

Acknowledgements **ix**

Preface **xi**

1 Introduction **1**

1.1 Reproducing Programs... 1

1.1.1 Motivations of Computer Virus Writers 2

1.1.2 A Short History of Computer Viruses 3

1.1.3 Academic Study of Computer Viruses 3

1.1.4 Detection of Malware 5

1.1.4.1 Static Analysis 6

1.1.4.2 Dynamic Analysis 7

1.2 ... and Other Reproducing Things 8

1.3 Formal and Informal Approaches to Problem Solving 10

1.4 Overview of the Thesis 11

1.4.1 Computer Viruses and Artificial Life 13

1.4.2 A Note on the Inclusion of Computer Virus Code 14

2 Formal Detection of Metamorphic Computer Viruses **15**

2.1 Introduction 15

2.1.1 Algebraic Specification 16

2.1.2 Chapter Overview 16

2.2 Metamorphic Computer Viruses 17

2.2.1 Types of Code Metamorphosis 18

2.2.1.1 Junk Code Insertion 18

2.2.1.2 Variable Renaming 19

2.2.1.3 Unconditional Jump Insertion 19

2.2.1.4 Instruction Reordering 19

2.2.1.5 Pseudo-Conditional Jump Insertion 19
2.2.1.6 Arithmetical/Boolean Mutation 20
2.2.1.7 Payload Mutation 20
2.2.1.8 Pseudo Branching 20
2.3 Algebraic Specification in Maude 20
2.4 Specifying Intel 64 Assembly Language 22
2.4.1 Specifying the Syntax of Intel 64 23
2.4.2 Specifying the Semantics of Intel 64 24
2.4.2.1 Intel 64 Stack Semantics 26
2.4.3 Using the Maude Specification as an Interpreter 27
2.5 Equivalence of Instruction Sequences 28
2.6 Dynamic Analysis 31
2.6.1 Example 1: Win95/Bistro 31
2.6.2 Example 2: Win9x.Zmorph.A 35
2.7 Static Analysis 36
2.7.1 Equivalence in Context 36
2.7.2 Examples Using Win9x.Zmorph.A 38
2.8 Applications to Detection of Metamorphic Viruses 41
2.8.1 Dynamic Analysis 41
2.8.1.1 Signature Equivalence 41
2.8.1.2 Signature Semi-Equivalence 42
2.8.2 Static Analysis 42
2.8.2.1 Formally-Verified Equivalent Code Libraries 42
2.8.2.2 Equivalence in Context 43
2.8.3 Combination With Other Approaches 44
2.9 Summary 44
2.9.1 Related Work 45
2.9.1.1 Control- and Data-Flow Analysis 46
2.9.1.2 Semantics Template Matching 48
2.9.1.3 Program Rewriting and Normalisation 49
2.9.1.4 Metamorphic Engine Analysis 51
2.9.1.5 Neural Network Approaches 52
2.9.1.6 Industrial Approaches 52
2.9.2 Comparisons with Related Work 53
2.9.2.1 Static and Dynamic Analysis 53
2.9.2.2 Formal and Informal Approaches 54
2.9.2.3 Generality and Readiness for Application 56

2.9.2.4 Applications Beyond Computer Virology 57

3 Formal Affordance-based Models of Reproduction 59

3.1 Introduction 60
3.1.1 The Theory of Affordances 60
3.1.2 Structure of the Chapter 61
3.2 One Possible Classification Scheme 62
3.2.1 Type I Reproducers 64
3.2.1.1 Example: von Neumann Reproducing Automaton . . . 64
3.2.1.2 Example: Langton's Loop 64
3.2.2 Type II Reproducers 65
3.2.2.1 Example: Tape from von Neumann's Reproducing Automaton 65
3.2.2.2 Example: T4 Bacteriophage 66
3.2.2.3 Example: Source Code Computer Virus 66
3.2.3 Type III Reproducers 66
3.2.3.1 Example: Compiler 67
3.2.3.2 Example: Damaged Cell 67
3.2.4 Type IV Reproducers 67
3.2.4.1 Example: Game of Life Gliders 67
3.2.4.2 Example: The Photocopy 68
3.2.5 Questions about Affordance-based Classification 68
3.2.5.1 The Assisted Reproduction Conjecture 68
3.2.5.2 The Unassisted Reproduction Conjecture 69
3.2.5.3 Varying Degrees of Assistance 70
3.2.5.4 Other Means of Classifying Reproducers Using Affordances 71
3.3 Towards Formal Reproduction Models 71
3.4 Formal Models of Reproduction 72
3.4.1 Classifying Reproduction Models 75
3.4.2 Refinement of Reproduction Models 78
3.4.3 Allowed Refinements of Reproduction Models 82
3.5 The Unassisted and Assisted Reproduction Theorems 82
3.5.1 The Unassisted Reproduction Theorem 83
3.5.2 The Assisted Reproduction Theorem 86
3.5.3 Further Classification Using Aspects 88
3.6 Further Examples 90
3.6.1 Langton's Loop 90
3.6.2 Conway's Game of Life Gliders 93
3.7 Summary 94

3.7.1 Related Work 95
3.7.1.1 Löfgren's Approach to Modelling Reproduction 96
3.7.1.2 A Universal Framework for Self-Replication 98
3.7.1.3 Autopoiesis 99
3.7.1.4 Reproduction in Cellular Automata 100
3.7.1.5 Reproduction Classification by Dawkins 102
3.7.1.6 Reproduction Classification by Taylor 102
3.7.1.7 Reproduction Classification by Luksha 104
3.7.2 Comparisons with Other Approaches 105
3.7.2.1 Comparison with Löfgren's Approach 105
3.7.2.2 Comparison with A Universal Framework 106
3.7.2.3 Comparison with Cellular Automata 107
3.7.2.4 Arbitrariness of Assistance 108
3.7.2.5 Sexual Reproduction 110
3.7.2.6 Triviality and Non-triviality 110
3.7.2.7 Comparison with Multiagent Systems 112
3.7.2.8 Comparison with Formal Methods for Concurrent Systems113
3.7.3 Comparison with Rosen's Ideas on Life 114
3.7.3.1 Life Itself 114
3.7.3.2 Rosen's Paradox 115
3.7.4 Reproduction as Preservation of Information Over Time 116
3.7.5 Further Application to Artificial Life 118

4 Formal Affordance-based Models of Computer Viruses 121
4.1 Introduction 121
4.1.1 Chapter Overview 122
4.2 Computer Virus Reproduction Models 123
4.2.1 Formal Models of Computer Virus Reproduction 123
4.2.2 Classifying Computer Viruses 126
4.2.3 Modelling a Unix Shell Script Virus 127
4.2.4 Modelling Virus.VBS.Archangel 129
4.2.5 Modelling Virus.Java.Strangebrew 132
4.2.6 Modelling an Assembly Language Computer Virus 136
4.3 Automatic Classification 138
4.3.1 Behaviour Monitoring and Classification 140
4.3.2 Static Analysis of Virus.VBS.Baby 141
4.3.3 Static Analysis of Virus.VBS.Archangel 143
4.3.4 Dynamic Analysis of Virus.VBS.Baby 144

4.3.5 Metrics for Comparing Assisted Viruses . . . 149
4.3.5.1 A Simple Metric for Comparing Assisted Viruses 149
4.3.6 Comparing Behaviour Monitor Configurations . . . 150
4.3.7 Algorithms for Automatic Classification . . . 151
4.4 Summary . . . 152
4.4.1 Related Work . . . 153
4.4.1.1 Classification by Adleman . . . 154
4.4.1.2 Classification by Bonfante et al . . . 155
4.4.1.3 Phylogenetic Classifications . . . 156
4.4.1.4 Classification by Spafford . . . 157
4.4.1.5 Classification by Weaver et al . . . 159
4.4.1.6 Industrial Classifications . . . 160
4.4.2 Comparisons with Related Work . . . 162
4.4.2.1 Comparison with Formal Classifications . . . 162
4.4.2.2 Comparison with Informal Classifications . . . 164
4.4.2.3 Classification of Models versus Classification of Computer Viruses164
4.4.2.4 General Comments on Affordance-based Classification . 166

5 Conclusion **169**
5.1 Novel Contributions . . . 169
5.2 Directions for Future Research . . . 171
5.2.1 Complexity of Detecting Metamorphic Computer Viruses . . . 172
5.2.2 Further Detection of Metamorphic Computer Viruses . . . 173
5.2.3 Detection of Virtualization by Metamorphic Code Generation . . 174
5.2.4 Modelling Reproduction at Different Abstraction Levels . . . 176
5.2.5 Metrics for Reliance on External Agency . . . 179
5.2.6 Strategies for Reproduction . . . 179
5.2.7 Advanced Metrics for Assisted Computer Virus Classification . . 180
5.2.8 Evaluation of Anti-virus Techniques . . . 181
5.2.9 Affordance-based Models and Multi-Reproducers . . . 183

A Intel 64 Specification **185**

B Unix Computer Virus Specification **203**

C Bacteriophage Specification **207**

D Anti-Virus Specification **215**

List of Figures **221**

Bibliography **223**

Index **239**

Abstract

Formal Models of Reproduction: from Computer Viruses to Artificial Life

Matthew Paul Webster

In this thesis we describe novel approaches to the formal description of systems which reproduce, and show that the resulting models have explanatory power and practical applications, particularly in the domain of computer virology. We start by generating a formal description of computer viruses based on formal methods and notations developed for software engineering. We then prove that our model can be used to detect metamorphic computer viruses, which are designed specifically to avoid well-established signature-based detection methods. Next, we move away from the specific case of reproducing programs, and consider formal models of reproducing things in general. We show that we can develop formal models of the ecology of a reproducer, based on a formalisation of Gibson's theory of affordances. These models can be classified and refined formally, and we show how refinements allow us to relate models in interesting ways. We then prove that there are restrictions and rules concerning classification based on assistance and triviality, and explore the philosophical implications of our theoretical results. We then apply our formal affordance-based reproduction models to the detection of computer viruses, showing that the different classifications of a computer virus reproduction model correspond to differences between anti-virus behaviour monitoring software. Therefore, we end the main part of the thesis in the same mode in which we started, tackling the real-life problem of computer virus detection. In the conclusion we lay out the novel contributions of this thesis, and explore directions for future research.

Acknowledgements

I thank my Ph.D. supervisor, Dr. Grant Malcolm, for his wisdom, encouragement and indispensable advice throughout the course of my studies. Most of the work in this thesis was shaped in some way by Grant, and I don't think any aspiring scientist could find a better mentor.

I am much indebted to my secondary Ph.D. supervisor, Dr. Alexei Lisitsa, and my thesis adviser, Prof. Michael Fisher, who have given me invaluable guidance throughout. I thank Dr. Ray Paton, who helped me a great deal when I applied to study for a Ph.D. There are many other people in the Department of Computer Science at the University of Liverpool who have helped me over the years — certainly far too many to name here. Suffice it to say, I owe this great institution a huge debt of thanks.

I must also thank Prof. Lt. Col. Éric Filiol, whose generous support, advice and encouragement are very much appreciated.

Finally, I must mention my partner, Katie: a person whose impact on me is so profound that I have difficulty even beginning to express it. For all your love, support and psychological debugging — thank you.

— Matt Webster, October 2008

Preface

During the final year Honours project for my Bachelor's degree at the University of Liverpool, for which I studied under the supervision of Dr. Alexei Lisitsa, I created a formal model of computer viruses using a formalism known as Abstract State Machines. This piqued my interest in the research area at the intersection of computer science and biology. At first I was interested particularly in the class of reproducing programs, which includes computer viruses, network worms, reproducing structures within cellular automata, artificial life simulations, and so on. However, the distinction between reproducing programs and reproducers (i.e., things that reproduce) in general is blurred, as we can apply the computational paradigm to biological, psychological, economical and other domains in which reproducing forms can be identified. For example, we can see a reproducing organism as an automaton, programmed by genes with the sole intent of creating copies of those genes. My interest in reproducing programs expanded naturally to an interest in computational approaches to the problem of describing reproduction.

Biological examples of reproduction are obvious and numerous, and include most living things. Psychological examples include *memes*, a term used to describe reproducing thought-forms such as catchy tunes or methods for ensuring survival. In the field of economics, firms have been identified as a kind of reproductive system. There are even more examples of reproduction in fields such as chemistry, e.g., seeding crystals or fire. One might even think of fixed points of mathematical functions as being a kind of reproduction, or of modelling reproduction at a very abstract level. The list of reproducers goes on and on. In fact, reproduction is encountered so commonly that one begins to suspect that either (i) it is a universally re-occurring (or perhaps, reproducing?) phenomenon; (ii) something about the psychological make-up of human beings makes us prone to observing cyclic phenomena in a way that we perceive as reproductive; or (iii) both of the above are true.

In order to focus my research, I identified a number of key questions that I wanted to address in order to reach a greater scientific understanding of reproductive systems:

1. Are there any better means of detecting reproducing malware?

2. What do we mean by "replication" and "reproduction"?

3. What, if any, are the bounds on replication and reproduction?

4. Are there different kinds of reproduction?

5. What kinds of formalisms are best suited to describe replicative and reproductive systems?

6. How can we describe reproduction differently in different systems?

7. Are there any formalisms in which reproduction (or some variant of it) is not expressible?

It was largely the pursuit of answers to the above questions that has directed my Ph.D. research, which has in turn generated the material in this thesis. Question 1 is addressed in Chapters 2 and 4, in which methods for malware detection are discussed. Questions 2–6 are addressed in Chapters 3 and 4, in which affordance-based reproduction models are introduced and discussed. Question 7 still remains untackled within this thesis, and is, perhaps, a direction for future research. The presentation of this thesis, therefore, follows an arc. In Chapter 2 we start "in the trenches", analysing computer viruses and developing ways to detect them. In Chapter 3 we move away from the specifics of computer viruses to consider reproduction in general. We present our formal affordance-based reproduction models, which we then take with us into Chapter 4, in which we show how their classification mirrors the technical problem of detecting computer viruses using behaviour monitors. Therefore we end in the same place we started: developing techniques for securing computer systems against unwanted reproducing programs.

Whilst I have attempted to answer most of the questions above, I certainly do not claim to have answered them completely. However, I hope that my humble attempts to answer these questions in the forthcoming chapters are a testament to the incredible complexity and diversity of problems at the intersection of computer science and biology, and an encouragement to other researchers to engage in the fascinating work to be conducted there.

Matt Webster, October 2008

Chapter 1: Introduction

The aim of this thesis is to describe novel approaches to the formal description of systems which reproduce, and show that the resulting models have explanatory power and practical applications, particularly in the domain of computer virology. We start by generating a formal description of computer viruses based on formal methods and notations developed for software engineering. We then demonstrate that our model can be used for detecting unwanted reproducing software. Next, we move away from the specific case of reproducing programs, and instead consider formal models of reproducing things and their environments, showing that we can classify and refine these models in interesting ways. Finally, we take these formal models and apply them to the real-life practical problem of detecting the behaviour of unwanted reproducing software, showing that the different classifications within the formal model correspond to differences between behaviour monitoring software. Therefore, the aim of the next two sections is to provide the reader with a broad background knowledge and history of the study of reproducing programs, and reproduction in general. In Section 1.3, we describe what we mean by formal techniques, and in Section 1.4, we end the chapter with an overview of the rest of the thesis.

1.1 Reproducing Programs...

We shall begin this section with an overview of some informal terms used in the literature. As with any informal terms, there is some laxity in the definition, and therefore the purpose of this discussion is to define and clarify these terms, as they will be used throughout this thesis.

Malicious software, or *malware*, is a general term that captures the notion of dangerous or unwanted programs, that typically execute without the legitimate users' consent. (Of course, the writer of malware becomes effectively an illegitimate user of any computer system that becomes a host to their malware.) *Reproducing programs* are a kind of computer program that are able to create copies of themselves within other stored programs. A *computer virus* is a reproducing program, and a *worm* is a program that

is able to create a copy of itself in a stored program which may be located elsewhere in a network of computers. Therefore, by these informal definitions, worms are a type of computer virus.

It is important to clarify that not all computer viruses are malicious; in fact, the early theoretical history of computer virology was to the contrary. One of the first descriptions of a computer virus in the academic literature was the work by Shoch & Hupp on the worm programs, which they proposed as an autonomous form of distributed computation, in which a program designed to solve a lengthy task could create a copy of itself, also capable of reproduction, within another node on the network in order to divide the work and shorten the length of time needed [131]. The same principle was also used for routine maintenance tasks (such as network diagnostics). The worm program was carefully designed to avoid impinging on other users; it would only commandeer a computer if it was not being used, and would retreat as soon as a user required it. Cohen also suggested several positive applications of computer viruses, including a compression virus that compresses executables when it spreads, thereby freeing up storage space [31, 33]. When the infected executable is run, the virus decompresses the file. Depending on the relative speeds of processing and storage access, the virus can even speed up the time taken to execute an infected program.

Therefore, not all computer viruses are malware. Since the term "malware" captures the notion of malicious software, any program written with the intent to access a computer against the wishes of the legitimate user must be considered to be malware. It is obvious that these programs are not necessarily reproducing, and therefore we know that not all malware programs are reproducing programs.

1.1.1 Motivations of Computer Virus Writers

We cannot ignore the fact that computer viruses are a product of human ingenuity, and though sociological and criminological factors play a part in mitigating the threat to our computer systems, we must assume that computer viruses will continue to thrive on our computers and networks. A similar assumption seems to permeate the academic literature on computer viruses, in that there is little mention of the virus writers[1]; viruses are described as though they were naturally-occurring phenomena that we must learn to control. For this reason, the rest of this thesis will discuss computer viruses in the detached manner used in much of the literature, without mention of the people who write viruses, or their motivations.

[1] Despite this trend, there has been some interesting work on the psychology and sociology of malware writers by Gordon, e.g., [63].

1.1.2 A Short History of Computer Viruses

Computer viruses came to prominence during the personal computer revolution of the 1980s [139]. It is likely that many of the first computer viruses were written as an intellectual game, the idea of a computer program that reproduces being a computational curiosity. Indeed, Elk Cloner, one of the first computer viruses, was created by an American high school student as a prank [38]. Since then, computer viruses have grown dramatically in technical sophistication, destructiveness and number, and the commonly-held belief in the computer security community is that this trend will continue [48]. The threat to computer systems posed by viruses has already spawned a multi-billion US dollar anti-virus software market, as well as causing billions of US dollars of damage all over the world [62, 2]. Therefore the development of technical solutions to the problem of detecting malware, such as those described in this thesis, is a timely and crucial part of computer science.

1.1.3 Academic Study of Computer Viruses

We have seen that computer viruses are (often harmful) computer programs that reproduce autonomously through computer file systems. In general, they are segments of a stored program that, when executed, are able to create a copy of themselves in another stored program. The canonical formal description of computer viruses was given by Cohen [32], in which computer viruses are described as a class of Turing machine strings capable of reproducing themselves (see Figure 1.1). In this thesis we use Cohen's definition when we refer to computer viruses.

Whilst the study of computer viruses began in formal, abstract, theoretical terms, much human labour has since been expended in the generation of software tools for the detection and removal of reproducing malware, more commonly known as *anti-virus software*. A common misconception is that computer viruses are simply an information security problem, like stack buffer overflows [113] or denial-of-service (DoS) [23] attacks, i.e., a contemporary problem best left for industry, or the vendors of commercial software, to solve. In fact, computer viruses are inherent in stored program computers, since they spread from program to program over time. Most modern computers are based on the stored program (von Neumann) model, and are therefore vulnerable to illicit reproducing programs. As the ability to process information increases, and the financial cost of computers decreases, we expect to reach a stage of ubiquity, or pervasiveness, at which point computer systems will be fully integrated into everyday life [67]. The control of reproducing programs across widespread, networked, stored program computers is a significant challenge. The challenges faced by researchers in

For all M and V,
the pair (M, V) is a viral set if and only if:
V is a non-empty set of Turing machine sequences and M is a Turing machine and
for each computer virus $v \in V$, for all histories of machine M,
    For all times t and cells j
    if
        the tape head is in front of cell j at time t and
        M is in its initial state at time t and
        the tape cells starting at i hold the virus v
    then
        there is a virus v' in V, a time $t' > t$, and place j' such that
            at place j' far enough away from v
            the tape cells starting at j' hold virus v
            and at some time t'' between time t and time t'
            v' is written by M.

Figure 1.1: Cohen's formal definition of computer viruses [32].

computer virology are also closely related to other disciplines. For instance, metamorphic computer viruses employ code obfuscation techniques to hide the intent of their code from static analysis detection techniques like signature scanning. The techniques for detecting metamorphic computer viruses are therefore related to the functions researched in subjects such as code optimisation and program transformation, in which code is mutated for different reasons.

Another common misconception is that research into computer viruses and their detection is likely to attract unwanted interest from those able or willing to create computer viruses, or that descriptions of computer viruses in the literature may provide ammunition to computer virus writers. The implication of this point of view is that open research into computer virology is a potential security risk. This attitude is in clear violation of one of the most fundamental tenets of cryptography, Kerckhoff's principle, which says that a cryptographic system should not depend on secrecy, and it should be able to fall into the enemy's hands without disadvantage. As Schneier describes,

> "...Kerckhoffs's principle *applies beyond codes and ciphers to security systems in general*[2]: every secret creates a potential failure point. Secrecy, in other words, is a prime cause of brittleness — and therefore something likely to make a system prone to catastrophic collapse. Conversely, openness provides ductility." [100]

Therefore, the application of Kerckhoff's principle to computer virology means that

[2]Emphasis added.

the details of the means by which computer viruses operate, and the means by which we defend our computer systems from unwanted code (which are intimately linked), should not be kept secret, and the open publication of information about computer viruses promotes the development of secure computer systems[3].

The growing sophistication of computer viruses, combined with the immediate security challenges posed by pervasive computing [67, 120, 145, 166, 107, 121], indicate the need for a precise, formal and open understanding of reproducing programs. The anti-virus software produced by industry serves an obvious present-day security concern, but the knowledge of the basic principles of computer viruses, their dangers, limitations and behaviour, must not be known only by a select group of industrial specialists, but rather must be open and easily accessible. This is essential to ensure the long-term security of our computer systems from reproducing programs [45].

1.1.4 Detection of Malware

Detection of computer viruses is a difficult problem. Cohen proved that detection of computer viruses in general is undecidable [32]; Chess & White proved that there exist computer viruses for which there are no detection algorithms that work without false positives [26]; Filiol [40] and Filiol & Josse [47] established the characteristics of some of these undetectable viruses; results from Adleman [4], Borello et al [17, 18], Spinellis [136] and Zuo et al [169, 72] show that even where computer virus detection is decidable, it can be intractable for certain computer viruses.

Despite the existence of computer viruses that are intractable or impossible to detect, many computer viruses are not so sophisticated, and can be detected tractably and reliably. Therefore, a primary motivation within the research into reproducing programs is the development of means of detection of those programs. Naturally, much research on this theme is conducted by the vendors of anti-virus software, but the specific details of the implementation of anti-virus software are often left unpublished[4] However, in recent years there has been a dramatic increase in the amount of literature published on computer virus detection methods, both by the academic and commercial communities. For example, the Journal in Computer Virology (published by Springer) was launched in 2005. Around the same time, Ször published a book on the art of computer virus research and defence, giving details of the many ways in which malware can be detected by commercial anti-virus software [139], and Filiol published a book on the formal foundations of computer virology [43]. Filiol described the different malware

[3] An in-depth discussion of this position can be found in the Preface of [43].

[4] This is possible due to the fact that they represent the trade secrets of the anti-virus software vendors, and careless talk is at the expense of commercial advantage.

detection techniques from a more abstract position, based on a taxonomy of the various approaches. The overview of the state of the art of malware detection presented here is based on this abstract perspective.

Malware detection techniques can be divided into those based on *static analysis*, in which the executable code within stored programs is analysed as data, and *dynamic analysis*, in which programs are analysed during execution. All methods of malware detection are prone to errors of two kinds[5]:

- *False positive*: a method that should only identify a particular set of programs instead identifies a program outside that set.
- *False negative*: a method that should identify a set of programs fails to identify at least one of the elements in the set.

We will now describe the various methods of malware detection, together with their advantages and disadvantages. As we shall see, a combination of the many different computer virus detection methods must be used in order to ensure the best possible protection.

1.1.4.1 Static Analysis

Signature scanning involves the extraction of signatures from known computer viruses. The signatures consists of numeric data, which in the ideal case uniquely identify a particular virus. Signatures must be sufficiently incriminating, so that they identify a particular virus or variant, and non-incriminating, so that they do not identify another virus or uninfected program. The advantages of signature scanning include a high efficiency, as it can be implemented using much-researched string-matching algorithms, and a low number of false positives. The main disadvantage is that this technique relies on a signature database, which can only contain signatures of known viruses. If the signature database is not updated regularly, or if the repository from which updates are made is corrupted, the effectiveness of the technique is severely compromised.

Spectral analysis is based on a statistical analysis of the varying occurrences of different instructions within a program. Computer viruses tend to have increased numbers of certain instructions, meaning that the relative occurrence of these instructions can be used as an heuristic for computer virus detection. For example, Filiol presents the Intel 64 assembly language instruction `xor` x`,` x which is commonly used to set the value of variable x to zero. Some metamorphic computer viruses will change this instruction to a less commonly used instruction, e.g., `mov` x`, 0` which has the same effect. Therefore

[5]These definitions are given formal meaning in the domain of computer virus detection by Filiol & Josse [47].

a higher-than-normal occurrence of the latter instruction could be used to flag a certain program as suspect. The main advantage of this approach is that it can be used to detect previously-unseen computer viruses. However, it is an heuristic, and therefore can often result in false positives.

Heuristic analysis is where a suspect program is checked for the occurrence of a particular set of instructions that correspond to some malicious behaviour. Heuristic analysis can therefore be seen as a kind of spectral analysis that is focussed upon a single instruction or instruction sequence. For example, within Unix shell scripts the statement `rm -R /*` deletes every file on the current device, and has few legitimate uses. Therefore, a suspect program containing such a command can be flagged as potential malware. The advantages and disadvantages of this approach are similar to spectral analysis: previously-unseen computer viruses can be detected, but there is a significant risk of false positives.

File integrity checking involves the creation of a database in which every sensitive file's name is stored along with the result of a hash function, e.g., MD5 or SHA-1. The hash function assigns different numbers to different inputs, meaning that any modification of an input results in a different output. The aim of this technique is to associate with executable files some value, which is checked regularly so that any modification of an executable file is detected. Since many computer viruses modify files when they infect them, file integrity checking can be used to detect illicit activity by a computer virus. In principle, file integrity checking can be used to detect any attempt by a computer virus to infect another file. However, there are several disadvantages. First, malware may be able to corrupt the database so that the number corresponding to a file is not changed upon infection. Second, there may be legitimate processes within the operating system which must modify executable files, e.g., compilers, and therefore there must be some authentication system by which these processes can register themselves with the file integrity checker as legitimate. Therefore, malware may be able to fake legitimacy. Third, not all computer viruses modify stored programs: companion viruses and computer viruses which exist only in memory are examples. Fourth, the hash functions used may be insecure, allowing the computer virus to fake legitimacy. Fifth, there may be numerous locations for data whose integrity is not checked, such as temporary or configuration files. In addition, the need to check legitimacy of file modification operations may result in false positives.

1.1.4.2 Dynamic Analysis

Behaviour monitoring involves the constant presence of a behaviour monitor, usually as part of an anti-virus program. Suspect programs can be monitored for activities often

exhibited by computer viruses, including opening executable files in read/write mode, modification of sensitive system files or becoming memory-resident. A main advantage of behaviour monitoring is that previously-unseen computer viruses can be detected, and even prevented from infecting any other files or executing payloads. There are several disadvantages. First, computer viruses may not access system resources in ways expected by the behaviour monitor. Second, there is a considerable demand on system resources in the constant monitoring of behaviour. Third, false positives are likely as legitimate programs, e.g., compilers, may display suspicious activity.

Code emulation involves the execution of suspect programs within a "sandbox" in order to determine the presence of a computer virus. A sandbox is a virtual machine, an isolated region of memory in which a computer virus could be executed without further infections. The aim is to monitor the behaviour and observe characteristics of computer viruses, including heuristics such as opening executable files, as well as known features of certain computer viruses, e.g., the state of the processor stack at a certain point in execution. Advantages of this approach include detection of computer viruses without further infection and dynamic analysis without the constant overhead of behaviour monitoring (i.e., code emulation detection can be performed during times of low processor load); however, the technique relies on the correct implementation of the sandbox, which may not be guaranteed. For instance, a computer virus could manipulate a sandbox during execution, resulting in an infection of the main, non-virtual computer system.

1.2 . . . and Other Reproducing Things

The concept of life is notoriously difficult to define. A textbook biological definition is that an organised, genetic unit capable of metabolism, reproduction and evolution, is alive [116]. However, this leaves the problem of what we mean by genetics, metabolism, reproduction and evolution. As well as the inherent problems of loose definitions like these, most definitions of life will either exclude some things that appear to be alive, or include some things which don't seem to be alive. For example, by the definition above, a worker bee (that cannot reproduce) might not be considered to be alive, despite the fact that, intuitively, an active, self-directed system like the worker bee would seem to qualify as alive.

Much of mainstream biology is concerned with life as we find it, i.e., life on Earth[6]. However, there are many systems which display life-like characteristics, such as metabol-

[6]Should we find life beyond Earth, then biology would presumably expand to incorporate the studies of the new life forms. As it is, the study of potential extra-terrestrial life is called *astrobiology*.

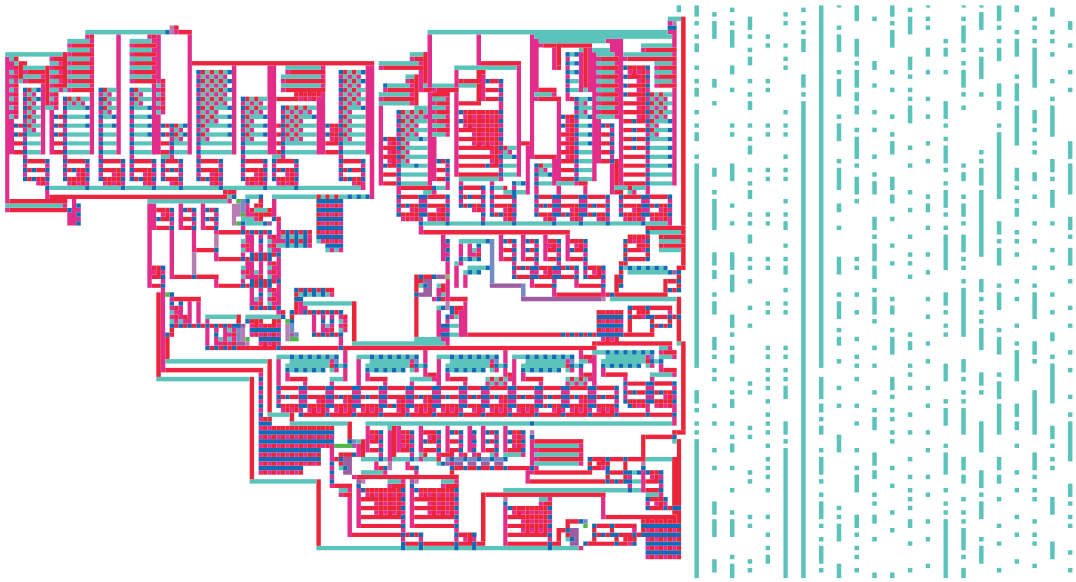


Figure 1.2: Von Neumann's reproducing automaton on a two-dimensional cellular automaton grid. The universal constructor on the left reads in the "tape" on the right, which instructs the constructor to produce a copy of itself and the tape. Image sources: [73, 112].

ism, reproduction and evolution, but which may not be biochemical in nature. The study of these systems is called Artificial Life [91, 7, 132].

Artificial life is therefore the study of life as it may be [7]. Much of artificial life is concerned with the generation of mathematical and/or computational models of systems which display life-like behaviour, although it includes the documentation of the life-like features of non-biological systems, such as firms [99] or computer viruses [135]. One of the first studies in this vein was undertaken by von Neumann, in his work on the construction of a fully-specified reproducing automaton [149]. Von Neumann's automaton was highly complex, requiring computational resources so extensive that a fully working implementation has only recently been developed [112]. Since then, cellular automata have been exploited by many researchers in the field to develop models of reproduction. Codd [30] vastly simplified von Neumann's original model, and Langton [89] removed the requirement for universal computation. The intellectual progeny of von Neumann's work continue to be developed within the field of artificial life [133], including exotic versions displaying evolution [129], sexual reproduction and parasitic infection [111]. Other notable artificial life systems include Tierra [117, 118], Core War [37], Cosmos [142] and related systems [133]: virtual computational environments in which computer programs can reproduce, mutate, combine and compete [33].

The field of artificial life is wide-ranging and multi-disciplinary, taking inspiration

from and inspiring researchers in biology, philosophy, mathematics, artificial intelligence and more [91, 132]. Much of the work in Chapters 3 and 4, in which we focus on a formal approach to modelling reproduction, is closely related to the theoretical, mathematical and philosophical branch of artificial life. As we noted earlier, reproduction is one of the signifiers of life in most definitions, and therefore the results of this thesis may be of interest to researchers in any of the fields listed in this section. The abstract, non-functional approach to modelling reproduction presented in Chapters 3 and 4 is also related to the fields of cybernetics [163] and systems theory [85], which study the organisation of systems independent of the substrate in which they are embodied.

1.3 Formal and Informal Approaches to Problem Solving

A central theme of this thesis is the application of formal approaches to questions about reproduction. Before we start to describe these approaches, it is necessary to describe what we mean by a "formal approach".

An approach to solving a problem can be said to be formal if it is based on some logical/mathematical description of the problem and its solution. If an approach is not formal, it is informal. The advantage of formal approaches over informal approaches is that their logical/mathematical nature gives the potential to prove interesting properties about a system. As long as we accept the given axioms, we must accept any conclusions which are drawn from them. Therefore, solving a problem within a formal system is a matter of choosing which axioms we want, so that we can phrase potential solutions to the problem as formal statements which may then be proven (incompleteness allowing [57]).

Clearly, formal approaches to solving problems have advantages over informal approaches. However, there are also disadvantages. For example, if the problem that is being solved is, by its nature, not well-defined, then it is likely that a formal system cannot easily be developed, since formal systems require axioms to be encoded within some language whose syntax and semantics are clear. If a formal system *is* possible, then there is the problem of the amount of time and effort required to (i) generate the formal system, and (ii) prove statements within it. So, an informal solution to a problem can exist where a formal solution might not, and an informal solution might be much more easily produced than a formal one. These are clear advantages of using informal systems over formal systems.

To clarify these points one can think of the (general) difference in approaches to solving problems in the sciences versus the humanities. In science the problems being tackled are well-defined (in general), and therefore formal solutions are the usual goal.

However, in the humanities, there may be more nebulous questions that are not as well-defined as those in science. Therefore, in science, formal systems are often favoured, because the systems being analysed can be modelled formally, and the results are conclusive. However, in the humanities, informal systems are often favoured, perhaps because otherwise solutions to problems could not even be attempted. This difference of approach can also be seen within science, where more general questions (such as "Are computer viruses alive?" [135]) require informal approaches, and more specific questions (such as "What is the computational nature of computer viruses?" [32, 4, 15]) can be answered using formal approaches.

In this thesis we hope to demonstrate the usefulness of formal approaches to (i) problems that at first seem complex and nebulous, such as "What is reproduction?"; and (ii) problems that at first appear to require little theoretical analysis, such as "How can we search for all instances of a reproducing program, and delete them from the computer?"

1.4 Overview of the Thesis

A classic structure for a Ph.D. thesis is as follows: an introduction, setting the scene and providing background information; a review of the literature related to the novel contribution of the thesis; the novel contribution itself; and finally, a conclusion, reviewing the novel contribution relative to the related work in the literature.

Whilst the main theme of this thesis is the construction of formal models of reproduction, the related work in the literature spans three distinct areas: detection of metamorphic computer viruses, models and classifications of reproduction, and classifications of computer viruses. Therefore the structure of the work presented in this thesis necessitates a departure from the classic Ph.D. thesis form. Here, we devote one chapter to each of the three main areas in which novel contributions have been made, with the relevant literature review and comparison located at the end of each chapter. We hope that the reader will permit this alternative structure, as it has been designed with the reader's needs in mind. A more classical structure would have been possible, of course, but perhaps would have been less logical, and therefore more difficult to read.

In **Chapter 2** we apply a formal software verification technique called algebraic specification to the problem of detecting a particular kind of computer virus that is able to change the syntax of its own code from generation to generation, called a metamorphic computer virus. We give an overview of the different kinds of code metamorphism, and describe the specification of a subset of the Intel 64 assembly programming language, which is used in the majority of personal computers worldwide, and a common

implementation language for computer viruses. We show that this formal specification can be used directly for dynamic analysis of computer virus code, and that it can be used to prove the equivalence or non-equivalence of programs. Under certain circumstances, which we call semi-equivalence, only a subset of the variables in a particular state are equal after execution of two programs. We prove that we can extend semi-equivalence to equivalence under certain circumstances, which enables detection of metamorphic computer viruses through static analysis. We give fully worked examples of detection using both dynamic and static analysis based on real-life metamorphic computer virus code, and we end the chapter with a critical appraisal of our approach relative to other approaches to metamorphic computer virus detection given in the literature.

In **Chapter 3** we begin by describing a method of classification of reproduction based on Gibson's theory of affordances. We demonstrate an informal approach, based on the division of "responsibility" for the self-description and reproductive mechanism within the act of reproduction. We show that this informal classification is deficient in a number of regards, and raises several interesting questions. We then attempt to clarify our classification and ontology by defining formal reproduction models in which a formal notion of affordances is used to attribute responsibility to various actors within the reproduction system. We give formal definitions of assistance and triviality with respect to our formal affordance-based reproduction models, and prove that every assisted reproduction model has a related model in which the same reproductive act is described, but is classified as unassisted. This theorem is complemented by another theorem which says that any unassisted model has a related model, which again describes the same act of reproduction, but is classified as assisted. At every stage we illustrate our approach with worked examples from the fields of biology, computer virology, and artificial life, in order to demonstrate the applicability of, and relevance to, real examples of reproduction. We conclude the chapter with an overview of other formal and informal reproduction models and classifications from the literature, and compare our work with the related work based on a number of different criteria. Finally, we discuss some of the philosophical implications of our formal models and classification.

In **Chapter 4** we describe a practical application of the formal affordance-based reproduction models from Chapter 3. We define affordance-based computer virus reproduction models as those models in which the computer virus is the reproducer. We demonstrate how these models can be constructed at different levels of abstraction: low-level models specify each action as the execution of an instruction or a statement, and are suitable for short, simple computer viruses; high-level models specify abstract actions corresponding to specific behaviours, e.g., opening a file for reading and writing. We give examples of low- and high-level models applied to real-life computer viruses

programmed as Unix shell scripts, Visual Basic scripts, Java executables and $x86$ assembly language executables, illustrating the flexibility of the approach. In each case we classify the models as assisted and unassisted, and describe how the difference between the two is analogous to classification as detectable or undetectable by anti-virus behaviour monitors. Therefore, the ability of affordance-based reproduction models to allow multiple classifications of the same reproducer (explored in Chapters 3 and 4) can mirror the multiple classifications of the same computer virus as detectable or undetectable, depending on the behaviour monitor's ability. Behaviour monitors monitor interactions between a computer virus and its environment, e.g., when a computer opens a file, it must request a handle to that file from the operating system. We propose that if an anti-virus behaviour monitor is able to detect a given behaviour of a computer virus, it is actually detecting communication between a computer virus and some element in its environment, e.g., the operating system. It is therefore logical to specify this element as a separate entity in the reproduction model describing this computer virus. In the case where the behaviour monitor cannot detect a behaviour, then the monitor cannot "distinguish" between the computer virus and the resource with which it interacts. With this in mind, we describe how automatic classification can be achieved, either by static or dynamic analysis, and give worked examples of the two different methods for Visual Basic script computer viruses. We describe how metrics for classification can be developed to sub-classify assisted computer viruses, discuss algorithmic implementation issues, and describe how this approach might be used to improve the efficiency of anti-virus software. Finally, we given an overview of other computer virus classifications in the literature, and compare our approach with the others on a number of different criteria.

Much of the novel research in this thesis has been peer reviewed and published previously [154, 153, 158, 159, 160, 156], at various stages of development. This thesis contains the most up-to-date account of the research conducted during the author's Ph.D.[7], as well as significant sections that do not appear elsewhere, including this chapter, Chapter 5 and the literature reviews and comparisons at the ends of Chapters 2–4. Chapter 2 includes research published in [154, 153, 158], Chapter 3 covers [159, 160], and Chapter 4 is based on [159, 160, 156].

1.4.1 Computer Viruses and Artificial Life

As we have described, this thesis is concerned broadly with formal models of reproduction. We start with formal models of computer viruses in Chapter 2, which we

[7] In between the initial and final submissions of this thesis, the work presented has been extended further [161, 157, 155].

extend to formal models of reproduction in general in Chapter 3, before returning to formal models of computer viruses in Chapter 4. However, this is not the first time that computer viruses and artificial life have been linked.

In his book, "Computer Viruses, Artificial Life and Evolution" [97], Ludwig describes in detail the relationship between computer viruses, artificial life and the study of evolutionary processes. The scope of the book is wide, and covers several fundamental philosophical issues: reproduction, emergence, evolution, the philosophy of life and evolution. There is much overlap, therefore, in the subject material of Ludwig's book and this thesis. While philosophical issues are highly relevant to this work and appear several times (particularly in Chapter 3), this thesis aims primarily to describe a number of novel contributions to the fields of computer virology and artificial life. For more information on the philosophical issues at the heart of this thesis (as well as the fields of computer virology and artificial life), the reader is encouraged to read Ludwig's work for an excellent overview[8].

Computer viruses are also given as a form of artificial life by Spafford; this work is described in detail in Section 4.4.1.4.

1.4.2 A Note on the Inclusion of Computer Virus Code

For the demonstration of various computer virus detection methods, it has been necessary to include in this thesis excerpts from the source code of some reproducing malware for illustrative purposes, in the vein of Cohen [33] and Filiol [43], who have published virus source code for similar reasons. In order to prevent dissemination of exploitable code we have omitted significant sections of code, and in the remaining code we have introduced subtle errors. Therefore, the source code in this paper cannot be executed, but can be used by the reader to verify the methods for computer virus detection, modelling and classification that we describe.

The computer virus code used in this thesis was obtained from a number of sources, including "VX Heavens" [1] and "The Quine Page" [143]. We would like to thank Bruce Ediger for his permission to include a variant of his quine program. The variant is shown in Figure 4.1, and the original appeared in [143].

[8] Ludwig has also published two excellent introductory texts on computer viruses [95, 96].

Chapter 2: Formal Detection of Metamorphic Computer Viruses

2.1 Introduction

As we saw in the previous chapter, computer viruses are (often harmful) computer programs that replicate autonomously through computer file systems. They are typically designed to replicate without the user's consent, and are able to damage data or software on infected machines. In general, computer viruses replicate using the legitimate infrastructure of an operating system (e.g., disk input/output routines), and consequently the spread of computer viruses is difficult to prevent absolutely without restricting the operating system (OS) in some way.

Academic study of computer viruses began in 1987, when Cohen defined an abstract computer virus and gave a proof of the undecidability of computer virus detection, proving that there would be no detection-based panacea for the computer virus problem [31]. Preventing computer viruses through the use of severely restricted operating systems is possible, since the computer virus can only work by modifying stored programs in memory so that they contain a copy of the virus. However, the computer architecture that allows program creation and modification is at the core of the flexibility and efficiency of modern computers; for example, no compiler could run without the creation and modification of stored programs. Thus, finding a cure for computer viruses by restricting their environments is impractical, and reliable detection is impossible. Therefore the fight to stop illicit computer viruses becomes a question of optimization, i.e., "How can we best protect ourselves against computer viruses?" In addition, tractability is a primary concern due to the trade-off between efficiency and thoroughness of anti-virus (AV) scanners. With this and the sustained proliferation of computer viruses in mind, general theories of computer viruses and their environments would be useful. Formal specification of the behaviour of different virus types can provide insight to developers of anti-virus software by highlighting the commonality between different computer viruses, e.g., by encouraging the reuse of detection and disinfection methods. Knowl-

edge of which specific details of the implementation of computer systems afford survival for viruses can be derived from formal models, and can be used to restrict computer virus behaviour through the development of systems that are inherently (and provably) more secure.

2.1.1 Algebraic Specification

The formalisations of computer viruses presented in this chapter are based on algebraic specification. Universal algebra is the branch of mathematics that deals the abstract definition of algebras. An algebra consists of a carrier set S together with a finite set Υ of operations on that set [58]. Many common mathematical structures such as Peano arithmetic, groups, rings, etc., are examples of algebras. Algebras are very similar to the concept of abstract data types in computer science, since an abstract data type consists of a set of data (cf. the carrier set S) together with a set of functions (cf. the set of operations Υ) which manipulate that data.

Therefore, algebraic specification can be used to give a semantics for abstract data types. Based on this, we can use algebraic specification to give semantics for a programming language if we specify an abstract data type in which there is a sort of data corresponding to the set of *stores*, in which the values of variables are stored, and a store evaluation operation which takes a store s and a variable name v and returns the value of v in the store s. This is one method of algebraic specification of programming languages that is enabled by Maude, a formal high-level language based on rewriting logic and algebraic specification [29].

Therefore, algebraic specification provides a way of defining the formal semantics of algorithms and programming languages, and in this chapter we use it to give a formal definition of computer viruses, as well as to explore means of detecting metamorphic computer viruses (MCVs), which change their syntax (whilst keeping their behaviour constant) in order to evade anti-virus signature-based detection.

2.1.2 Chapter Overview

In Section 2.2 we introduce the problem of metamorphic computer virus detection, a difficult theoretical problem[1]in computer virology for which there is yet no standard solution [140, 139]. Then, in Sections 2.4–2.8 we describe an approach towards the detection of metamorphic computer virus using an algebraic specification of a subset of the Intel®64 and IA-32 Architectures assembly programming language instruction

[1]Indeed, detection of metamorphic computer viruses is undecidable in general, as proven by Chess & White [26].

set[2]. In Section 2.4 an overview of the specification is given, and in Section 2.5 we introduce formal notions of equivalence and semi-equivalence of instructions and instruction sequences. The Maude specification, when combined with the Maude term rewriting engine, can be used as an interpreter for programs in Intel 64, and this in turn can be used for dynamic analysis of computer viruses. In Section 2.6 this dynamic analysis is used to prove the equivalence and semi-equivalence of real-life metamorphic computer virus code fragments. In Section 2.7 we prove that semi-equivalence can be extended to equivalence under certain conditions ("equivalence-in-context"), and show how these results can be applied to static analysis of metamorphic computer viruses. Then, in Section 2.8 we describe how the specification of Intel 64 can be applied to the detection of metamorphic computer virus detection through static and dynamic analysis. Finally, in Section 2.9 we give a summary and critical appraisal of our approach through a comparison with related work on the problem of metamorphic computer virus detection.

2.2 Metamorphic Computer Viruses

The undecidability of computer virus detection is one of the oldest results in the field of computer virology, but anti-virus software has traditionally compensated for this by exploiting a weakness common to many computer viruses: constant syntax. The string of binary digits corresponding to a particular computer virus usually remains unchanged from one generation to the next, as does the placement of the computer virus within the infected executable file, e.g., one particular virus might be placed at the start of any executable it infects. Anti-virus software would search executable files for virus "signatures" – strings of bits that correspond to a particular virus – at the usual sites of infection within executables files. The presence of a signature would signal that the executable was infected, and that further steps needed to be taken to disinfect the file.

In order to avoid detection, the writers of computer viruses began to develop ways of obfuscating the suspect virus code. One attempt at this, the polymorphic computer virus, which changes its syntactic (binary digit) representation using encryption, fails to remain hidden from signature scanners once its means of decryption has been discovered. Once decrypted, all generations of polymorphic computer viruses look alike, and the signature-based approach to detection can be used. A more powerful means of

[2]Intel®64 consists of an extension of IA-32 to facilitate 64-bit memory addressing [75]. There are a few extra instructions in Intel®64 that are given to enable use of the new memory structure, but these instructions are not used in this chapter. However, the instructions used here are common to both architectures, and so for the sake of consistency, the Intel®64/IA-32 architectures will be henceforth referred to as "Intel 64".

detection avoidance is employed by the metamorphic computer virus. Each successive generation of a metamorphic computer virus modifies the syntax, but leaves the semantics unchanged. (Any two generations of the same metamorphic computer virus that differ syntactically are called *allomorphs*.) In this way the behaviour of each successive generation is the same, but the virus appears to be different. Thus, it becomes much more difficult for anti-virus software to detect a metamorphic computer virus using a signature-based approach, and it is not possible to keep a record of all possible generations because there may be an infinite number. In this way the metamorphic computer virus can successfully avoid detection [140].

Metamorphic computer viruses are particularly difficult to detect; there are known cases where detection is computationally intractable [136, 17, 18, 169, 72], or even undecidable [26, 40, 47]. Not all metamorphic computer viruses are so difficult to detect, however, and therefore research into practical ways to detect metamorphic computer viruses is essential.

2.2.1 Types of Code Metamorphosis

Metamorphic computer viruses conceal their code from anti-virus software using a variety of semantics-preserving, syntax-mutating methods [88, 140]. A non-exhaustive list of the different kinds of code metamorphosis is given in order to demonstrate the many and varied ways in which metamorphic computer viruses can use syntactic camouflage to defend themselves against static analysis based detection. Several of these types are also described by Lakhotia & Mohammed [88]. Where appropriate, we give examples of the metamorphoses using the Intel 64 assembly language.

2.2.1.1 Junk Code Insertion

Junk code is code that is superfluous to the main function(s) of the virus, and is inserted to create syntactic variants. There are different types of junk code, including but not limited to:

- Code that reverses the effects of a previous instruction or instructions, thus making the previous instruction(s) and the inverse code into junk. For example, the instruction sequence `xchg eax,ebx ; xchg eax,ebx` — which swaps the values in registers `eax` and `ebx` twice — would fall under this category. (Note that, throughout this chapter, we use a semicolon (`;`) to indicate sequential composition of assembly language instructions.)

- Code that performs a computation that is not utilised in any of the outputs of the program. For example, the first instruction in the following instruction list

does nothing as the result is overwritten by the next instruction: `mov eax,0 ; mov eax,ebx`.

2.2.1.2 Variable Renaming

Variables are renamed in successive generations of metamorphic computer viruses such as Win9x.Regswap [140]. For instance, `mov eax,0 ; push eax ; pop ebx` could be replaced by the equivalent instruction sequence `mov ecx,0 ; push ecx ; pop ebx`.

2.2.1.3 Unconditional Jump Insertion

A block of instructions is broken up into more than one smaller blocks of instructions linked by unconditional jumps. For example:

```
pop edx
mov edi,0004h
mov esi,ebp
mov eax,000Ch
```

```
pop edx
jmp label1
label2:
jmp label3
label1:
mov edi,0004h
mov esi,ebp
jmp label2
label3:
mov eax,000Ch
```

2.2.1.4 Instruction Reordering

Blocks of data-independent instructions are reordered to create syntactic variants. For example, `mov eax,ebx ; mov esi,edi` can be reordered to `mov esi,edi ; mov eax,ebx`.

2.2.1.5 Pseudo-Conditional Jump Insertion

A sequence of instructions ends in a conditional jump that depends entirely on information encoded in the preceding instructions. An example of this would be the following instruction sequence, `mov eax,20 ; sub eax,20 ; je label1`, in which the conditional jump `je` ("jump if the zero flag is set to 1") is effectively unconditional because the preceding instructions always set the zero flag to 1.

2.2.1.6 Arithmetical/Boolean Mutation

Arithmetical and Boolean operations can be easily mutated into other, equivalent forms. A good example of this can be found in the Win9x.Zmorph.A virus (see Figure 2.2, Section 2.6.2).

2.2.1.7 Payload Mutation

Some viruses only reproduce on certain days of the week, or when the hour of the day is an even number, for example. These conditionalities can be mutated by a metamorphic computer virus. The payload of the virus could also be mutated.

2.2.1.8 Pseudo Branching

Here, the same code is executed whether the condition of a conditional jump is true or not[3]. For example, the following two code fragments are equivalent with respect to the `eax` register:

```
je label1              mov eax, 435098
mov eax, 435098        sub eax, 340934
sub eax, 340934         ...
jmp label2
label1:
mov eax, 435098
sub eax, 340934
label2:
 ...
```

2.3 Algebraic Specification in Maude

Maude is a formal high-level language based on rewriting logic and algebraic specification [29]. Maude is strongly related to its predecessor, OBJ [60], a formal notation and theorem prover based on equational logic and algebraic specification. Like OBJ, Maude can be used for algebraic specification [105], as the operators of an algebra can be specified in terms of the sorts of their operands and values, and given meaning (i.e., turned into operations) using equations. These collections of sort and operation specifications

[3]This form of metamorphism has not been seen in any metamorphic computer virus, to the author's knowledge. It is included as a likely future development of metamorphic computer viruses. This is justified with the following quote from Filiol et al [45] on the ethics of the computer virology community: "We cannot rely on a 'wait and see' approach, but we must anticipate technological evolutions."

are called *modules*. Within these modules we can define the syntax and semantics of operations, which represent the behaviour of the system we wish to describe.

For example, the syntax of the natural numbers in Peano notation could be laid out in Maude as follows:

```
fmod PEANO is
  sort Nat .
  op  0 : -> Nat .
  op  s_ : Nat -> Nat .
  op  _+_ : Nat Nat -> Nat .
endfm
```

The first line simply introduces the name (`PEANO`) of the specification. The keyword `fmod` denotes that this is a functional module, i.e., it uses equations to give semantics to its operators. This specification uses only one sort of data: natural numbers, whose name (`Nat`) is declared in the second line. The next three lines declare three operators. The first, `0` is a nullary operator (i.e., a constant) and can be used by other non-nullary operators (which have an operand of sort `Nat`) to generate more complex terms. `s_` is a unary operator that takes a `Nat` and returns a `Nat`, and `_+_` is an infix binary operator that takes two `Nat`s and returns a `Nat`. `s_` is the successor function, and `_+_` is addition.

In Maude the semantics of operators can be given using equations, which are used by the Maude interpreter as term rewriting rules. An equation in Maude has the form

```
eq t1 = t2 .
```

where `t1` and `t2` are terms of the same sort. So, if we wanted to give the semantics for the `_+_` operator above, we could do this as follows:

```
fmod PEANO-SEMANTICS is
  using PEANO . *** Import the PEANO module
  vars M N : Nat .
  eq  M + 0     =  M .
  eq  M + s(N)  =  s(M + N) .
endfm
```

The first equation states that any term of the form `M + 0` is equal to a term of the form `M`, i.e., we have stated Peano's axiom of natural numbers, that $x + 0 = x$. The second equation states that terms of the form `M + s(N)` are equal to terms of the form `s(M + N)`, which is another of Peano's axioms for natural numbers.

Now we have specified the semantics of addition, or rather, we have made the `_+_` operator behave as an operation which returns the value of the sum of two natural number operands in Peano notation. The `=` is effectively a rewriting operator, showing that the term on the left is rewritten to the term on the right. Using the Maude interpreter, we can perform a reduction (a sequence of rewrites) in order to reduce a term to the most reduced form possible, i.e., the Maude interpreter keeps applying rewrite rules as long as there is a rewrite rule that will apply. A typical reduction using the specification above would be as follows,

```
Maude> reduce s(s(s(0))) + s(s(0)) .
   ==>
       s(s(s(s(0))) + s(0))
   ==>
       s(s(s(s(s(0))) + 0))
   ==>
       s(s(s(s(s(0)))))
```

where `==>` denotes the application of a rewrite rule from the specification.

An important notion in Maude is that of reduction as proof. Since each of the equations above holds for the natural numbers, the above reduction is a proof that "$s(s(s(0))) + s(s(0)) = s(s(s(s(s(0)))))$", or "$3 + 2 = 5$" in Arabic numerals.

2.4 Specifying Intel 64 Assembly Language

Algebraic specification of imperative programs is one of the many uses of Maude [105]. In a similar way to the example of Peano's natural numbers above, in which the semantics of the `_+_` operator was defined in Maude using equations, the semantics of programming language statements and functions can be defined using equations in Maude. Goguen & Malcolm describe an approach based on *store semantics*, similar to denotational semantics, in which the semantics of a programming language statement can be determined by the effects on variables relative to a store (i.e., state) [58]. In this section we present an algebraic specification of a subset of the Intel 64 assembly language instruction set based on store semantics. (The complete algebraic specification can be seen in Appendix A.)

The algebraic specification in Maude can be used to calculate the effects on any variable of any instruction sequence. Thus it is possible to prove equivalence of Intel 64 assembly language instruction sequences by applying reductions — sequences of equational rewrites — using the Maude term rewriting engine.

The Intel 64 architecture is used by the vast majority of personal computers worldwide, and it follows that many computer viruses will (at some point in their reproductive cycle) be manifest as a sequence of Intel 64 instructions. This section describes how a subset of Intel 64 has been formally specified algebraically using Maude.

2.4.1 Specifying the Syntax of Intel 64

We specify the syntax of Intel 64 assembly language instructions in Maude by declaring operations for each assembly language instruction. Since each instruction may work with program variables[4] (e.g., `eax`) or integers (e.g., `430549402`), we begin our specification by stating that these three sorts of data are used:

```
sorts Instruction Variable Int .
```

Intel 64 assembly language instructions can have either zero, one or two operands. If there are two operands, we call the first one the *destination* operand, and the second the *source* operand. The reason is that the source operand specifies an input program variable or integer for the instruction, and the destination operand specifies an output variable. Since the source operand can be either a variable or an integer, we can make the specification of Intel 64 simpler by declaring program variables and integers to be types of expression. We do this using by declaring a subsort:

```
subsort Variable Int < Expression .
```

We can now define the syntax of Intel 64 instructions in Maude. For example,

```
op mov_,_ : Variable Expression -> Instruction .
```

says that `mov` v`,` e is an instruction for any program variable v and expression e. We can define program variables `eax` and `ebx`, representing the Intel architecture registers of the same name, as follows:

```
ops eax ebx : -> Variable .
```

The sort `Variable` is a subsort of `Expression`, meaning that every program variable is an expression. Therefore, we have defined given a formal syntax for a part of the Intel 64 assembly language, in which assembly language statements are valid terms of sort `Instruction`, e.g.:

[4] "Variable" has a double-meaning in this context, as it refers to the "mathematical" variables used in Maude equations, as well as the variables used in programs. Therefore, to avoid any confusion we will refer to the latter as "program variables".

```
mov eax,ebx
mov ebx,eax
mov eax,eax
```

Likewise, we can define the syntax of other assembly language instructions, e.g.:

```
op add_,_ : Variable Expression -> Instruction .
op push_  : Expression          -> Instruction .
op nop    :                     -> Instruction .
```

Therefore, the following are all recognisable with our Maude specification of the syntax of Intel 64:

```
add ecx,edx
push eax
nop
```

Since we have declared the sort of integers (a built-in sort in Maude) as a subsort of expression, instructions with literal numeric values are also recognisable with our Maude syntax, e.g.:

```
mov eax, 0
add ecx, 43409924
push 3143
```

The syntax of a subset of Intel 64 is stored in a module called `I-64-SYNTAX`, and can be seen in Appendix A.

2.4.2 Specifying the Semantics of Intel 64

In Intel 64, as in any other imperative programming language, computation is achieved by updating the state of the machine that interprets the instructions of the language. We use the sort `Store` to represent this state, which comprises the values stored in the registers, stack, and various flags of the Intel 64 architecture. Notionally, we can think of a store as a function mapping variable names to values:

```
op _[[_]] : Store Expression -> Int .
```

The double-bracket operator takes a store and an expression (e.g., a number or a literal numeric value) and returns the integer value of that expression within that store.

Instructions (when executed) can modify the store, which we express in Maude by specifying the `_;_` operator:

```
op _;_ : Store Instruction -> Store .
```

Therefore we can take some generalised store, `s`, and evaluate the effects of an instruction. This mirrors classical denotational semantics, where an instruction sequence denotes a function that takes a starting state as an argument, and returns the updated state that results from running the instruction sequence in the starting state. Using these two operators, we can define equations in Maude which express the semantics of Intel 64 instructions, e.g.,

```
eq S ; mov V,E [[V]] = S[[E]] .                    (2.1)
```

can be read as, "the value of any program variable `V` after executing `mov V,E` in any store `S` is equal to the value of expression `E` in the original store `S`." We also need to specify that the values of all other variables are unchanged. We can do this using a conditional equation in Maude:

```
ceq S ; mov V1,E [[V2]] = S[[V2]] if V1 =/= V2 .   (2.2)
```

Therefore, we have specified that the value of any variable `V2` is unchanged after executing `mov V1,E` in any store `S`, as long as `V1` $\neq$ `V2`.

As we described earlier, Maude interprets equations as rewrite rules, in which the term on the left-hand side is rewritten to the term on the right-hand side. Therefore, by applying a sequence of rewrites to certain terms, we can evaluate the effects of instructions on a generalised store. We can extend this technique to sequences of instructions by overloading the `_;_` operator, so that it can be used to denote sequential composition of instructions. To do this, we declare a sort of `InstructionSequence`s, of which ordinary `Instruction`s are a subsort, before defining the second possible use of the `_;_` operator:

```
sort     InstructionSequence .
subsort  Instruction < InstructionSequence .
op _;_ : InstructionSequence InstructionSequence
         -> InstructionSequence .
```

For example, we may wish to know the effect of the following instruction sequence on a store:

```
mov ecx, eax ; mov eax, ebx ; mov ebx, ecx
```

The `mov` instruction assigns the value of its right-hand (source) operand to the variable in its left-hand (destination) operand. Based on this informal definition of the semantics

of `mov`, we can see that the instruction sequence above swaps the values of program variables `eax` and `ebx`. We can prove this using our formal semantics in Maude by performing reductions:

```
Maude> reduce s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[eax]] .
         ==>
s ; mov ecx, eax ; mov eax, ebx  [[eax]]
         ==>
s ; mov ecx, eax [[ebx]]
         ==>
s[[ebx]]
```

We ask Maude to reduce the term in the first line. The Maude term rewriting engine then applies (2.2), followed by (2.1) and finally (2.2) again to get the result: `s[[ebx]]`. Therefore, we know that the value of `eax` in any store `S` after executing the instruction sequence is equal to the original value of `ebx` in store `S`, i.e., `eax` now has the value of `ebx`. Similarly, we can prove that `ebx` gets the value of `eax`:

```
Maude> reduce s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ebx]] .
         ==>
s ; mov ecx, eax ; mov eax, ebx [[ecx]]
         ==>
s ; mov ecx, eax [[ecx]]
         ==>
s[[eax]]
```

We have therefore proven formally that the instruction sequence above swaps the values of `eax` and `ebx`. As we shall see in the coming sections, we can apply this technique to determine whether any two programs are equivalent.

2.4.2.1 Intel 64 Stack Semantics

The stack is a special type of program variable within the Intel 64 architecture, since it is essentially a sequence of integer values. We define an operator, `next`, which lets us build up lists of integers:

```
op _next_ : Int Stack -> Stack .
```

We also define a constant, `stackBase`, which denotes the base (i.e., bottom) of the stack:

```
op stackBase : -> Stack .
```

Therefore, all of the following are recognisable stack states within our specification:

```
stackBase
3 next stackBase
10 next 3 next stackBase
```

Stacks are lists of variables, and therefore we have to define the update semantics for the stack program variable differently from other program variables. We define a special form of the double-bracket operator:

```
op _[[stack]] : Store -> Stack .
```

We can then define the semantics of instructions which affect the state of the stack. The instruction `push` e pushes the value of expression e onto the stack. We specify this as follows:

```
eq S ; push E [[stack]] = S[[E]] next S[[stack]] .
```

We can also specify that other instructions, such as `mov V,E`, do not affect the state of the stack:

```
eq S ; mov V,E [[stack]] = S[[stack]] .
```

In other words, the value of the stack program variable after executing any `mov` is unchanged.

2.4.3 Using the Maude Specification as an Interpreter

Using the above techniques we can calculate the effects of any sequence of instructions on any program variable. The full specification, which can be seen in Appendix A, contains definitions of the syntax and semantics of the `MOV`, `ADD`, `SUB`, `XOR`, `TEST`, `AND`, `OR`, `PUSH`, `POP` and `NOP` instructions. In principle, there is no reason to stop here; it would be quite feasible to specify the semantics of Intel 64 in its entirety using Maude. Indeed, equational logic formalisms such as Maude have been shown to be a useful tool for the specification of imperative languages [105, 60, 58].

When the syntax and semantics of a programming language are defined in a formal executable language such as Maude, an interpreter and program analysis tool for that programming language are obtained essentially for free [105, 104]. In the example above, we proved that the effect of an instruction sequence was to swap the values of two variables by performing reductions in Maude. In fact, reductions turn the static Maude specification of a programming language into an executable specification, which can then be used as an interpreter, i.e., we can calculate the effects of arbitrary programs by evaluating them relative to variables in the store.

2.5 Equivalence of Instruction Sequences

Metamorphic computer viruses change their syntax without changing their behaviour when they reproduce, so we are particularly interested in applying our semantics for Intel 64 to show that two segments of code have the same behaviour.

We begin by defining notions of equivalent and semi-equivalent behaviour for code segments. Let S, V, and I denote the sets of all stores, variables, and instructions, respectively; the variables V include the registers, stack and flags of Intel 64. Let $\_[\![\_]\!] : S \times V \to \mathbb{Z}$ and $\_;\_ : S \times I \to S$, so that $s;p$ denotes the state of an updated store after executing instruction sequence p in store s, and $s;p[\![v]\!]$ denotes the value of the variable v in updated store $s;p$, as described in the Maude specification outlined above.

Two stores are *equivalent* if and only if every variable has the same value, and *semi-equivalent* if and only if a subset of variables have the same values.

Definition 1. *For $W \subseteq V$, stores s_1 and s_2 are* semi-equivalent with respect to W, *written $s_1 \equiv_W s_2$, iff for all variables $v \in W$,*

$$s_1[\![v]\!] = s_2[\![v]\!] \ .$$

In the case that $W = V$, we say that s_1 is equivalent to s_2, *and write $s_1 \equiv s_2$.*

Furthermore, we say that two sequences of instructions are equivalent if and only if they behave equivalently with respect to the set of variables in the store, and that they are semi-equivalent if and only if they behave equivalently with respect to a subset of the set of variables in the store.

Definition 2. *For $W \subseteq V$, instruction sequences p_1 and p_2 are* semi-equivalent with respect to W, *written $p_1 \equiv_W p_2$, iff for all stores s, and all variables $v \in W$:*

$$s;p_1[\![v]\!] = s;p_2[\![v]\!] \ .$$

In the case that $W = V$, we say that p_1 is equivalent to p_2, *and write $p_1 \equiv p_2$.*

Note that these notions of equivalence are in fact equivalence relations.

Our end goal is to be able to prove that two allomorphic sequences of code are equivalent. If $p_1 \equiv_W p_2$ then these instruction sequences may have different effects on variables that are not in W. However, if these instruction sequences are composed with another instruction sequence ψ whose behaviour does not depend on such variables, then we may have:

$$p_1;\psi \equiv p_2;\psi \ .$$

If these conditions are met by some p_1, p_2 and ψ then we say that p_1 and p_2 are *equivalent in context of* ψ.

For the purposes of static analysis, we identify the variables that are read or written to by instructions. We identify $V_{out}(\theta)$ as the set of variables that could be modified by some instruction θ.

Definition 3. *For an instruction* θ*, define* $V_{out}(\theta)$ *by* $v \in V_{out}(\theta)$ *iff there is an* $s \in S$ *such that* $s; \theta[\![v]\!] \neq s[\![v]\!]$.

Example 1. *We can determine the value of* V_{out} *for an instruction using the Maude specification of Intel 64. Suppose we wish to know the value of* V_{out}(`mov v1,v2`)*. By the above definition, we must show that for every program variable* $v \in V_{out}$(`mov v1,v2`) *that* v *is different after executing* `mov v1,v2` *in some store* s*. For example, after inspecting the Maude specification we may suspect that* `v1` *is in* V_{out}(`mov v1,v2`)*. We can prove this by assuming that the values of program variables* `v1` *and* `v2` *in some store* `s` *are different. We can express this in Maude notation as*

```
eq s[[v1]] = value1 .
eq s[[v2]] = value2 .
```

where `value1` *and* `value2` *are the (numeric) values of* `v1` *and* `v2` *respectively. Then, by performing reductions in Maude we can calculate the value of* `v1` *before and after executing* `mov v1,v2`*:*

```
reduce s[[v1]] .
result Int: value1
reduce s ; mov v1, v2[[v1]] .
result Int: value2
```

These reductions tell us that the value of v1 has changed from `value1` *to* `value2` *by executing* `mov v1,v2`*. Therefore, we know that* `v1` $\in V_{out}$(`mov v1,v2`)*.*

The same process can be applied to show that the instruction pointer program variable `ip` $\in V_{out}$(`mov v1,v2`) and that $v' \notin V_{out}$(`mov v1,v2`) for all v' not equal to `v1` or `ip`, i.e., V_{out}(`mov v1,v2`) $= \{$`v1`, `ip`$\}$. The complete Maude proof scripts can be seen in Appendix A.

Now that we have defined $V_{out}(\theta)$ as the set of variables that can be affected by the execution of instruction θ, we want $V_{in}(\theta)$ to be the set of variables that could affect the behaviour of some instruction θ in some way.

Definition 4. *For an instruction* θ*, define* $V_{in}(\theta)$ *by* $v \notin V_{in}(\theta)$ *iff for all* $s, s' \in S$*,* $s \equiv_{V-\{v\}} s'$ *implies* $s; \theta \equiv_{V_{out}(\theta)} s'; \theta$.

Example 2. *We can determine $V_{in}(\theta)$ for an instruction θ based on the Maude specification of Intel 64. By the definition of V_{in}, we know that if there exist stores $s, s' \in S$ such that $s \equiv_{V-\{v\}} s'$ and $s;\theta \not\equiv_{V_{out}(\theta)} s';\theta$ then $v \in V_{out}(\theta)$. Inspection of the Maude specification might result in the suspicion that* `v2` $\in V_{in}$(`mov v1, v2`)*. We can prove this by assuming that $s \equiv_{V-\{\texttt{v2}\}} s'$, which we can specify in Maude as follows:*

```
eq s[[v2]] = value1 .
eq s'[[v2]] = value2 .
ceq s[[V]] = s'[[V]]
    if V =/= v2 .
```

The first two equations say that `v2` *is different in stores* `s` *and* `s'`*, and the last equation says that every variable apart from* `v2` *has the same value in stores* `s` *and* `s'`*. Now, we can test using reductions in Maude whether the variables in V_{out}*(`mov v1, v2`) *are equal after executing* `mov v1, v2`*. Since V_{out}*(`mov v1, v2`) $= \{$`v1`, `ip`$\}$*, we can test these values using reductions:*

```
reduce s ; mov v1, v2 [[v1]] .
result Int: value1

reduce s' ; mov v1, v2 [[v1]] .
result Int: value2

reduce s ; mov v1, v2 [[ip]] .
result 1 + s'[[ip]]

reduce s' ; mov v1, v2 [[ip]] .
result 1 + s'[[ip]]
```

We can see that the value of `ip` *after executing* `mov v1, v2` *is the same in both stores, but the value of* `v1` *is different. Therefore, we know that* `v2` $\in V_{in}$(`mov v1, v2`)*.*

We can perform similar reductions to show that `ip` $\in V_{in}$(`mov v1,v2`) and that $v' \notin V_{in}$(`mov v1,v2`) for all v' not equal to `v2` or `ip`, i.e., V_{in}(`mov v1,v2`) $= \{$`v2`, `ip`$\}$. The complete Maude proof scripts can be seen in Appendix A.

Additionally, these functions extend naturally to sequences of instructions:

Definition 5. *For instruction sequences ψ_1 and ψ_2:*

$$\begin{aligned} V_{in}(\psi_1;\psi_2) &= V_{in}(\psi_1) \cup V_{in}(\psi_2), \text{ and} \\ V_{out}(\psi_1;\psi_2) &= V_{out}(\psi_1) \cup V_{out}(\psi_2) \ . \end{aligned}$$

In the following section the Maude specification of Intel 64 is used for dynamic analysis in order to prove equivalence/semi-equivalence of metamorphic computer virus code fragments, and in Section 2.7 we will explore an approach to static analysis of metamorphic computer viruses based on these definitions.

2.6 Dynamic Analysis

Using the formal specification of Intel 64 described in Section 2.4.2 it is possible to prove the equivalence or semi-equivalence of various allomorphs of metamorphic computer viruses using reductions in Maude, by using the Maude specification as an interpreter for dynamic analysis. The technique is used on allomorphic code fragments of two metamorphic computer viruses: Win95/Bistro and Win9x.Zmorph.A. The application of this technique to the detection of computer viruses is discussed in Section 2.8.

Before we begin, it is necessary to establish that if there is some sequence of instructions ψ for which $v \notin V_{out}(\psi)$, then the value of v is unchanged after executing ψ. We formalise this in

Proposition 1. *Let $\psi = \theta_1, \ldots, \theta_n$ be some sequence of instructions. Then for all stores $s \in S$, $s; \psi[\![v]\!] = s[\![v]\!]$ if $v \notin V_{out}(\psi)$.*

Proof. Proof is by induction. By Definition 5, we know that $v \notin V_{out}(\theta_i)$ for $0 \leq i \leq n$. By Definition 3, $s; \theta_1[\![v]\!] = s[\![v]\!]$ for all stores s. Let ψ_m be the subsequence of ψ consisting of the first m instructions in ψ, i.e., $\psi_m = \theta_1; \ldots; \theta_m$. Now, assume that $s; \psi_m[\![v]\!] = s[\![v]\!]$. Then by Definition 3, taking $s = \psi_m$ and $\theta = \theta_{m+1}$, we know that $s; \psi_{m+1}[\![v]\!] = s; \psi'[\![v]\!] = s[\![v]\!]$. Therefore $s; \psi[\![v]\!] = s[\![v]\!]$, as desired. □

2.6.1 Example 1: Win95/Bistro

Win95/Bistro applies equivalent sequence replacement to generate syntactic variants. Figure 2.1 shows two allomorphic fragments from Win95/Bistro.

The fragments have been divided up into three blocks each. The first two blocks consist of instruction sequences which alter the state of the stack, the `ebp` register and the instruction pointer (`ip`). We can analyse the effects on these variables using a Maude reduction. First we define two instruction sequences `a` and `b`, one for each block:

```
ops a b : -> InstructionSequence .
```

Next we define the instruction sequences corresponding to `a` and `b` (this is a shorthand that allows more concise use of the instruction sequences):

```
push ebp                          push ebp
mov ebp, esp                      push esp
                                  pop ebp

mov esi, dword ptr [ebp + 08]     mov esi, dword ptr [ebp + 08]
test esi, esi                     or esi, esi
je 401045                         je 401045

mov edi, dword ptr [ebp + 0c]     mov edi, dword ptr [ebp + 0c]
or edi, edi                       test edi, edi
je 401045                         je 401045
```

Figure 2.1: Allomorphic fragments of Win95/Bistro. [140]

```
eq a = push ebp ; mov ebp,esp .
eq b = push ebp ; push esp ; pop ebp .
```

Now, using the semantics of Intel 64 as specified in Maude, we can use a reduction to calculate the effects of any instruction sequence on any variable. We can also use the `_is_` operation to prove that two instruction sequences have the same effect on the same variable, and are therefore equivalent with respect to that variable.

Proposition 2. *Instruction sequences* `a` *and* `b` *are equivalent with respect to every variable apart from the instruction pointer, i.e.,* $\texttt{a} \equiv_W \texttt{b}$ *where* $W = V - \{\texttt{ip}\}$.

Proof. Since $V_{out}(\texttt{a}) = V_{out}(\texttt{b}) = \{\texttt{stack}, \texttt{ebp}\}$ we need only prove equivalence with respect to $\{\texttt{stack}, \texttt{ebp}\}$ and non-equivalence with respect to `ip`, because Proposition 1 states that all other variables (i.e., those outside $V_{out}(\texttt{a})$) will be unchanged.

```
Maude> reduce s ; a [[stack]] is s ; b [[stack]] .
result Bool: true
Maude> reduce s ; a [[ebp]] is s ; b [[ebp]] .
result Bool: true
Maude> reduce s ; a [[ip]] is s ; b [[ip]] .
result Bool: false
```

Therefore, `a` and `b` are equivalent with respect to every variable except the instruction pointer.

□

Next we can tackle the second pair of allomorphic fragments. This time we define a constant, `dword1`, to stand for the value of `dword ptr [ebp + 08]`, which is the same in both fragments.

```
op dword1 : -> EInt .
```

We define `c` and `d` in a similar way to last time:

```
ops c d : -> InstructionSequence .
eq c = mov esi, dword1 ; test esi, esi .
eq d = mov esi, dword1 ; or esi, esi .
```

The `test` instruction performs a Boolean-and operation on its operands, and sets the value of three flags (`zf`, `sf` and `pf`) in the EFLAGS register according to the result, and sets the value of two other flags (`cf` and `of`) in EFLAGS to zero (no other memory locations are updated) [74]. `or` performs a Boolean-or operation on its operands, and sets the value of three flags (`zf`, `sf` and `pf`) in the EFLAGS register according to the result, and sets the value of two other flags (`cf` and `of`) in EFLAGS to zero (also, the variable in the source operand is set to the result of the Boolean-or) [74]. Clearly, a Boolean-and is not equivalent to a Boolean-or, however these two instructions are equivalent if the source and destination operands in both instructions are the same variable. The Win95/Bistro virus uses this fact to generate allomorphs. We express this truth, the idempotent law of Boolean-and and Boolean-or, using two equations:

```
eq I | I = I .
eq I & I = I .
```

Proposition 3. `c` *is equivalent to* `d`*, i.e.,* $\texttt{c} \equiv \texttt{d}$.

Proof. Proof is with a reduction. Since

$$V_{out}(\texttt{c}) = V_{out}(\texttt{d}) = \{\texttt{esi}, \texttt{ip}, \texttt{zf}, \texttt{sf}, \texttt{pf}, \texttt{cf}, \texttt{of}\}$$

we need only prove equivalence with respect to these variables because Proposition 1 states that all other variables (i.e., those outside $V_{out}(\texttt{c})$) will be unchanged.

```
Maude> reduce s ; c [[esi]] is s ; d [[esi]] .
result Bool: true
Maude> reduce s ; c [[ip]] is s ; d [[ip]] .
result Bool: true
Maude> reduce s ; c [[zf]] is s ; d [[zf]] .
result Bool: true
Maude> reduce s ; c [[pf]] is s ; d [[pf]] .
result Bool: true
Maude> reduce s ; c [[sf]] is s ; d [[sf]] .
```

```
result Bool: true
Maude> reduce s ; c [[cf]] is s ; d [[cf]] .
result Bool: true
Maude> reduce s ; c [[of]] is s ; d [[of]] .
result Bool: true
```

Therefore, `c` is equivalent to `d`. □

The third pair of code fragments can be dealt with in a similar way to the second, as the same instructions are used.

We define another constant, `dword2`, to stand for the value of `[ebp + 0c]`, which is the same in both fragments.

```
op dword2 : -> EInt .
```

We define `e` and `f` in a similar way to `c` and `d`:

```
ops e f : -> InstructionSequence .
eq e = mov edi, dword2 ; or edi, edi .
eq f = mov edi, dword2 ; test edi, edi .
```

Proposition 4. `e` *is equivalent to* `f`*, i.e.,* $\mathtt{e} \equiv \mathtt{f}$.

Proof. Proof is with a reduction. Since

$$V_{out}(\mathtt{e}) = V_{out}(\mathtt{f}) = \{\mathtt{esi}, \mathtt{ip}, \mathtt{zf}, \mathtt{sf}, \mathtt{pf}, \mathtt{cf}, \mathtt{of}\}$$

we need only prove equivalence with respect to these variables because Proposition 1 states that all other variables (i.e., those outside $V_{out}(\mathtt{e})$) will be unchanged.

```
Maude> reduce s ; e [[esi]] is s ; f [[esi]] .
result Bool: true
Maude> reduce s ; e [[ip]] is s ; f [[ip]] .
result Bool: true
Maude> reduce s ; e [[zf]] is s ; f [[zf]] .
result Bool: true
Maude> reduce s ; e [[pf]] is s ; f [[pf]] .
result Bool: true
Maude> reduce s ; e [[sf]] is s ; f [[sf]] .
result Bool: true
Maude> reduce s ; e [[cf]] is s ; f [[cf]] .
```

```
mov edi, 2580774443        mov ebx, 535699961
mov ebx, 467750807         mov edx, 1490897411
sub ebx, 1745609157        xor ebx, 2402657826
sub edi, 150468176         mov ecx, 3802877865
xor ebx, 875205167         xor edx, 3743593982
push edi                   add ecx, 2386458904
xor edi, 3761393434        push ebx
push ebx                   push edx
push edi                   push ecx
```

Figure 2.2: Allomorphic fragments of Win9x.Zmorph.A.

```
result Bool: true
Maude> reduce s ; e [[of]] is s ; f [[of]] .
result Bool: true
```

Therefore, `e` is equivalent to `f`.

□

The proof scripts in Maude for Propositions 2–4 can be found in Appendix A.

2.6.2 Example 2: Win9x.Zmorph.A

Intel 64 code that was found after the disassembly of two Win9x.Zmorph.A allomorphs can be seen in Figure 2.2. It is known that this virus decrypts itself onto the stack from hardcoded numbers [80]. As both allomorphs were retrieved from the entry points of two executables infected with Zmorph, we might expect that the code fragments in Figure 2.2 to be equivalent with respect to the stack. We will now prove that this is the case.

In a similar way to the last proposition, we assign the two instruction sequences to `g` and `h` respectively.

Proposition 5. `g` *and* `h` *are equivalent with respect to the stack and instruction pointer, i.e.,* $\texttt{g} \equiv_W \texttt{h}$ *where* $W = \{\texttt{stack}, \texttt{ip}\}$.

Proof. We prove this by performing reductions to determine that the values of the stack and the instruction pointer are equal after executing `g` and `h`:

```
Maude> reduce s ; g [[stack]] is s ; h [[stack]] .
result Bool: true
Maude> reduce s ; g [[ip]] is s ; h [[ip]] .
result Bool: true
```

Therefore $\mathtt{g} \equiv_W \mathtt{h}$ where $W = \{\mathtt{stack}, \mathtt{ip}\}$. □

We can check the resulting state of the stack by performing an additional reduction:

```
Maude> reduce s ; g [[stack]] .
result Stack: 1894369473 next 2281701373 next 2430306267 next
              s[[stack]]
```

The original state of the stack is denoted by `s[[stack]]`, and the `_next_` operator delimits individual values placed on the stack.

Therefore, the two allomorphic fragments are equivalent (with respect to the stack) to the following Intel 64 instruction sequence:

```
push 2430306267 ; push 2281701373 ; push 1894369473
```

2.7 Static Analysis

In the previous section we showed how the formal definition in Maude of the syntax and semantics of a subset of Intel 64 assembly language could be used for dynamic analysis of metamorphic computer viruses. In this section we show how the formal definitions of behavioural equivalence and semi-equivalence of programs given in Section 2.5 can be used to prove that semi-equivalence can be extended to equivalence under certain circumstances. We will show how this proof can be used for static analysis of metamorphic computer viruses.

2.7.1 Equivalence in Context

We will now prove that for certain stores s_1, s_2 and instruction sequences ψ, if $s_1 \equiv_W s_2$ then $s_1; \psi \equiv s_2; \psi$. We say that s_1 and s_2 are *equivalent in context of* ψ.

We begin by establishing that if two stores s_1 and s_2 are semi-equivalent with respect to $V_{in}(\theta)$ for some instruction θ, then $s_1; \theta$ and $s_2; \theta$ are equivalent with respect to $V_{out}(\theta)$.

Lemma 1. *For all instructions θ and for all stores s_1, s_2:*

$$s_1 \equiv_{V_{in}(\theta)} s_2 \quad \textit{implies} \quad s_1; \theta \equiv_{V_{out}(\theta)} s_2; \theta \ .$$

Proof. Let $x_1, \ldots, x_n$ be an enumeration of $V \setminus V_{in}(\theta)$. Let $s_{1,1}$ be some state identical to s_1, except

$$s_{1,1}[\![x_1]\!] = s_2[\![x_1]\!] \ .$$

Likewise, let $s_{1,i+1}$ be some state identical to s_i except

$$s_{1,i+1}[\![x_{i+1}]\!] = s_2[\![x_{i+1}]\!] \ .$$

By Definition 4,

$$s_1;\theta \equiv_{V_{out}(\theta)} s_{1,1};\theta \equiv_{V_{out}(\theta)} s_{1,2};\theta \equiv_{V_{out}(\theta)} \ldots \equiv_{V_{out}(\theta)} s_{1,n};\theta = s_2;\theta \ ,$$

and therefore $s_1;\theta \equiv_{V_{out}(\theta)} s_2;\theta$, as desired. □

Next, we show that if there are two stores that are semi-equivalent with respect to W, and the set $V_{in}(\theta)$ for some instruction θ is covered by W, then the resulting stores after executing θ are equivalent with respect to $W \cup V_{out}(\theta)$.

Lemma 2. *If $s_1 \equiv_W s_2$ and $V_{in}(\theta) \subseteq W$ then:*

$$s_1;\theta \equiv_{W \cup V_{out}(\theta)} s_2;\theta \ .$$

Proof. Assume $s_1 \equiv_W s_2$. By the previous lemma, we know that $s_1;\theta \equiv_{V_{out}(\theta)} s_2;\theta$, so we need only consider variables in W and not in $V_{out}(\theta)$. For any $w \notin V_{out}(\theta)$, we have $s_1;\theta[\![w]\!] = s_1[\![w]\!]$ and $s_2;\theta[\![w]\!] = s_2[\![w]\!]$, by Definition 3. If $w \in W$, then $s_1[\![w]\!] = s_2[\![w]\!]$ by assumption that $s_1 \equiv_W s_2$, so $s_1;\theta[\![w]\!] = s_2;\theta[\![w]\!]$. Therefore, $s_1;\theta \equiv_{W \cup V_{out}(\theta)} s_2;\theta$, as desired. □

Now we can incrementally chain together sets of variables into equivalences for instruction sequences with our main

Theorem 1. *Let ψ be an instruction sequence such that $\psi = \theta_1;\theta_2;\ldots;\theta_m$, where $\theta_{1 \le i \le m}$ are instructions. If $s_1 \equiv_W s_2$ and for all j with $1 \le j \le m$*

$$V_{in}(\theta_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(\theta_i) \tag{2.3}$$

then $s_1;\psi \equiv_{W \cup V_{out}(\psi)} s_2;\psi$.

Proof. By induction on m. The base case, where $m = 1$, is shown in Lemma 2. For the induction step, assume $s_1 \equiv_W s_2$ and for $1 \le j \le m$

$$V_{in}(\theta_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(\theta_i) \tag{2.4}$$

so that, by the induction hypothesis, $s_1;\psi' \equiv_{W \cup V_{out}(\psi')} s_2;\psi'$, where $\psi' = \theta_1;\theta_2;\ldots;\theta_{m-1}$. Now apply Lemma 2 again (taking the s_1 of that lemma to be $s_1;\psi'$, s_2 to be $s_2;\psi'$,

noting that, by (2.4) $V_{in}(\theta_m) \subseteq W \cup V_{out}(\psi'))$, and since $V_{out}(\psi) = V_{out}(\psi') \cup V_{out}(\theta_m)$, this gives $s_1; \psi \equiv_{W \cup V_{out}(\psi)} s_2; \psi$, as desired. □

It is possible to recover equivalence of instruction sequences from semi-equivalence in some cases. If $s_1 \equiv_W s_2$, then s_1 and s_2 have different values for variables in $V - W$ (which we henceforth write as $\overline{W}$); but if all variables in $\overline{W}$ are overwritten in the same way by some instruction sequence ψ, despite the differences in $\overline{W}$, then s_1 is equivalent to s_2 in the context of ψ, as stated in

Corollary 1 (Equivalence in Context)**.** *If $p_1 \equiv_W p_2$ and $p_1; \psi \equiv_{W \cup V_{out}(\psi)} p_2; \psi$ for instruction sequences p_1, p_2, ψ and $\overline{W} \subseteq V_{out}(\psi)$ then $p_1; \psi \equiv p_2; \psi$.*

Proof. If $\overline{W} \subseteq V_{out}(\psi)$ then $p_1; \psi \equiv_{W \cup \overline{W}} p_2; \psi$. Since $W \cup \overline{W} = V$ it follows that $p_1; \psi \equiv p_2; \psi$. □

2.7.2 Examples Using Win9x.Zmorph.A

In Section 2.6 we proved that two allomorphic instruction sequences from the Win9x.-Zmorph.A metamorphic computer virus, called `g` and `h`, were equivalent with respect to the stack and the instruction pointer. In this subsection we present two examples of the equivalence in context as applied to `g` and `h`, based on instruction sequences ψ and ψ'. To our knowledge, neither of these instruction sequences appear in the source code of Zmorph, but are constructions designed to illustrate the practical application of equivalence in context.

In the first example we will show that `g` and `h` are equivalent in context of another instruction sequence ψ, by applying the result from Corollary 1.

Example 3. *By Proposition 1 we know that $s; \mathtt{g}[\![v]\!] = s[\![v]\!]$ for all $v \notin V_{out}(\mathtt{g})$, and $s; \mathtt{h}[\![v]\!] = s[\![v]\!]$ for all $v \notin V_{out}(\mathtt{h})$. Therefore $s; \mathtt{g}[\![v]\!] = s; \mathtt{h}[\![v]\!]$ for all $v \notin V_{out}(\mathtt{g}) \cup V_{out}(\mathtt{h})$, and so $\mathtt{g} \equiv_W \mathtt{h}$ where $V \setminus W = \{edi, ebx, ecx, edx\}$. Given that a metamorphic computer virus exhibits the same behaviour, but only changes its syntax, it is reasonable to assume that any use of semi-equivalent code may not result in a difference in behaviour, i.e., the non-equivalent variables will not be used to determine the effects of the rest of the program. We will show how an instruction sequence ψ executed immediately after `g` and `h` results in an equivalent store, which allows the metamorphic computer virus to freely swap `g` and `h` as long as ψ executes next.*

Let $\psi =$ `mov edi, 0 ; mov ebx, 0; mov ecx, 0 ; mov edx, 0`*. In order to apply Theorem 1, we must first check the values of $V_{in}(\theta)$ and $V_{out}(\theta)$ for all instructions θ in ψ:*

$$
\begin{array}{lcllcl}
V_{in}(\texttt{mov edi, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov edi, 0}) & = & \{\texttt{edi}, \texttt{ip}\} \\
V_{in}(\texttt{mov ebx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov ebx, 0}) & = & \{\texttt{ebx}, \texttt{ip}\} \\
V_{in}(\texttt{mov ecx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov ecx, 0}) & = & \{\texttt{ecx}, \texttt{ip}\} \\
V_{in}(\texttt{mov edx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov edx, 0}) & = & \{\texttt{edx}, \texttt{ip}\}
\end{array}
$$

The following therefore hold:

$$
\begin{array}{lcl}
V_{in}(\texttt{mov edi, 0}) & \subseteq & W \\
V_{in}(\texttt{mov ebx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \\
V_{in}(\texttt{mov ecx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \\
V_{in}(\texttt{mov edx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \cup V_{out}(\texttt{mov ecx, 0})
\end{array}
$$

Therefore by Theorem 1, $\texttt{g}; \psi \equiv_{W \cup V_{out}(\psi)} \texttt{h}; \psi$*, and since* $\overline{W} \subseteq V_{out}(\psi)$*, we know by Corollary 1 that* $\texttt{g}; \psi \equiv \texttt{h}; \psi$*.*

Indeed, we can check using our Maude specification of Intel 64 that this is the case. We introduce an instruction list in Maude which modifies a store `S` as follows:

```
op psi : -> InstructionSequence .
eq psi = mov edi, 0 ; mov ebx, 0 ; mov ecx, 0 ; mov edx, 0 .
```

The instruction list `psi` is therefore the same as instruction list ψ. We can then check using a reduction that executing `psi` in the stores resulting from the execution of `g` and `h` results in an equivalent store:

```
reduce s ; g ; psi [[stack]] is s ; h ; psi [[stack]] .
result Bool: true
reduce s ; g ; psi [[ip]] is s ; h ; psi [[ip]] .
result Bool: true
reduce s ; g ; psi [[edi]] is s ; h ; psi [[edi]] .
result Bool: true
reduce s ; g ; psi [[ebx]] is s ; h ; psi [[ebx]] .
result Bool: true
reduce s ; g ; psi [[ecx]] is s ; h ; psi [[ecx]] .
result Bool: true
reduce s ; g ; psi [[edx]] is s ; h ; psi [[edx]] .
result Bool: true
```

Since we know that $V_{out}(g) \cup V_{out}(h) \cup V_{out}(\psi) = \{\texttt{stack}, \texttt{ip}, \texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$, then by Proposition 1 we can take these Maude reductions as a second proof that $\texttt{g}; \psi \equiv \texttt{h}; \psi$.

In the example above we showed that by overwriting the non-equivalent variables from the semi-equivalent stores `g` and `h` in the instruction sequence ψ, that `g` and `h` are equivalent in context of ψ. In the following example we will show that equivalence can also be demonstrated where an instruction sequence ψ' contains instructions which overwrite the non-equivalent variables, as long as the instructions in ψ' are not dependent on the non-equivalent variables.

Example 4. *Once again we know that* $g \equiv_W h$ *where* $V \setminus W = \{edi, ebx, ecx, edx\}$. *Let* $\psi' =$ `pop edi ; pop ebx ; pop ecx ; mov ecx, edx`.

Once again we must check the values of $V_{in}(\theta')$ *and* $V_{out}(\theta')$ *for all instructions* θ' *in* ψ' *before we can apply Theorem 1:*

$$\begin{aligned} V_{in}(\theta'_1) &= \{ip, stack\} & V_{out}(\theta'_1) &= \{edi, ip\} \\ V_{in}(\theta'_2) &= \{ip, stack\} & V_{out}(\theta'_2) &= \{ebx, ip\} \\ V_{in}(\theta'_3) &= \{ip, stack\} & V_{out}(\theta'_3) &= \{ecx, ip\} \\ V_{in}(\theta'_4) &= \{ip, ecx\} & V_{out}(\theta'_4) &= \{edx, ip\} \end{aligned}$$

The following therefore hold:

$$\begin{aligned} V_{in}(\theta'_1) &\subseteq W \\ V_{in}(\theta'_2) &\subseteq W \cup V_{out}(\theta'_1) \\ V_{in}(\theta'_3) &\subseteq W \cup V_{out}(\theta'_1) \cup V_{out}(\theta'_2) \\ V_{in}(\theta'_4) &\subseteq W \cup V_{out}(\theta'_1) \cup V_{out}(\theta'_2) \cup V_{out}(\theta'_3) \end{aligned}$$

Therefore by Theorem 1, $\texttt{g}; \psi' \equiv_{W \cup V_{out}(\psi')} \texttt{h}; \psi'$, *and since* $\overline{W} \subseteq V_{out}(\psi')$, *we know by Corollary 1 that* $g; \psi' \equiv h; \psi'$.

Again, it is possible to double-check that $g; \psi' \equiv h; \psi'$ using the Maude specification of Intel 64. We create an instruction sequence `psi'` $= \psi'$ as follows:

```
op psi' : -> InstructionSequence .
eq psi' = pop edi ; pop ebx ; pop ecx ; mov edx, ecx .
```

Then, we can check the stores are equal be performing a reduction for each variable in $V_{out}(g) \cup V_{out}(h) \cup V_{out}(\psi') = \{\texttt{stack}, \texttt{ip}, \texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$:

```
reduce s ; g ; psi' [[stack]] is s ; h ; psi' [[stack]] .
```

```
result Bool: true
reduce s ; g ; psi' [[ip]] is s ; h ; psi' [[ip]] .
result Bool: true
reduce s ; g ; psi' [[edi]] is s ; h ; psi' [[edi]] .
result Bool: true
reduce s ; g ; psi' [[ebx]] is s ; h ; psi' [[ebx]] .
result Bool: true
reduce s ; g ; psi' [[ecx]] is s ; h ; psi' [[ecx]] .
result Bool: true
reduce s ; g ; psi' [[edx]] is s ; h ; psi' [[edx]] .
result Bool: true
```

2.8 Applications to Detection of Metamorphic Viruses

In the previous sections we have shown how the formal specification in Maude of the Intel 64 assembly programming language enables static and dynamic analysis to prove equivalence and semi-equivalence of code. We have shown how metamorphic computer viruses use equivalent and semi-equivalent code in order to avoid detection by signature scanning. Therefore, given the techniques for code analysis described above, it seems reasonable that static and dynamic analysis based on the formal specification of Intel 64 should give ways to detect metamorphic computer viruses by proving the equivalence of different generations of the same virus to some virus signature, thus enabling detection of metamorphic computer viruses by a signature-based approach.

Implementation of a industrial standard tool for metamorphic computer virus detection is beyond the scope of this thesis, but some suggestions for possible applications of the techniques for proving equivalence of metamorphic code are as follows.

2.8.1 Dynamic Analysis

2.8.1.1 Signature Equivalence

The most obvious application for detection is based on the techniques used in Section 2.6 to prove by dynamic analysis the equivalence of code fragments. Suppose that a signature σ is stored in a disassembled form, and that there is a fragment of suspect code c within an executable file. Then, the effects of c and σ on a generalised store could be discovered by performing Maude reductions as in Section 2.6.1. The resulting stores could be compared, and if equal, would prove that $c \equiv \sigma$. Computer virus signatures must be *sufficiently discriminating* and *non-incriminating*, meaning

that they must identify a particular virus reliably without falsely incriminating code from a different virus or non-virus (ch.5, [43]). If a suspect code block was proven to have equivalent behaviour to a signature, this would result in identification to the same degree of accuracy as the original signature. (Since a signature uses a syntactic representation of the semantics of a code fragment to identify a viral behavioural trait, any equivalent signature must therefore identify the same trait.) If the code block is only semi-equivalent, then the accuracy of detection could be reduced. However if equivalence-in-context could be proven then accuracy would again be to the same degree as the original signature.

2.8.1.2 Signature Semi-Equivalence

It might be the case that a given metamorphic computer virus is known to write certain values onto the stack, and therefore the state of the stack at a certain point in the execution of the metamorphic virus could be a possible means of detection. In Section 2.6.2, two variants of the Win9x.Zmorph.A metamorphic computer virus were shown to be equivalent with respect to the stack, meaning that the state of the stack was affected in the same way by both generations of the virus. Therefore, the same technique could be used for detection. In this case, equivalence need not be proven, as the detection method relies on equivalence with respect to a subset of variables, i.e., semi-equivalence.

2.8.2 Static Analysis

2.8.2.1 Formally-Verified Equivalent Code Libraries

One important result in the field of algebraic specification is the Theorem of Constants (p.38, [58]). Informally, the theorem states that any nullary operator (i.e., constant) used in a reduction within an algebraic specification system such as Maude, can be used as a variable in that reduction. This holds because the definition of variables within Maude is that they are actually constants within a supersignature, i.e., a variable in a Maude module is a constant within another module that encompasses it. This lets us use constants in place of variables, e.g., for the reductions used in Proposition 2 we use a constant `s` to denote any store s.

This means that the proofs of equivalence and semi-equivalence of the code fragments in Propositions 2–4 still hold if we swap the program variable names for other program variable names of the same sort (e.g., we don't interchange stack variables and

"ordinary" variables such as the `eax` register). In Proposition 2 we show that

$$\texttt{push ebp ; mov ebp,esp} \quad \equiv_W \quad \texttt{push ebp ; push esp ; pop ebp} \tag{2.5}$$

where $W = V - \{ip\}$.

By the Theorem of Constants we can replace `ebp` with `eax`, and `esp` with `edx`, for example, and the statement of semi-equivalence still holds. Therefore, we might rephrase the above with a more standard mathematical notation, e.g.:

$$\texttt{push } x \texttt{ ; mov } x\texttt{,}y \quad \equiv_W \quad \texttt{push } x \texttt{ ; push } y \texttt{ ; pop } x \tag{2.6}$$

Therefore, if we know that metamorphic computer viruses might use a set of equations similar to Equation 2.6, then we may wish to build up a library of equivalent instruction lists based on those equations. In doing so we could decide, for instance, that all instances of the left-hand side of Equation 2.6 should be "replaced by" the right-hand side. If there was a metamorphic computer virus that exhibited only this kind of metamorphism, then we would have effectively created a normal form of the virus that would enable detection by straightforward signature scanning. Of course, this is example is intentionally kept simple, and many metamorphic computer viruses will employ code mutation techniques which are far more complex, but the general idea of code libraries which we are formally verified using a formal specification language such as Maude is perhaps useful.

2.8.2.2 Equivalence in Context

As shown in Sections 2.6.1 and 2.6.2, metamorphic computer viruses can use semi-equivalent code replacement in order to produce syntactic variants in order to evade signature-based detection. The obvious advantage of this stratagem is that restricting metamorphism to code sequences that are equivalent limits the number of syntactic variants. An obvious example is that metamorphic computer viruses may wish to use code that treats all variables equivalently except the instruction pointer, i.e., equivalent code of differing length that is semi-equivalent with respect to every variable except the instruction pointer. Clearly, this will not pose a problem for the metamorphic computer virus as long as there is no part of its program that is dependent on the value of the instruction pointer at a given point after the mutated code.

It is likely, therefore, that a code segment c of a suspect executable will be semi-equivalent to some signature σ of a metamorphic computer virus. If it were possible to prove equivalence-in-context, i.e., that $c;\psi \equiv \sigma;\psi$, where ψ is some code appearing immediately after c in the suspect executable, then it would be known that σ was a

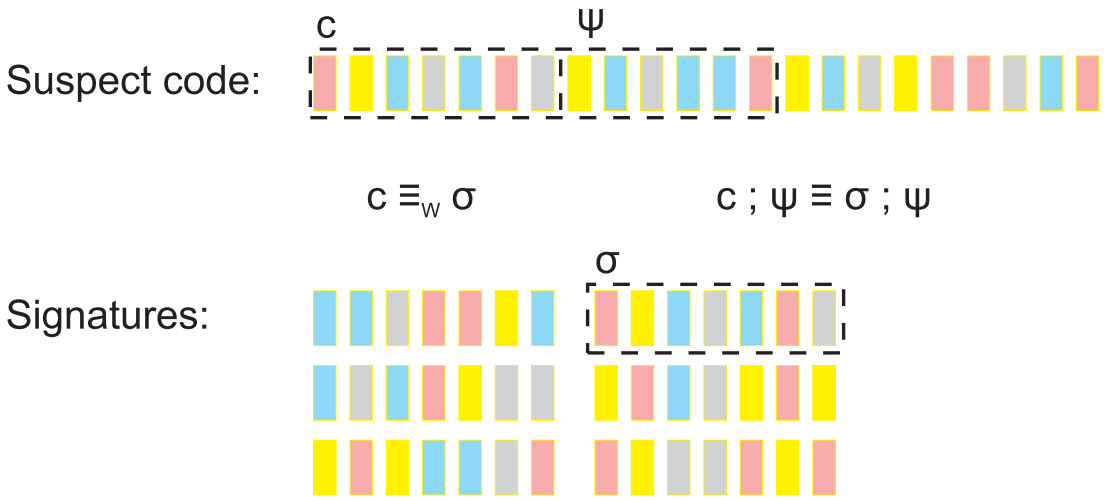


Figure 2.3: Signature-based detection of a metamorphic computer virus, by application of equivalence-in-context. Instruction sequences c and σ are semi-equivalent with respect to W. Applying the result in Corollary 1 to c, σ and ψ reveals that in fact $c;\psi \equiv \sigma;\psi$ and therefore c has been identified as equivalent to signature σ, resulting in detection of the virus.

successful match to c and detection of the virus would be achieved. (See Figure 2.3 for an illustrated example.)

2.8.3 Combination With Other Approaches

Another potential application of the methods for computer virus detection described above, is to combine them with other means of metamorphic computer virus detection. For instance, the formally-verified equivalent code library described above may not always result in reduction of every generation of a metamorphic computer virus to a normal form. However, the overall syntactic variance of the set of all generations may be significantly reduced, so that another technique may be used to enable detection. For instance, the neural network-based approach of Yoo et al (described below in Section 2.9.1.5) relies on the identification of similar code structures, and therefore may be assisted by an equivalent code library.

2.9 Summary

As we said in Section 1.1.4, detection of computer viruses is, in general, an undecidable problem. Metamorphic computer viruses are particularly difficult; there are known cases where detection is computationally intractable [136, 17, 18, 169, 72], or even undecidable [26, 40, 47]. Not all metamorphic computer viruses are so difficult to

detect, and therefore research into practical ways to detect metamorphic computer viruses is essential.

In this chapter we showed how a formal algebraic specification of an assembly programming language and a formal notion of equivalence can be used to detect metamorphic computer viruses through static and dynamic analysis. In Section 2.2 we described the difficulties of detecting metamorphic computer viruses using existing detection techniques, and gave an overview of the various forms of code metamorphism employed by metamorphic computer viruses. Then, in Section 2.3 we gave an overview of the Maude formal notation, and described its applicability to the specification of the syntax and semantics of programming languages. In Section 2.4 we showed how a formal specification of the syntax and semantics of a subset of the Intel 64 assembly language can be given using Maude, and how Maude's built-in rewriting engine can turn a static specification into an executable specification in which we gain an interpreter, essentially for free.

In Section 2.5 a notion of program equivalence was defined as the equality of stores after the execution of two programs, and program semi-equivalence was defined as the case where only a subset of variables in the resulting stores are equal. In Section 2.6 we demonstrated proofs of the equivalence and semi-equivalence of real-life metamorphic computer virus code using the formal specification of Intel 64, proving its use as a tool for dynamic analysis of metamorphic computer viruses in particular, and assembly language programs in general. In Section 2.7 we proved that under certain circumstances semi-equivalence can be extended to equivalence, and we showed how this could be used for static analysis in the detection of metamorphic computer viruses. Finally, in Section 2.8 we gave some directions for the application of our approach to the practical problem of metamorphic computer virus detection.

Now we will critically assess the novelty and contribution of the research described in this chapter through a comparison with related work in the field.

2.9.1 Related Work

There are currently many avenues of research into the detection of metamorphic computer viruses, both academic and industrial. The aim of this section will be to compare the work presented in this chapter with other approaches to metamorphic computer virus detection given in the literature. It is likely that much research into this area is conducted by commercial anti-virus companies; however, much of this is not published and therefore an accurate comparison is not possible. There is some information in the literature concerning methods used by commercial anti-virus software for metamorphic computer virus detection, and this is covered in Section 2.9.1.6. However, given that

the most detailed information on comparable approaches is in the academic literature, a comparison between this work and the work presented in this chapter will be the focus for this section.

2.9.1.1 Control- and Data-Flow Analysis

Common assembly programming languages such as Intel 64 are in the imperative paradigm, that is, successive statements within the language describe successive updates to a store, which maps variable names to values. In assembly languages, loops are achieved through branching statements which update the value of the instruction pointer, effectively allowing for the repetition of statement lists within a program. Control-flow analysis consists of extracting the flow of control from a sequence of statements, itself dependent on the ordering of jump statements within the program. Since different generations of the same computer virus might use the same flow of control, control-flow analysis can be used to detect metamorphic computer viruses. Data-flow analysis is similar to control-flow analysis, except the flow of information through the program is analysed, rather than the flow of control. For example, an Intel 64 assembly language instruction that updates the value of register `eax` with the value of register `ebx`, is itself informationally-dependent on a previous instruction which set the value of `ebx`. Therefore, both control and data-flow analyses calculate the dependencies of instruction, and enables the construction of graphs which display this information.

Lakhotia & Mohammed describe an algorithm for imposing order on high-level language programs based on control- and data-flow analysis [106, 88]. First, a *program tree* is generated, showing the control-flow of a sequence of statements. Using data-flow analysis, the program tree is then partitioned into sets of statements that are re-orderable. Each statement in the program tree is assigned a number in a depth-first manner, with statements in re-orderable sets given unique numbers according to a specially-defined numbering system. A reconstructed program tree is then generated in which the statements are ordered as per the numbering system in the previous step.

The focus of this approach, therefore, is to reverse the effects of statement re-ordering, variable renaming (see Sections 2.2.1.4 and 2.2.1.2 respectively) and "expression reshaping" (a kind of metamorphism not seen in assembly languages and therefore not given in Section 2.2.1) using a static analysis-based approach. The above algorithm attempts to generate a normalised ("zero") form of statement lists mutated by these metamorphism types. The authors note that their algorithm is an heuristic, and therefore it will not always reduce programs to a normalised form, but empirical evidence given suggests that the reduction of syntactic variants to be significant.

Bruschi et al [19] describe a similar method for malware detection to the one

described by Lakhotia & Mohammed, which uses code normalisation for the following:

- To identify junk code.
- To simplify algebraic expressions.
- To remove the reliance on intermediary variables used in assignments.
- To remove dead paths by control flow analysis.
- To simplify the control flow of the program.

The above strategies are employed in order to reduce the number of syntactic variants of a single program behaviour, and therefore to aid in the detection of metamorphic computer viruses. The detection algorithm described has a two-stage process. First the *code normaliser* decodes the disassembled code into a form which allows control- and data-flow analysis, before performing a number of transformations in order to normalise the code. The output of the code normaliser is then passed to the *code comparator*, which works on the assumption that inter-procedural control-flow graphs are unchanged across metamorphic computer virus generations. Therefore, the problem of detection is reduced to the problem of deciding whether two sub-graphs are isomorphic. If the code comparator finds that the inter-procedural control-flow graph of a suspect executable is isomorphic to some signature inter-procedural control-flow graph of a metamorphic computer virus, then the suspect executable is identified as being infected by the virus. A test on 115 generations of the MetaPHOR computer virus revealed that the code detection method accurately identified all generations as having the same control-flow graph, therefore making detection possible.

Another approach is described by Bonfante et al [16]. In a similar way to Bruschi et al's approach, a control flow graph is extracted from a computer virus and is used as a signature for that malware. Therefore, detection is reduced to the problem of ckecking isomorphism between two graphs: one graph is the signature, the other is the suspect executable. Next, the authors tackle the problem of signature database management. Rather than store the graphs themselves in a database, the graphs are translated into canonical terms. A tree automaton can then be constructed which recognises all of the canonical term signatures. Then, detection consists of extracting a control flow graph from a suspect executable e, translating it into a canonical term e_t and executing the tree automaton with e_t as an input. The tree automaton will indicate whether or not e contains any of the malware control flow graphs. The efficiency of the tree automaton enables indication of infection of a suspect executable of size n in $O(n^2)$ time, or in $O(n)$ time with the use of some heuristics in the construction of the control flow graphs.

Finally, the authors describe experiments in which different lower bounds for the size of the control flow graph extracted from malware are tested. It is found that control flow graphs of 19 nodes or greater are sufficient to reduce false positives to less than 0.1%.

Another approach involving control flow graphs and call graphs was given by Bilar [11]. A *call graph* is a graph in which the nodes are functions from an executable program, and the edges indicate that one function calls another. A sequence of instructions that has no branching into its middle and ends in a branching instruction (e.g., `jmp`) is called a *basic block*. A set of 400 executables was used as a sample set, in which 280 were non-malware and 120 were malware. The *indegree* and *outdegree* of a function are the number of calls to and from the function respectively. The correlation between indegree and outdegree was calculated for the sample set, but no correlation was found for either malware or non-malware. Additionally, both the malware and non-malware sample groups were grouped into three groups with similar function counts in order to test for a correlation between instruction count in functions and complexity of the executable, but no correlation was found. Next, the author tested whether the indegree, outdegree and the basic block count followed a truncated power law distribution, i.e., whether the probability of the occurrence of a function with a given indegree, outdegree or basic block count decreases as a power law with respect to the size. After finding power law exponenents for each variable, the author found that the power law exponent for basic block size of the malware resulted in a steeper slope that for the non-malware, meaning that malware executables tend to have a lower number of basic blocks. Therefore this fact could be used as an heuristic to detect malware. The author theorised that malware executables tend to be simpler, have more limited functionality, are compiled differently and may be designed differently than non-malware, therefore resulting in a tendency towards lower block count[5].

2.9.1.2 Semantics Template Matching

Christodorescu et al describe an approach to metamorphic computer virus detection using a signature-matching approach, in which the signatures contain information regarding the semantics, as well as the syntax, of the metamorphic computer virus [28]. A formal semantics of assembly language instruction sequences is given based on the following:

- A *template* describes a sequence of instructions, together with set of variables and symbolic constants that appear in that instruction sequence.

[5]The author would like to thank Dr. Bilar for his comments on this review.

- An *execution context* for a template describes a pre-condition for the execution of the template, in which every symbolic constant from the template has a specified value.

- A *template state* consists of a function mapping variables to values, a value of the program counter (i.e., instruction pointer) and a function that assigns values to memory address.

The execution of the instruction sequence defined in the template is described naturally as a labelled transition system in which the states are the template states, the actions are instructions and the labelled transition relation describes states and their successive states after execution of an instruction. It is possible to define a signature for a metamorphic computer virus using a template. Detection then proceeds by matching the code within a suspect executable to one of a set of templates stored in a database. This is described formally using an algorithm.

The authors also present some interesting formal results. First they prove that the problem of template matching is undecidable in general. They also prove that the aforementioned detection algorithm is sound with respect to the template matching problem. In a later paper Preda et al [115] are able to prove the correctness of this approach with respect to instruction reordering, variable renaming and junk code insertion (see Section 2.2.1 for definitions).

Finally, the detection algorithm is tested for three different real-life worm examples: Netsky, B[e]agle and Sober. Templates are defined for a decryption loop and a mass email procedure common to worms found in the wild. Empirical analysis showed that the decryption loop was detected in all three worms, whereas the mass email procedure was found in only two of the worms. Therefore the approach is applicable to real-life detection scenarios.

2.9.1.3 Program Rewriting and Normalisation

Bruschi et al describe a normalisation procedure based on program rewriting [20, 21]. This approach to metamorphic computer virus detection differs from the approach by Bruschi et al described in Section 2.9.1.1, although it is also based on normalisation.

First, the assembly language instructions are translated into a "meta-representation" which describes (informally) the semantics of the instructions. For example, the instruction `pop eax` is meta-represented as rules "r10 = [r11]" and "r11 = r11 + 4", where r10 and r11 denote register variables. Then, the code is rewritten using optimisation techniques commonly used by compilers:

- Propagation: values assigned to intermediary variables are used directly instead of the intermediary variable. For example the meta-representation instruction sequence "r10 = 0 ; r11 = r10" can be simplified to "r10 = 0 ; r11 = 0" using propagation.
- Dead code elimination: removal of instructions whose results do not affect the behaviour of the program. "Dead code" is the same as the junk code discussed in Section 2.2.1.
- Algebraic simplification: algebraic laws of number theory can be applied to the meta-representation in order to simply expressions, e.g., $0 + x = x$ can be used to simplify any term matching the left-hand side by rewriting it to the term on the right-hand side.
- Control flow graph compression: false conditional and unconditional jumps are unravelled using control-flow analysis similar to that described in Section 2.9.1.1.

Normalisation will not always be enough to uniquely identify metamorphic computer viruses, so the authors combine their optimisations with a number of metrics designed to detect metamorphic computer viruses heuristically. For example, metrics can be the number of nodes in a control-flow graph, or the number of indirect jumps in a normalised code sample. The values of the n various metrics are stored as a point in a Euclidean n-space, and the difference between two code samples is then calculated as the Euclidean distance between the two points.

An empirical analysis of the efficacy of the approach was conducted using a sample of generations of the MetaPHOR metamorphic computer virus. It was found that after normalisation, the average Euclidean distance of the generations was significantly lower than before, and therefore the approach can reduce the amount of variation in syntax seen in MetaPHOR, and possibly other metamorphic computer viruses.

Another approach to rewriting-based normalisation was taken by Walenstein et al [150]. This time, rewriting of the assembly language programs was done directly, bypassing the need for a meta-representation. First the problem is framed as a term-rewriting problem; the rules by which the metamorphic code engine creates new generations of the virus are described as rewrite rules within a term-rewriting system. The authors consider the problem of constructing a normalisation procedure for an arbitrary metamorphic computer virus, which they call the *normaliser construction problem (NCP)*: given a non-confluent metamorphic engine term rewriting system M, construct a canonical (i.e., confluent and terminating) term rewriting system N called the normaliser such that all terms from M rewritten using N are equivalent semantically.

The method used is as follows. First, termination is ensured by using a *reordering procedure* to order the rewrite rules so that the term is shortened on each rewrite. Then, confluence is attempted by using the Knuth–Bendix completion procedure, although the authors note that the algorithm is not guaranteed to terminate with success. For the case where the algorithm does not terminate, the authors give some possible directions for producing approximate solutions to the NCP.

As an example, the authors create an NCP for the Win32.Evol metamorphic computer virus. After extracting the metamorphic engine term rewriting system, an NCP called P_0 was created using the reordering procedure, but to ensure termination confluence was not assured. However, it was still useful as it resulted in a reduction of the number of synactic variants of the virus across generations. The authors also created a canonical term rewriting system, P_1, which was able to reduce all generations of Evol to the same normalised form. The application of the technique to Evol proves that the approach is effective for detection of metamorphic computer viruses.

2.9.1.4 Metamorphic Engine Analysis

Choucane & Lakhotia describe an approach to metamorphic computer virus detection based on the assumption that metamorphic computer viruses often use the same metamorphism engine, and that by assigning an *engine signature* it ought to be possible to assign a probability that a suspect executable is an output of that engine [27].

The rewrite rule set of the metamorphic code engine is obtained in a similar to manner to Walenstein et al's approach described in the previous section. The rewritten forms of the rules (the "right-hand sides") are seen as "clues" for detection of that particular engine. Since the rules are applied by metamorphic computer viruses probabilistically according to the specification of the engine, each rule is accompanied by a *rule application probability*. A scoring function takes an instruction sequence and set of rules and their application probabilities, and returns a score proportional to the evidence linking the set of rules to the instruction sequence, i.e., the higher the score, the more likely it is that the instruction sequence is a product of that particular engine.

Experiments were performed in order to demonstrate that the scoring function increases as the number of the generation of the metamorphic computer virus increases, i.e., the offspring of the virus scores more highly than its ancestors. This is the expected result, as it is likely that more of the code will be re-written according to the rules over successive generations, and therefore the offspring's code will contain more clues. The authors do not provide a baseline score for uninfected executables, so it is difficult to determine whether the approach would be useful for metamorphic computer virus detection, however, given a known infected executable the method could

clearly aid identification of which metamorphic computer virus engine was responsible for producing the code, assuming that its rules have already been included in the clue database.

2.9.1.5 Neural Network Approaches

Yoo & Ultes Nitsche [167, 168] present a unique approach to metamorphic computer virus detection, which involves training a type of artificial neural network known as a self-ordering map (SOM). SOMs exhibit unsupervised learning, and therefore can be trained to recognise computer viruses within executable files. The approach is completely non-signature based, and once the SOM has been trained it can be applied to any suspect executable. For example, the authors apply train a SOM to detect the Win32.Apparition metamorphic computer virus, which can then be detected by the SOM. The other experiments conducted by the authors are for polymorphic, rather than metamorphic computer viruses, so it is difficult to assess whether this approach could be generalised for other kinds of metamorphic computer viruses. An additional problem noted by the authors is the sensitivity of the approach to operator skill, as the parameters of the SOM must be set manually, and the success of the detection method is highly dependent on their particular values.

2.9.1.6 Industrial Approaches

Commercial anti-virus software has the ability to detect certain types of metamorphic computer virus. As the technical details of commercial anti-virus scanners tend to be kept secret, it is difficult to know the exact state of the art. However, recent publications by Ször [139, 140] give a few abstract technical details:

- *Geometric detection*: alterations to the file structure indicate the presence of some (metamorphic) computer viruses. For example, the Win95/Zmist metamorphic computer virus increases the virtual size of the data section of the infected executable by 32 KB, but the physical size remains unchanged. This fact allows Zmist to be detected by anti-virus software.

- *Detection by Disassembly*: once a metamorphic computer virus is disassembled, it can be easier to detect, as there are sometimes instructions which are common to computer viruses, such as `cmp ax, "ZM"` for checking whether a file is executable. The presence of this instruction can be used a heuristic for detection.

- *Detection by Emulation*: metamorphic computer viruses can be detected through dynamic analysis by emulating the suspect code. The state of the machine at a

given point may be an heuristic for detecting a particular computer virus.

2.9.2 Comparisons with Related Work

Based on the overview of the various other approaches to metamorphic computer virus detection given above, we will now appraise critically the novelty and contribution of our approach. A natural way to compare a variety of metamorphic computer virus detection methods would be to compare empirically their effectiveness with respect to a set of known metamorphic computer viruses; however, much of the work in the literature, including the work in this chapter, is at the proof-of-concept stage, and therefore the tools for such a comparison do not exist. Therefore, the comparisons in this section will be qualitative.

First, we will examine the benefits of the formal nature of our approach in comparison with the both formal and informal approaches above. Then, we will contrast our general, theoretical approach to metamorphic computer virus detection to some of the related work, which may be described as more specific and engineering-based. Finally, we will discuss the applicability of the techniques described in the chapter to problems beyond detection of metamorphic computer viruses.

2.9.2.1 Static and Dynamic Analysis

Most of the approaches described earlier tackle metamorphic computer viruses using static analysis, in which virus code is scanned and analysed. The results of this analysis can then be used to detect metamorphic computer viruses in a variety of ways. One exception is detection by emulation, which is a form of dynamic analysis. As we have demonstrated in this chapter, the formal Maude specification of Intel 64 assembly language is very flexible, and can be used for both static and dynamic analysis of metamorphic computer virus code. When combined with the Maude term rewriting engine, the specification of Intel 64 becomes executable, essentially providing an interpreter/emulator for that language. In addition, the interpreter is formally specified; as long as we agree with the definitions of the syntax and semantics of Intel 64 in the Maude specification, we can be satisfied that the results of dynamic analysis are correct. This is a clear advantage over virtual machines or emulators, whose bugs may result in incorrect analysis of code. Of course, the Maude term rewriting engine is a software application also, and may also contain bugs. However, unlike virtual machines and emulators, the Maude software has been designed with formal verification in mind, and is open source, well-documented and well-supported. In addition, the Full Maude specification language is defined in terms of Core Maude [29], further reducing

the likelihood of problems due to bugs in the code. Another advantage of the Maude specification of Intel 64 is that it is easier to modify than an emulator. For example, Intel recently updated the 32-bit instruction set to include 64-bit instructions [75]. In order to support these new instructions, an emulator would need to be re-programmed and checked for bugs. However, support for the new instructions could be added easily to the Maude specification, by declaring new operators and equational rewrite rules in the same manner as the existing instructions.

As well as providing a formal tool for dynamic analysis, we showed earlier that the specification can also be applied to the problem of static analysis. Suppose we have two fragments of code which we know are semi-equivalent based on dynamic analysis. Then, as we showed in Examples 1 and 2, we can use the Maude specification to derive the values of V_{in} and V_{out} for the fragment of code which follows. Then, by applying the equivalence-in-context theorem, we can prove that semi-equivalent code is actually equivalent in context.

Therefore, the Maude specification of the Intel 64 assembly language described in this chapter is useful for both static and dynamic analysis. Indeed, we are not limited to the methods given here; it seems likely that there are other forms of static analysis, similar to the equivalence-in-context theorem, which can be supported by the Intel 64 specification. Dynamic analysis could be expanded too, to include applications to problems posed by virtualization-based malware. We discuss the potential of this methodology in more detail in Chapter 5.

2.9.2.2 Formal and Informal Approaches

Most of the approaches to metamorphic computer virus detection described above are based on some description of the syntax and semantics of a programming language. (The only exception is the approach of Yoo & Ultes-Nitsche to the detection of metamorphic computer viruses using neural networks, in which the semantics of the program being analysed are completed ignored, as the program code is treated only as data.) Perhaps then, the most distinctive feature of our approach to metamorphic computer virus detection is that the description of the programming language is both explicit *and* formal, i.e., it is based on a formal specification of the syntax and semantics of an assembly programming language written in a formal specification language. In contrast, many of the other approaches to detection, perhaps with the exception of the work by Christodorescu et al (see Section 2.9.1.2), are informal. For example, in control-flow analysis (see Section 2.9.1.1), the flow of control is extracted from a program based on an implicit assumption about the way that looping instructions work, i.e., they update the value of the instruction pointer. Based on this assumption, the control-flow graph

is constructed. Another example is Bruschi et al's approach to program rewriting and normalisation, in which a program is translated into a meta-representation based on an implicit knowledge of the behaviour of the program's instructions.

The advantage of a formal specification of a virus's programming language is that it is possible to prove properties of a section of code, which in turn allows for the development of methods of analysis which themselves are formally verifiable. A good example is the proofs of the equivalence of viral code in Sections 2.6 and 2.7. Assuming that we know that the implicit formal specification in Maude is accurate, then given the existence of reduction as proof, then by performing reductions within Maude we can prove a property of a program (in this example, its equivalence to another program) using a number of reductions in Maude. Checking the accuracy of the formal specification is equivalent to checking the accuracy of the axioms within a logical system, that is, we formulate the formal specification of the Intel 64 assembly language with truths (i.e., axioms) that we hold to be self-evident. For example, in the specification of the `mov` a, b instruction which assigns the value of variable b to variable a, then we specify that this the value of variable a after executing `mov` a, b as equal to the value of b before we executed the instruction using the following equational rewrite rule, which expresses this truth formally:

```
eq S ; mov V,E [[V]] = S[[E]] .
```

The danger in using an implicit and/or informal description of the programming language is that our assumptions are not made clear, and therefore any detection method or program analysis based on the description may not do the job it is designed to do. (The advantages of formal systems are well known, of course, and are discussed in more detail in Section 1.3.)

However, there is an obvious disadvantage to using a formal approach to program specification, verification and analysis. In order to reap the rewards of a formal specification of a programming language, first we must create it, which itself can be a time-consuming, but nevertheless straightforward, process. For example, in order to define the syntax and semantics of a 10-instruction subset of the Intel 64 assembly language instruction set for the proofs in Section 2.6, a Maude specification of around 180 lines had to be produced, which can be seen in Appendix A. The main difficulty was not in the writing or debugging of the Maude specification, but rather in the translation from the informal and implicit definitions of the instructions given in the official Intel literature (see [75]).

Once created, though, a formal specification of an assembly programming language could be applied to a number of different problems in the field of computer virology. For example, the approach of Lakhotia and Mohammed to control- and data-flow analysis

resulted in a rewritten version of a program called a zero form. The specification of Intel 64 could be used to prove the equivalence of the original program and its zero form through dynamic analysis in manner of Section 2.6. Another example would be in the code normalisation procedure described by Bruschi et al, in which the code is transformed into a meta-representation. A formal specification of the syntax and semantics of the meta-representation could be written in Maude in a similar manner to the Maude specification of Intel 64, and the translation of the Intel 64 into the meta-representation could be then formally verified through proofs that an instruction and the translated form have the same effect on a generalised store. (Indeed, a similar approach was taken by Hamel & Goguen, who use a formal OBJ specification to prove the correctness of an optimising compiler [68].)

2.9.2.3 Generality and Readiness for Application

Since one of the principal aims of the field of computer virology is to produce tools for improved management of the computer virus threat, an important consideration is the additional development time that might be required to produce a general-purpose working implementation from "proof-of-concept" systems described in research papers. For example, the approach of Chouchane & Lakhotia to metamorphic computer virus detection involves an observed characteristic of existing viruses: that metamorphism engines used by several different viruses tend to output similar code, and that this can be detected using static analysis. It seems likely, given the tool described in their paper, that the technique could be readily applied within real-life anti-virus software, as the technique described is straightforward and could be applied using well-proven string matching algorithms used within the majority of existing anti-virus software. In contrast, the techniques described in this chapter rely on a formal specification of Intel 64 in Maude, which requires a full Maude interpreter, or alternatively some specialised version of the interpreter designed to use only the specification of Intel 64. It is very unlikely that this capability currently exists within anti-virus software, and therefore there would be an increased cost of implementation.

However, the additional cost is mitigated by the generality of the approach. Whereas the metamorphic engine approach to detection would be a straightforward extension of existing technology, it relies on a particular feature of metamorphic computer viruses that is present today, but may not continue. For example, if metamorphic computer viruses did not re-use their metamorphism engines, then the technique for detection would not work. Alternatively, if metamorphic computer virus technology allows for behavioural mutation of a metamorphism engine, then again, the technique would not work. In contrast, our approach is based on a formal specification of an assem-

bly programming language, and the methods for proving program equivalence and equivalence-in-context are applicable to any programs written in this language. Even if the predominant programming language in which computer viruses are written should change from Intel 64, the general technique of proving equivalence and semi-equivalence is applicable as long as there is a formal specification of that language in Maude. Given the success of the Rewriting Logic Semantics Project discussed in the next section, it is likely that such a specification would indeed be possible. In addition, the equivalence-in-context technique given in Section 2.7 should be applicable to any programming language in which statements are executed sequentially, and in which variables are the means of data storage.

Therefore, it could be argued that whilst the cost of implementation of our approach could be greater than other approaches given in the literature, our approach is more general and therefore more likely to be useful for a greater period of time, and in a greater number of scenarios.

2.9.2.4 Applications Beyond Computer Virology

The applicability of the Maude specification of Intel 64 goes beyond detection of metamorphic computer viruses. It was mentioned in Section 2.1.1 that the practice of formal specification using Maude is well-understood, and has its roots in the formal specification language OBJ. The syntax and semantics of many programming languages have been given using Maude and OBJ [105], including generic imperative programming languages [58, 59], Java (both the language [128] and compiled bytecode [41, 42]), Scheme [35], Concurrent ML [25], the ABEL hardware description language [81] and the C preprocessor [53].

The Intel 64 specification also supports one of the primary aims of the Rewriting Logic Semantics Project [105], which exists to develop formal specifications of various programming languages in order to provide a formal software verification infrastructure. The Intel 64 programming language is used on the majority of personal computers at the time of writing, and therefore the formal specification presented in this thesis has numerous applications.

There are numerous advantages to specifying programming languages in Maude. The resulting specifications are general purpose, and can be used to prove properties of Intel 64 assembly language programs. The executable nature of Maude specifications means that we obtain an interpreter for specified languages, such as Intel 64, essentially for free. There are several generic formal tools based on Maude that can be used with any specification of a language, such as the Maude model checker and inductive theorem prover [105]; therefore, we do not need to develop these tools for Intel 64, as they are

part of the Maude framework in which Intel 64 has been specified.

To our knowledge, our formal algebraic specification of the syntax and semantics of a subset of the Intel 64 assembly programming language is novel. Whilst we have not yet specified every instruction within the language, we have proven that such a specification is straightforward to produce using a specification language like Maude. Indeed, based on the numerous applications of Maude to programming language specification in the literature, we are confident that the rest of the Intel 64 instruction set can be specified in a similarly straightforward manner.

Chapter 3: Formal Affordance-based Models of Reproduction

This chapter describes a formal approach to modelling reproduction, and formal methods for classification and refinement of reproduction models. The obstacles faced by those seeking a rigorous definition of reproduction were explored by Nehaviv & Dautenhahn [108], in which they identified a number of recurring problems in this domain:

- Von Neumann required that non-trivial reproducing automata be capable of universal computation and construction [149]. However, a biological cell is a reproducer that can be argued to have neither. Therefore, strictly formal definitions of reproduction, such as von Neumann's, can often contradict the intuition about reproductive systems.

- Identifying reproduction is a direct result of observation, i.e., definitions of reproduction may vary with observer bias. This could conflict with an idealised Platonic notion of the "logic of life", i.e., some set of formal rules which all living things must display.

- Many observed forms of reproduction include dissimilarity between parents and offspring, and multiple parents. Many formal definitions of reproduction, such as those by von Neumann, do not take these factors into consideration.

In this chapter we do not intend to establish a rigorous definition of reproduction or life. Instead, we wish to establish a means for modelling behaviour which we have already identified as being reproductive. We hope to demonstrate that such an abstract, systems-based approach can be beneficial to our understanding of the nebulous concepts of reproduction and life.

3.1 Introduction

The classification of life, both natural and artificial, is relevant to several fields, including artificial life, biology and computer virology. In order to develop a classification of life forms, one must first determine what constitutes the class of living things. Defining the boundary between animate and inanimate often gives false positives and negatives. Reproduction, on the other hand, is comparatively simple to define, and is an essential characteristic in most definitions of what it means to be alive (see, e.g., [152, 130]).

A common informal definition of reproduction is that it is the act of producing offspring. Reproducers are then the actors which engage in reproduction. Of course, this leaves the problem of what is meant by "producing offspring", but as it is partly the intention of this chapter to illuminate this problem, we shall ignore this problem until Section 3.3. Similar terms to "reproduction" exist in the literature, including "production", "self-production", "self-reproduction", "replication" and "self-replication". Some of these terms are used to distinguish between unaided and aided reproduction, however, we take "reproducer" (and consequently, "reproduction") to mean any actor which engages in an act of reproduction, regardless of its level of involvement in the reproductive process.

In the literature there are clear and paradigmatic examples of reproducers: biological organisms and the genes that control them [36], von Neumann's reproducing automaton [149], computer viruses [31] and other forms of reproducing malware [43], and so forth. However, there are other examples that stretch intuitive definitions of reproduction: photocopies [71], gliders in cellular automata [51], seeding crystals, fixed points of functions, or even a pen on a desk which, in being a static object, "reproduces" from one instant to the next thanks to the physical laws of the Universe.

An important distinction between reproduction systems, that might help to distinguish between paradigmatic and "rogue" examples, is whether the reproducer appears to rely on external agency, or not. For example, biological viruses are often argued to be on the boundary between animate and inanimate [148] — perhaps this is partly a result of the fact that their reproductive process involves the "hijack" of the reproductive process of a host cell.

3.1.1 The Theory of Affordances

Gibson's theory of affordances describes the functional relationships between an animal and its environment [55]. For an animal, an affordance is an "opportunity for action." For example, a piece of food affords an animal nourishment, a tree affords it the ability to climb to safety, and a cave affords shelter. We use affordances to form a classifica-

tion of reproducers based on the functionality that they afford to themselves and the functionality that their environment affords them, with respect to reproduction. Gibson also describes a niche in an environment, into which an organism fits, as a "set of affordances." Therefore, in this chapter we categorise different reproducers according to the sets of affordances corresponding to their reproductive niches.

Here we use "affordance" metaphorically; we apply it to reproducers rather than animals, replacing "animal" with "reproducer" in Gibson's original definition. If we allow this slight expansion of the definition of the word, this affordance-based ontology can afford us an intuitive and novel way of approaching the classification of reproducers.

For historical reasons "affordance" has come to mean "perceived affordance" in the domain of human–computer interaction [144]. It is important to note that apart from the slight expansion of the definition that we mention above, we take "affordance" to mean "opportunity for action", as in the original definition given by Gibson.

3.1.2 Structure of the Chapter

The rest of this chapter is devoted to the demonstration of the applicability of the affordance metaphor described above to the problem of reproduction modelling, classification and refinement. Gibson presented the theory of affordances as an "ecological theory of perception", in which the relationship between an organism and its environment could be described in terms of the relationship between the organism and other entities within its ecology which afford some action. We could perhaps describe our approach to reproduction modelling and classification based on affordances as an ecological theory of reproduction.

In Section 3.2 we give an informal introduction to the application of the affordance metaphor to reproduction through a classification of reproduction into four main types, which we call Types I, II, III, and IV. These classes are based on predicates concerning whether the sets of actions corresponding to the self-description and reproduction mechanism of the reproducer are assisted by entities other than the reproducer, or not. We show how this approach can be used to classify trivial and non-trivial reproducers differently. However, the informality of this approach leads to a number of unanswered questions, which in Section 3.3 motivates the formal approach to reproduction modelling and classification.

In Section 3.4 we give formal definitions of our formal models of reproduction. We describe how affordance-based reproduction models can be classified as unassisted or assisted, trivial or non-trivial. In order to relate different reproduction models, we describe a formal notion of refinement, in which one model can be refined to another.

In Section 3.5 we prove that all reproduction models, both unassisted and assisted,

have a related model that has the opposite classification; these are presented as the Unassisted and Assisted Reproduction Theorems, respectively. We prove that the reproduction model space is structured, as non-trivial reproduction models cannot be refinements of trivial models, and trivial and assisted reproduction models cannot be refined to trivial and unassisted reproduction models. We generalise the approach from Section 3.2, based on notions of affordance of the self-description and reproduction mechanism, in terms of aspects, and prove that the class Type I is equivalent to the class of unassisted reproduction models. At every stage we illustrate our approach with worked examples, in which reproduction models of reproducers from biology and computer virology are defined, refined and classified. Then, in Section 3.6 we demonstrate the application of affordance-based reproduction models and the Unassisted and Assisted Theorems to examples of reproduction from artificial life.

In Section 3.7 we summarise our work, and explain how the questions raised in Section 3.2 have been answered by formal affordance-based reproduction models. We describe how our work compares with similar work in mathematics, artificial life and theoretical biology, and based on this, give a critical appraisal of the work described in this chapter.

3.2 One Possible Classification Scheme

One of the earliest formal analyses of reproduction was given by von Neumann [149]. One of the aims of his work was to prove by construction the existence of a formalised automaton capable of reproduction (see Figure 1.2). Von Neumann split his automaton into two separate parts: a self-description stored on a tape (analogous to a Turing machine tape), which was passed as input to a constructor capable of fabricating any configuration of cells within the cellular automaton in which the reproducing automaton existed. This constructor was therefore a *universal* constructor; "universal" having an analogous meaning to "universal" Turing machines. Von Neumann's reproducing automaton was highly complex and based on a complex cellular automaton, but even the subsequent, and much more simple reproducers such as Langton's loops [89] still retain an architecture based on a self-description and reproductive mechanism.

Affordance theory lets us talk about a particular act by an actor within an environment being attributable in some way to some other actor. Therefore, given the acts corresponding to the acquisition of the self-description, and the use of it by the reproductive mechanism in order to complete the act of reproduction, it is possible to create a classification based on whether these acts are are afforded by entities external to the reproducer, or not. Since these acts seem essential to reproduction, it is reasonable to

classify reproducers according to this scheme.

Therefore, we will present a classification which is based upon this division of an act of reproduction into self-description and reproductive mechanism phases. As well as von Neumann's reproducing automaton and Langton's loop, this abstract architecture is used by biological organisms, in the form of the genetic code (self-description) and the complex biological machinery which translates this code into offspring (reproductive mechanism). Biological viruses also use this architecture, although a significant part of their reproductive mechanism is given by an external entity: whichever organism the virus infects enables the reproduction of the virus, since viruses typically lack a sufficient reproductive machinery in order to complete their reproductive process. Other well-known reproducers such as Tierran organisms [118] generate a self-description by self-analysis. Their self-descriptions are not encoded as is the case with a genome, however the organisms still provide the description to themselves by their reproductive behaviour. The organisms copy their self-description one byte at a time to create an offspring, which constitutes the organism's reproductive mechanism. Computer viruses and other forms of reproducing malware can generate a self-description in a similar way to the Tierran organisms (i.e., by self-analysis), or they can carry an encoded self-description within their own code (e.g., the Zmorph virus described in Chapter 2). The reproductive mechanism is then the algorithm which creates a copy (offspring) of the virus based on the self-description. More "trivial" reproducers such as photocopies still employ a self-description and reproductive mechanism in their reproductive processes, although they are given by an external entity (the photocopier). Cellular automaton gliders such as those seen in Conway's Game of Life [51] can also be seen to have a self-description and reproductive mechanism, although it is given to the reproducing gliders by an external actor: the self-description is contained in the state of the cellular automaton, and the reproductive mechanism is the state transition rule which creates the offspring (to be contained in the successive state).

The first step in classification involves identifying the parts of the reproductive process which correspond to the self-description and reproductive mechanism. Next we must determine which actors assist in the acts of self-description and reproductive mechanism. As we shall see in the forthcoming descriptions, definitions and examples, if the reproducer is not assisted in its reproductive process, i.e., if it is not afforded its self-description and reproductive mechanism by an external entity, then we classify it as "Type I". A reproducer which is afforded either its reproductive mechanism or its self-description is either "Type II" or "Type III" respectively. A reproducer which is afforded both the self-description and reproductive mechanism by an external entity is "Type IV".

Therefore, we can form a two-dimensional classification, with a dichotomy on each dimension. These four types are therefore the four corners of a reproducer classification space, and reproducers (such as biological organisms, artificial organisms, and computer viruses) occupy different points within this space depending on their reproductive reliance on themselves and the external actors in their environment.

3.2.1 Type I Reproducers

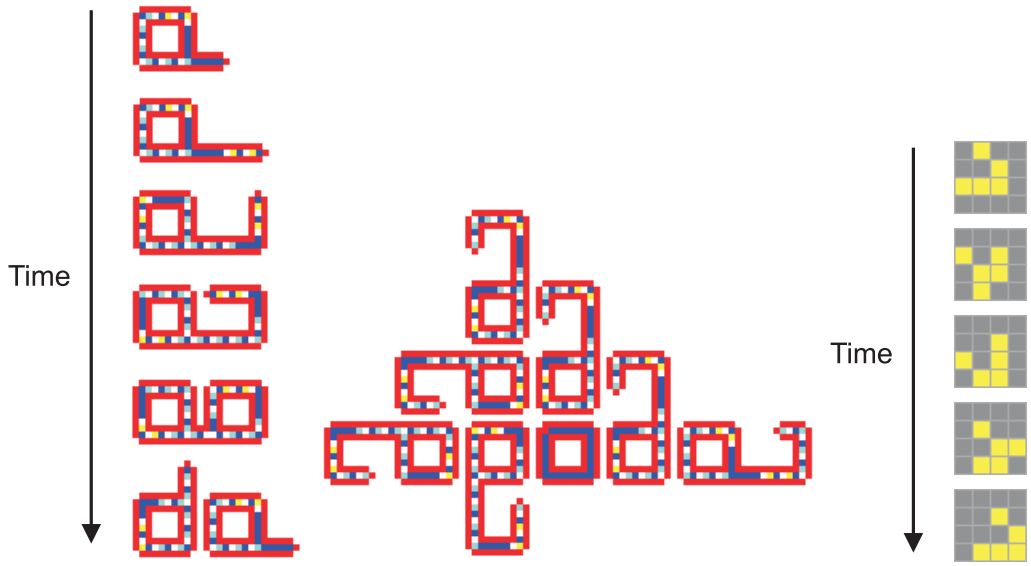
Type I reproducers do not have any part of their reproductive process, either the self-description or the reproductive mechanism, afforded to them by any other actor in the reproductive system.

3.2.1.1 Example: von Neumann Reproducing Automaton

Von Neumann's reproducing automaton (see Figure 1.2) was the first mathematical model of a reproductive process. Von Neumann identified that reproduction is possible when the reproducer has a self-contained self-description, and a reproductive machinery that interprets the self-description as a set of instructions for producing an offspring [149]. From an abstract point of view, this is the same process that microorganisms (such as bacteria) undergo during reproduction, where the genome (qua self-description) is interpreted by the reproductive machinery within the organism to produce an offspring. We will call this mode of reproduction, in which the acts of self-description acquisition and reproductive mechanism are not afforded by any actor other than the reproducer itself, Type I. The von Neumann reproducer, which reproduces on a cellular automaton grid, is completely self-reliant with respect to reproduction — the self-description consists of a tape that is attached to the automaton, and the reproductive mechanism of the automaton consists of the method by which the tape is read and translated into a sequence of instructions, which are fed to the constructing arm in order to construct the offspring.

3.2.1.2 Example: Langton's Loop

Langton's loops are reproducing patterns on a cellular automaton grid (see Figure 3.1). They consist of an outer sheath which encapsulates a sequence of cells with various states (a data signal), which effectively correspond to instructions to the construction arm for building an offspring loop [89]. The self-description here is the data signal within the sheath, and the reproduction mechanism is the means by which the instruction in the self-description are "interpreted" by the constructing arm. Both the self-description and reproductive mechanism are self-contained, and therefore this particular case of



(a) Langton's loops. (b) Loops beget loops. (c) Game of Life glider.

Figure 3.1: Reproduction in cellular automata. Figure 3.1(a) shows the arm extensions that complete the reproductive process of Langton's loops, and Figure 3.1(b) shows a "colony" of loops generated by one initial loop. Figure 3.1(c) shows all five states of a Conway's Game of Life cellular automaton corresponding to the reproductive process of a glider. Image sources: [5, 101].

reproduction belongs in Type I.

3.2.2 Type II Reproducers

A Type II reproducer differs from Type I in that its reproductive mechanism is afforded by some actor, present in the reproducer's environment, and which is separate from the reproducer.

3.2.2.1 Example: Tape from von Neumann's Reproducing Automaton

An example of a Type II reproducer would be the self-description input tape from von Neumann's reproducing automaton. This tape is has no construction abilities at all. If it is to reproduce, it must be interpreted by a constructor present in its environment. Therefore, the tape reproduces has a self-description, but lacks a reproductive mechanism, which must be afforded by some separate actor in the tape's environment. The tape therefore qualifies as a Type II reproducer.

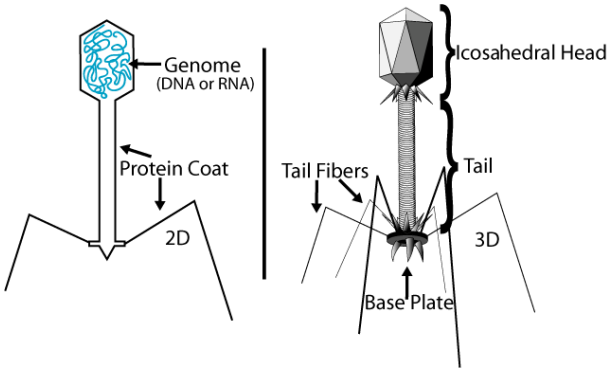


Figure 3.2: The T4 bacteriophage virus. The legs of the T4 attach themselves to the cell wall of a bacterium. Then, using the base plate and tail as an injection mechanism, the genetic material in the head is transferred to the interior of the host cell, where it hijacks the reproductive mechanism of the cell in order to reproduce. Image source: [77].

3.2.2.2 Example: T4 Bacteriophage

A T4 bacteriophage virus contains a self-description in the form of its genome, encoded as DNA or RNA (see Figure 3.2). However, the virus requires the help of its host cell in order to reproduce, and therefore we can say that the reproductive mechanism is afforded to the virus by the host cell, which we present as a separate actor. Therefore, the T4 bacteriophage is a Type II reproducer. (The reproductive behaviour of the T4 bacteriophage and its classification are presented in more detail in Section 3.4.)

3.2.2.3 Example: Source Code Computer Virus

Source code computer viruses carry around with them a copy of their own source code [43]. When they find a compiler on a host system, they use the compiler to create an offspring based on the source code. Therefore, source code viruses have their own self-description (source code), but require the help of another actor (the compiler) for the reproductive mechanism. Therefore, source code viruses can be classified as Type II, and are distinguished from typical parasitic computer viruses, which would naturally be classified as Type I.

3.2.3 Type III Reproducers

Type III reproducers differ from Type I reproducers in that their self-description is afforded by some actor, which is present in the reproducer's environment and is separate from the reproducer.

3.2.3.1 Example: Compiler

A compiler is a constructor of programs, which takes as its input a sequence of symbols in some programming language (i.e., source code), and produces as output some program corresponding to the sequence of symbols given as input. A compiler can be given its own source code (self-description) as input, and as a result it will create a copy of itself based on these instructions. Therefore, the compiler can only reproduce by being given its source code as input by the user. Therefore, we can say that the user is an entity in the compiler's environment which affords the self-description to the compiler.

3.2.3.2 Example: Damaged Cell

It is difficult to think of a biological example approximating Type III reproduction. However, we might imagine a cell that has been damaged so that it no longer contains any genetic material. In order to complete its reproductive process, the cell requires the help of some external entity that will provide a self-description. For example, one could imagine some means by which a genome is inserted into the cell by a specially-engineered virus. Therefore we can say that the virus affords the self-description to the cell, and therefore the cell is a Type III reproducer.

3.2.4 Type IV Reproducers

Type IV reproducers are those that are afforded both their self-description and reproductive mechanism by actor(s) in the reproducer's environment, and which are separate from the reproducer.

3.2.4.1 Example: Game of Life Gliders

Cellular automaton (CA) gliders in Conway's Game of Life [51] are able to reproduce trivially thanks to the transition rule of the CA. Here, the state of the CA at a particular instant stores the glider's description, and therefore acts as a self-description for the glider. The reproductive mechanism is provided by the transition rule which maps one state of the CA to the next, and causes the glider to reproduce to a new point on the CA grid. Therefore a glider is a Type IV reproducer, as its self-description and reproductive mechanism are both afforded by the cellular automaton in which the glider resides.

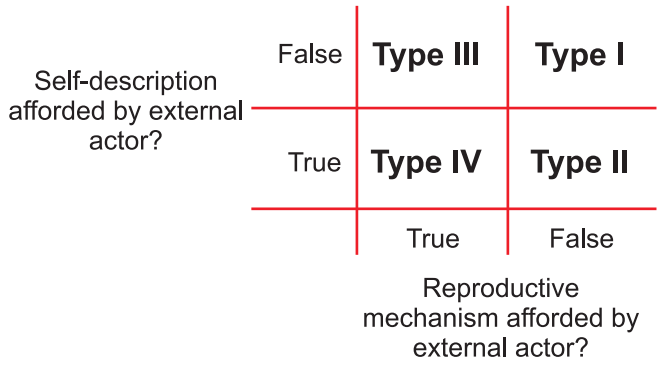


Figure 3.3: Informal reproducer classification based on affordances. Notions of "self-description" and "reproductive mechanism" result in a two-dimensional classification, with a dichotomy on each dimension.

3.2.4.2 Example: The Photocopy

The information stored in writing on a piece of paper is able to reproduce trivially in an environment which contains a photocopier machine. A piece of paper is fed into the photocopier, at which point it is scanned and a representation (digital or otherwise) which captures the appearance of the piece of paper is created. This representation is a self-description for the piece of paper. This representation is then fed to the printer within the photocopier, which creates a facsimile of the original piece of paper in the form of a photocopy. This printing process is the reproductive mechanism for the writing on the paper. The photocopy is therefore afforded its self-description and reproductive mechanism by the photocopier machine.

3.2.5 Questions about Affordance-based Classification

In this section we have seen how it is possible to classify reproducers informally using affordance theory. However, there are several questions that have been raised. The questions described in this subsection will be answered in Sections 3.3–3.5.

3.2.5.1 The Assisted Reproduction Conjecture

We showed that a glider within a cellular automaton can be classified as Type IV, because its entire reproductive process is performed by the cellular automaton in which it resides. Essentially, we are presenting the cellular automaton as an actor present in the glider's environment. Within cellular automata, the state of the automaton together with the state transition system form the laws of the universe within which different patterns can exhibit their behaviours. So, by taking the cellular automaton as an actor,

we have portrayed the underlying laws of the reproducer's environment as an actor. This is bound to result in classification as Type IV, as the self-description, reproductive mechanism, and indeed, anything else which occurs within the cellular automaton must happen as a result of the underlying laws of the environment. However, this raises a question: can we portray the underlying laws of the environment as an actor for any other reproducers, and therefore classify them as Type IV as well?

The answer would seem to be, "Yes." For example, the von Neumann reproducing automaton reproduces within a cellular automaton as well. If we portray the cellular automaton as an actor within the environment of the reproducing automaton (as we did with the glider), then it is rational to assume that every act within the cellular automaton grid must be afforded by the cellular automaton, as these acts could not take place without its presence. Therefore, the von Neumann reproducing automaton is afforded its acts of self-description acquisition and reproductive mechanism by the cellular automaton in which it resides, and therefore it must be a Type IV reproducer. However, we have already classified it in Section 3.2.1.1 as a Type I reproducer. In effect, by looking at the reproduction of the reproducing automaton in a different way, we have caused its classification to be changed from Type I to Type IV. In principle, it should be possible to take any non-Type IV reproducer and re-classify it as Type IV by simply considering there to be some actor, which is responsible for reproductive actions (and perhaps more). In the most extreme case, we can thing of this actor as being a "god-like" presence within the reproductive system, responsible for everything that happens within the system.

Based on the above, we can form the assisted reproduction conjecture: is it possible to re-classify any unassisted reproducer as assisted?

3.2.5.2 The Unassisted Reproduction Conjecture

If it is possible to introduce an actor that takes responsibility for everything that happens during an act of reproduction, then could we also re-define the actors present during reproduction so that the reproducer takes sole responsibility for the act of reproduction? In other words, could we "aggrandise" the reproducer so that it then affords itself everything in its act of reproduction that was previously afforded to it by some external entity?

Again, the answer would seem to be, "Yes". Let us use the example of the cellular automaton glider once again. In the Type IV classification of the glider, the self-description and reproductive mechanism were afforded by the cellular automaton to the glider. However, suppose we imagine a new actor, a conglomeration of the cellular automaton plus the glider. This conglomeration is defined in such a way that

everything that was previous afforded by the actors that make up the conglomerate to another actor is now afforded by the conglomerate to that actor. In the first model the cellular automaton afforded the reproductive acts of self-description acquisition and reproductive mechanism to the glider, and therefore in the updated model, the conglomerate of the cellular automaton and the glider now affords these acts to the glider — which is now a part of the conglomerate, and therefore these acts are now afforded by the conglomerate to itself. Since there is no other actor within the system that affords reproductive actions to the reproducer, we know that the conglomerate-reproducer is classified as Type I[1].

Based on the above, we can form the unassisted reproduction conjecture: is it possible to re-classify any assisted reproducer as unassisted?

3.2.5.3 Varying Degrees of Assistance

Reproducers outside Type I necessarily lack some essential part(s) of their reproduction mechanism: either the self-description, the reproductive mechanism, or both. These lacking parts must be provided by an external entity. However, some reproducers outside Type I may afford themselves an action which assists in the acquisition of a self-description and/or a reproductive mechanism in order to complete their reproductive process. For example, the T4 bacteriophage given in Section 3.2.2.2 has an injection mechanism for inserting its genetic code into a host cell: its "legs" attach to the cell that it will infect, and the virus penetrates the cell in order to inject its genetic code.

It is possible to imagine a T4 variant that is "legless", that is, it is exactly the same as a normal T4 bacteriophage except that it lacks an injection mechanism. However, the T4 variant is still classified as Type II because it is afforded a reproduction mechanism by the cell (assuming that the genetic code can somehow still enter the host cell), but the self-description is not afforded by any other actor. However, there is clearly a difference in the amount of assistance required by the T4 and its variant.

It is likely that there are many other examples of varying degrees of assistance, e.g., different computer viruses that require the assistance of the operating system may require varying degrees of assistance. How, then, can we specify or quantify this degree of assistance in reproduction? Could different degrees of assistance be used to form further sub-classes of Types II, III and IV?

[1]Of course, this assumes that the conglomerate of the cellular automaton and the glider is still a reproducer. However, we know that this is the case, as the both the cellular automaton and the glider are present at the start and end states of reproduction, and therefore we can say that they have reproduced in some manner.

3.2.5.4 Other Means of Classifying Reproducers Using Affordances

So far we have only looked at the abstract acts of self-description acquisition and reproductive mechanism as a means of classification by affordances. It is likely, however, that there may be other acts with which we can classify reproducers. For example, we could talk only about the overall act of reproduction (including both self-description and reproductive mechanism phases), and whether it is afforded by an actor that is separate from the reproducer, or not. Classification of this type would result in two different classes of reproducer, one in which the reproducers require the assistance of other actors in order to reproduce, and one in which they do not.

Other possible dimensions of affordance-based classification may be based on particular sub-classes of reproducer, e.g., computer viruses. We may wish to define an abstract act corresponding to the payload of the computer virus, for example, or we may wish to identify all of the acts (e.g., statements) of a computer virus that correspond to a particular kind of statement, e.g., those which use operating system API functions.

Clearly we should not limit affordance-based classification of reproducers to the acts of self-description acquisition and reproductive mechanism. We could expand affordance-based classification to include other acts, such as the payload of a computer virus, but then we face the same problem should we wish to extend the classification further. The best way of dealing with this, then, might be to separate the different possible acts on which we base our classifications from the act of affordance-based classification itself. For instance, we could define a general way of defining a reproductive act such as the self-description, and a general way of checking whether it is afforded by some other actor to the reproducer, and combine the two (if we so wish) to create a particular classification of the reproducer space. The challenge, then, is to find such a method of generalised affordance-based classification of reproducers.

3.3 Towards Formal Reproduction Models

The discussion thus far has focused on the distinguishing characteristics of different classes of reproduction. These characteristics were described in terms of the actions afforded by entities to other entities. The identification of the entities and actions involved in a process, and the allocation of affordances among these entities, depends upon the level of abstraction at which the process is viewed: in other words, what we are characterising is not so much reproduction itself, as models of reproduction.

In Section 3.2.5 we described four issues that were raised by our informal reproduction classification. In order to answer these questions, in the next section we introduce

formal notions of reproduction and classification, designed to capture formally the notions of entities, assistance and affordance which we have so far used only informally.

We start by re-formulating the unassisted and assisted reproduction conjectures as formal statements, which in Section 3.5 we prove to be true. The third question, on how we can capture the varying degrees of assistance, is answered in the formulation of reproduction models in Section 3.4 in which different actions within the act of reproduction are individually identified as being afforded by some set of entities. The final question, on whether there are other abstract actions for classifying reproducers than self-description and reproductive mechanism, is answered in Section 3.5.3 by generalising this kind of classification using "aspects".

3.4 Formal Models of Reproduction

Our goal is to classify and study the relationships between models of reproductive processes in a rigorous way. In this section we specify precisely what we mean by a model of a reproductive process: on the one hand, we want our notion of model to be general enough to cover as many examples as possible, while on the other hand, we want the notion to have enough structure to allow us to capture relevant similarities and differences between specific models. Since any model of reproduction necessarily identifies some *thing* that reproduces, it seems reasonable to take an individual-based approach to modelling reproduction, and we will assume that reproductive models identify a collection of individuals that play some role in the reproductive process, and that the reproducer itself is a particular individual in this collection. We also assume that the reproductive model specifies a state space and the events that occur to move from one state to another.

A state space together with events or "actions" that move the system from one state to another form a *labelled transition system*, which consists precisely of a set S of states, a set A of actions, and a ternary relation $\longmapsto \subseteq S \times A \times S$ specifying the transitions between states. Given such a labelled transition system, we usually write $s \overset{a}{\longmapsto} s'$ instead of $(s, a, s') \in \longmapsto$, to indicate that action a may move the system from state s to state s'. As an example, consider the life cycle of a bacteriophage virus, which consists of five stages: (i) **a**ttachment of the virus to the host cell; (ii) **i**ntroduction of the virus's genome to the interior of the cell; (iii) **s**ynthesis of new virus parts; (iv) **m**aturation of these parts into mature offspring; and (v) **r**elease of these offspring back in the environment. At this schematic level, there are five actions: $A = \{\mathbf{a}, \mathbf{i}, \mathbf{s}, \mathbf{m}, \mathbf{r}\}$, and the state space has six states: $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, where s_1 represents the initial state before the bacteriophage attaches to the cell, s_2 represents the state after

attachment, and so forth. The labelled transition system as a whole can be pictured thus:

$$s_1 \overset{\mathbf{a}}{\longmapsto} s_2 \overset{\mathbf{i}}{\longmapsto} s_3 \overset{\mathbf{s}}{\longmapsto} s_4 \overset{\mathbf{m}}{\longmapsto} s_5 \overset{\mathbf{r}}{\longmapsto} s_6 \tag{3.1}$$

In this schematic model of the bacteriophage, we might posit just two entities: the cell, which is present in all states except the final state, and the bacteriophage, which, as the reproducer, is present in at least the first state s_1, and final states s_5 and s_6. This gives us a very simple model of bacteriophage reproduction, in which we identify the bacteriophage with its own offspring. Note that Cohen [32] adopts such an identification in modelling computer viruses that may change their source code from generation to generation, introducing the terminology "viral set" for the set of all instances of the virus code that might be so generated. (We might say that our approach identifies entities modulo "self sets.") We might take an even more abstract view of identity, and identify the bacteriophage with its genome, in which case we would have a model where the bacteriophage is present in all states.

This very abstract and schematic model of bacteriophage reproduction gives an example of

Definition 6. *A* basic reproduction model *is a tuple*

$$(S, A, \longmapsto, \mathit{Ent}, r, \varepsilon, p) \ ,$$

where

- $(S, A, \longmapsto)$ *is a labelled transition system;*
- Ent *is a set of "entities" with* $r \in \mathit{Ent}$ *the particular entity that reproduces in the model;*
- $\varepsilon \subseteq \mathit{Ent} \times S$ *is a binary relation, with* $e\ \varepsilon\ s$ *indicating that entity* e *is present in the state* s*;*
- p *is a* path *through the transition system representing the reproduction of* r*, i.e.,* p *consists of a sequence* $s_0 \overset{a_1}{\longmapsto} s_1 \overset{a_2}{\longmapsto} \ldots \overset{a_n}{\longmapsto} s_n$ *with* $s_{i-1} \overset{a_i}{\longmapsto} s_i$ *for* $0 < i \leq n$*, and with* $r\ \varepsilon\ s_0$ *and* $r\ \varepsilon\ s_n$*.*

The last item in this definition states that there is at least one path through the transition system that shows that the reproducer does actually reproduce. In the bacteriophage example, the path is the entire system, as pictured in (3.1).

The model of bacteriophage reproduction described above is very much idealised, and *schematic* insofar as the states are just abstract labels for idealised stages in the life

cycle of a prototypical bacteriophage. In this regard, it is very similar to the diagrams that can be found in introductory textbooks on virology. We shall present more concrete models later on, but even at this schematic level of abstraction, we can see that the bacteriophage's life cycle involves an essential interaction between the bacteriophage and a — similarly prototypical — cell. Indeed, it is clear that the bacteriophage requires a cell for all stages in its life cycle, placing it at one end of a spectrum of reliance upon external agency; reproducers at the other end of the spectrum are those that are able to reproduce without the help of other entities: these might include single-celled organisms, or von Neumann's reproducing automaton, for example.

In order to capture a notion of assistance within our models, whereby certain actions happen as a result of the presence of one or more entities, we postulate a function *Aff* that assigns to any action a the set of entities, $\mathit{Aff}(a)$, that mutually *afford* the action a, i.e., those entities without whose presence a could not be performed. We make this formal in the following

Definition 7. *An* affordance-based reproduction model *is a tuple*

$$(S, A, \longmapsto, \mathit{Ent}, r, \varepsilon, p, \mathit{Aff}) \ ,$$

where $(S, A, \longmapsto, \mathit{Ent}, r, \varepsilon, p)$ *is a basic reproduction model, and*

$$\mathit{Aff} : A \to \mathcal{P}(\mathit{Ent})$$

such that, for all states s*, if* a *is possible in* s *(i.e.,* $s \overset{a}{\longmapsto} s'$ *for some state* s'*), then* $e \ \varepsilon \ s$ *for all* e *in* $\mathit{Aff}(a)$*.*

For example, in our model of the bacteriophage life cycle, given the set of entities $\mathit{Ent} = \{b, c\}$, with b the bacteriophage, which is present in all states, and c the cell, present in all states except s_6, we might set

$$\mathit{Aff}(a) = \{b, c\}$$

for all actions a. We could read this as saying that all actions are afforded by the cell to the bacteriophage, or that all actions require both the cell and the bacteriophage to be present. Note, however, that this is simply one way of modelling the bacteriophage's life cycle, and so depends entirely upon the intentions of the modeller. It would be just as acceptable, on a formal level, to set

$$\begin{array}{ccccccc} \mathit{Aff}(\mathbf{a}) & = & \mathit{Aff}(\mathbf{i}) & = & \mathit{Aff}(\mathbf{r}) & = & \{b, c\} \\ & & \mathit{Aff}(\mathbf{s}) & = & \mathit{Aff}(\mathbf{m}) & = & \{c\} \end{array}$$

which would imply that only the cell is necessary for synthesis and maturation. Yet again, we might set

$$\mathit{Aff}(a) = \{b\}$$

for all actions a, stating that the bacteriophage relies only on itself for reproduction, giving, once more, a different model to the previous ones, perhaps reflecting a modeller's assumption that, say, cells are a freely available resource.

We shall revisit in Section 3.4.2 the issue of different models of "the same" reproductive process, but with the notion of affordance we now have sufficient theoretical machinery in place to model a wide variety of reproductive systems, and turn now to the question of classifying such systems according to their reliance on external agency.

3.4.1 Classifying Reproduction Models

An obvious next step is to classify reproduction models according to whether or not the reproducer is assisted in its act of reproduction. We shall say that those reproduction models in which reproduction is not assisted by any other entity are *unassisted*, and those in which reproduction is assisted are, naturally, *assisted*.

In order to simplify the definition of assistance, it is useful to think of the set of all entities which aid the act of reproduction. We call this set the *ecology* of a model.

Definition 8. *The* ecology *of a model M, $\mathsf{E}(M)$, is the union of the sets $\mathit{Aff}(a_i)$ for a_i in the path p.*

We can then classify a model M as assisted or unassisted, depending on the ecology of M.

Definition 9. *An affordance-based reproduction model M can be classified as* unassisted *iff there is no entity e, different from the reproducer r, in $\mathsf{E}(M)$. Conversely, a model M is* assisted *iff there is some entity e different from r in $\mathsf{E}(M)$.*

As illustrated in this definition, for brevity we often refer to an affordance-based reproductive model simply as "a model".

As well as classifying models based on assistance, we can classify based on whether the reproducer participates in its own reproduction or not. The idea of a reproducer not participating in its own reproduction might seem paradoxical at first, but is quite natural in some circumstances. For instance, a photocopy does little in the act of its own reproduction. It is likely that many such "trivial" examples of reproduction involve reproducers that do not seem to participate in their own reproduction, e.g., gliders within Conway's Game of Life. We summarise this distinction in

Definition 10. *A model M is* trivial *iff the reproducer $r \notin \mathsf{E}(M)$. Conversely, M is* non-trivial *iff $r \in \mathsf{E}(M)$.*

We now demonstrate how a reproduction model for a copier computer virus can be defined and classified.

Example 5. *The following copier computer virus reproduces when it is executed by the Bourne Again Shell ("Bash") interpreter in Unix:*

```
cp $0 $0.copy
```

The command `cp` *takes as arguments two filenames, and copies the contents of the first file, if it exists, into the second file, which will be created if it does not already exist. The expression* `$0` *is a special variable that is set to the name of the shell script that is currently running.*

In general, for a computer virus, we would like to base a reproductive model on an operational semantics [114] for the programming language in which the virus is written. That is, the labelled transition system has programs as labels, and the states are those of an abstract machine that executes the language. An operational semantics formally specifies the transition relation $s \overset{p}{\longmapsto} s'$ by specifying which states s' may be reached by executing program p in starting state s.

In this case, we can represent the state of a computer running the Bash interpreter as a tuple `FS | B | CS`*, where* `FS` *represents the filestore,* `B` *represents the state of the Bash interpreter, and* `CS` *represents a sequence of shell script commands that are to be executed. For the sake of simplicity, we will assume that the filestore is just a sequence of shell scripts, and we will represent each script as* `[FH : CS]`*, where* `FH` *is the name ("file handle") of the script and* `CS` *is the sequence of shell-script commands in the script. The state of the Bash interpreter would consist of variable–value pairs for all of Bash's environmental variables; since for our example we are interested only in the variable* `$0`*, we will represent the state of the interpreter simply as* `$0 : FH`*, where* `FH` *is the value of the variable* `$0`*. As for the commands, we will restrict attention to names of shell scripts and commands of the form* `cp E1 E2`*, where* `E1` *and* `E2` *are expressions. Thus, for example,*

```
[virus : cp $0 $0.copy] | [$0 : null] | virus
```

represents a state where the only script in the filestore is the copier virus, the variable `$0` *has value* `null`*, and the command about to executed is a call of the shell script* `virus`*.*

We will not spell out the details of the operational semantics for this simplified Bash interpreter here; the interested reader can find a formal description written in

the specification language Maude [105] in Appendix B. For our present purposes, it is sufficient to note that the operational semantics permits the following path:

```
   [virus : cp $0 $0.copy] | [$0 : null] | virus
 get
⟼
   [virus : cp $0 $0.copy] | [$0 : virus] | cp $0 $0.copy
subst
⟼
   [virus : cp $0 $0.copy] | [$0 : virus] | cp virus virus.copy
 cp
⟼
   [virus.copy : cp $0 $0.copy] [virus : cp $0 $0.copy]
     | [$0 : virus] |
```

which shows that executing the virus causes its code to be reproduced.

In this model we identify three entities: the copier computer virus (v), the reproducer in this model; the string rewriting agent *(sra), which rewrites* `$0` *to the name of the script currently executing; and the cp command, which creates the copy of the virus. Therefore* $Ent = \{v, sra, cp\}$. *Since substitution for* `$0` *and copying file contents are basic functions of the Bash interpreter, we let sra and cp be present in all states — this is a reasonable choice for our simplified operational semantics; in a more detailed semantics, we might, for example, specify that cp is not present in certain "error" states arising from hardware or software failures that make the filestore unavailable. We further specify that v is present in all states where the command* `cp $0 $0.copy`*, or the result of substituting for* `$0` *in this, is present either in the filestore or as a command about to be executed by the interpreter.*

Thus far, we have defined a basic reproduction model; we make this an affordance-based model by specifying:

$$\begin{aligned} \mathit{Aff}(\mathit{subst}) &= \{\mathit{sra}\} \\ \mathit{Aff}(\mathit{cp}) &= \{\mathit{cp}\} \\ \mathit{Aff}(\mathit{get}) &= \{v\} \end{aligned}$$

It is readily checked that these equations satisfy the constraint of Definition 7, and that the result is a non-trivial assisted model.

Several variations on this model may be given by changing the definition of the function Aff. For example, if one feels that substitution for `$0` is a freely available resource that can be taken for granted, one may set $\mathit{Aff}(\mathit{subst}) = \emptyset$. Similarly, one may have $\mathit{Aff}(\mathit{cp}) = \emptyset$ if one feels that copying files may be taken for granted. Together, these two changes would give a model in which the entities *sra* and *cp* may be considered

surplus to requirements and dropped from the set *Ent*. This model would then be a non-trivial unassisted model.

3.4.2 Refinement of Reproduction Models

It should be clear from the preceding examples that we are not classifying reproducers *per se*; rather, we are classifying *models* of reproducers, and we allow for the possibility that a reproductive process may be modelled in many different ways. It is possible that this permissiveness might seem inappropriate. After all, it might be argued, the primary goal of a model is verisimilitude: things are one way or another, and the obligation on a model is to say which way things are; so if there are two different models of the same thing, then at least one model is wrong.

Our view, which may show a bias towards practices in Computer Science, is that it is often useful to allow different models of the same process, perhaps at different levels of abstraction, or reflecting different states of understanding of the process being modelled. In software engineering, for example, it is common to start with a very abstract specification of a system, and repeatedly refine this by adding more details and constraints, until a final, very concrete specification is reached. Each version of the specification can be seen as a model of the not-yet-realised system, at varying levels of abstraction. The important relationship between the different models is a form of consistency: the more concrete models impose more constraints on admissible behaviours, or every behaviour allowed by the concrete models is also allowed in the more abstract models. In this section we present a notion of refinement for affordance-based reproduction models that captures the idea that one model provides a more concrete view of the same process modelled by another. We will then show that it is possible to freely move between viewing a process as assisted or unassisted.

Consider the schematic model of a bacteriophage's life cycle presented in Section 3.4; the following gives a more concrete version of this life cycle.

Example 6. *We use terms to represent individual cells, bacteriophages, and bacteriophage RNA. For example, we use* `b-rna` *to name a particular bacteriophage RNA sequence, and write* `T4[b-rna]` *to denote an individual T4 bacteriophage with that sequence (we are not concerned with any specific mechanics of RNA reproduction in this model, so we need do no more than name the RNA here). Similarly, we write* `Cell[]` *for an individual cell, and we denote states where several individuals coexist by simply juxtaposing the terms for the individuals; thus, for example,*

```
Cell[] T4[b-rna] Cell[] T4[b-rna] T4[b-rna]
```

denotes a state containing two cells and three bacteriophages. We consider this state to be equivalent to any permutation of its constituent entities. Technically, we mean that juxtaposition is an associative and commutative operation; semantically, we think of this state as a "soup" in which the constituent entities can "move around" in order to interact with one another. We also allow a similar sort of soup to exist within a cell's membrane; for example,

```
Cell[ b-rna T4[b-rna] b-rna b-rna ]
```

denotes a single cell that contains three bacteriophage RNA strands, and one mature bacteriophage.

Such a situation can come to pass by a bacteriophage attaching to a cell and injecting its RNA. We write C-T for a cell C with attached bacteriophage T, and postulate two rewrite rules *that allow attachment and injection of RNA (we omit the labels of the actions):*

$$C\ T \longmapsto C\text{-}T$$
$$\texttt{Cell[}\ S\ \texttt{]-T4[}R\texttt{]} \longmapsto \texttt{Cell[}\ S\ R\ \texttt{]}$$

which say, respectively, that bacteriophage T can attach to cell C, and that when a bacteriophage with RNA R is attached to a cell that contains "internal soup" S, the RNA R can be injected into that internal soup. Similarly, we give rewrite rules that allow bacteriophage RNA to be replicated inside a cell, and that allow bacteriophage RNA to mature into T4 bacteriophages:

$$\texttt{Cell[}\ R\ S\ \texttt{]} \longmapsto \texttt{Cell[}\ R\ S\ R\ \texttt{]}$$
$$\texttt{Cell[b-rna }S\texttt{]} \longmapsto \texttt{Cell[}\ S\ \texttt{T4[brna]]}$$

Finally, in the life-cycle of the bacteriophage, we allow cells to rupture, releasing matured bacteriophages into the environment:

$$\texttt{Cell[}\ S\ \texttt{]} \longmapsto S$$

Clearly, these five rewrite rules correspond to the five schematic stages of the bacteriophage life-cycle. Moreover, we can see these stages applied to individuals, as in the following example:

```
 T4[b-rna] Cell[] T4[b-rna]
⟼
 T4[b-rna] Cell[]-T4[b-rna]
⟼
 T4[b-rna] Cell[ b-rna ]
```

```
⟼
 T4[b-rna] Cell[ b-rna b-rna ]
⟼
 T4[b-rna] Cell[ b-rna b-rna ]
⟼
 T4[b-rna] Cell[ T4[b-rna] b-rna]
⟼
 T4[b-rna] T4[b-rna] b-rna
```

which shows a T4 bacteriophage attaching to a cell, injecting its RNA, that RNA being copied, maturing, and then being released as the cell "ruptures" (albeit after minimal reproduction and maturing of the T4 RNA).

The states of this model are the terms of sort "soup", and the rewrite rules given above determine the actions and transitions. We can postulate entities comprising a cell, `Cell`*, which is present in a state iff that state has a subterm of the form* `Cell[...]`*, and bacteriophages, present in a given state iff the RNA,* `b-rna` *occurs as a subterm. Furthermore, we can specify that all the actions (attachment, injection, etc.) are afforded jointly by the cell and the bacteriophages (and hence this model is an assisted reproduction model).*

In order to relate this model (call it *T4Cell*) to the schematic model of Section 3.4, note first that we can map states of that model to states of *T4Cell* as follows[2]:

$$
\begin{aligned}
s_1 &\mapsto f(s_1) = \texttt{Cell[] T4[b-rna]} \\
s_2 &\mapsto f(s_2) = \texttt{Cell[]-T4[b-rna]} \\
s_3 &\mapsto f(s_3) = \texttt{Cell[b-rna]} \\
s_4 &\mapsto f(s_4) = \texttt{Cell[b-rna b-rna]} \\
s_5 &\mapsto f(s_5) = \texttt{Cell[b-rna T4[b-rna]]} \\
s_6 &\mapsto f(s_6) = \texttt{b-rna T4[b-rna]}
\end{aligned}
$$

This maps the path of the schematic model to the path of *T4Cell*; i.e., it preserves transitions, which have the same actions in both models. The schematic model had two entities, b and c, representing the bacteriophage and the cell, respectively. We can map these to entities $h(b) = \texttt{T4[b-rna]}$ and $h(c) = \texttt{Cell}$, respectively, noting that this preserves occurrences in states: if $e \ \varepsilon \ s$, then $h(e) \ \varepsilon \ f(s)$. Moreover, since all actions are afforded jointly by the cell and the bacteriophage in both models, affordances are

[2]For the interested reader, Maude specifications of the schematic model of Section 3.4, as well as the model *T4Cell*, can be found in Appendix C.

clearly preserved as well.

We generalise this example in

Definition 11. *For basic reproduction models* $M = (S_M, A_M, \longmapsto_M, Ent_M, r_M, \varepsilon_M, p_M)$ *and* $N = (S_N, A_N, \longmapsto_N, Ent_N, r_N, \varepsilon_N, p_N)$*, a* refinement $M \longrightarrow N$ *is a triple of functions* (f, g, h)*, where* $f : S_M \to S_N$*,* $g : A_M \to A_N$*, and* $h : Ent_M \to Ent_N$ *such that*

1. $s \overset{a}{\longmapsto}_M s'$ *implies* $f(s) \overset{g(a)}{\longmapsto}_N f(s')$*,*
2. $e \; \varepsilon_M \; s$ *implies* $h(e) \; \varepsilon_N \; f(s)$*, and*
3. $h(r_M) = r_N$*.*

Moreover, if M *and* N *are affordance-based models, then* $(f, g, h) : M \longrightarrow N$ *is a* refinement *iff, in addition,* $h(Aff_M(a)) \subseteq Aff_N(g(a))$ *for all actions* $a \in A_M$ *(note we write* $h(X)$ *for the set resulting from applying* h *to every element of the set* X*).*

Intuitively, a refinement $M \longrightarrow N$ indicates that M and N model the same process, but N provides a more detailed or concrete model, i.e., N refines M. Since transitions, occurrences and affordances are all preserved, all of the behaviour, entities, and affordances described in M also occurs in N, although N may provide more states, actions, and entities than figure in M.

Despite this intuitive image of a refinement as an inclusion of one model within another, there are many interesting refinements where one or more of the component functions is not injective. In particular, models may be refined by "amalgamating" two or more entities. We will see in the following section that such refinements can move freely between assisted and unassisted reproductive models.

Example 7. *We give a refinement of the copier computer virus model* M_v *from Example 5. Let* N_v *be as follows:*

- S_{N_v}*,* A_{N_v}*,* $\longmapsto_{N_v}$*,* $r_{N_v} (= v)$ *and* p_{N_v} *are identical to those in* M_v*;*
- $Ent_{N_v} = \{v, sra{+}cp\}$*;*
- $Aff_{N_v}(subst) = Aff_{N_v}(cp) = \{sra{+}cp\}$*; and*
- $sra{+}cp \; \varepsilon_{M'_v} \; s$ *iff* $sra \; \varepsilon_{M_v} \; s$ *or* $cp \; \varepsilon_{M_v} \; s$*.*

It is readily checked that the identity functions $1_{S_{M_v}} : S_{M_v} \to S_{M_v}$ *and* $1_{A_{M_v}} : A_{M_v} \to A_{M_v}$*, together with the function mapping* v *to* v *and both* sra *and* cp *to* $sra{+}cp$*, give a refinement* $M_v \longrightarrow N_v$*. Note that in this case, both* M_v *and* N_v *are assisted reproductive models.*

3.4.3 Allowed Refinements of Reproduction Models

The definitions of assistance and triviality given earlier are independent dichotomies, meaning that we can divide the space of reproduction models into four disjoint parts, depending on the assistance and triviality of a reproduction model. It is interesting to note that refinement between these four parts is limited in certain directions, which implies a structure of the space of reproduction models. Firstly, there are no refinements from non-trivial models to trivial models.

Proposition 6. *If M and N are models, and there is a refinement $M \longrightarrow N$, then M being non-trivial implies that N is non-trivial.*

Proof. If M is non-trivial then we know that $r_M \in \mathsf{E}(M)$. By Definition 11, we know that $h(r_M) = r_N$ and $h(\mathit{Aff}_M(a)) \subseteq \mathit{Aff}_N(g(a))$. Therefore $r_N \in \mathsf{E}(N)$ and N is non-trivial, as desired. □

Secondly, there are no refinements to trivial, unassisted models.

Proposition 7. *For all trivial, unassisted affordance based models, N, there is no refinement $M \longrightarrow N$, where M is trivial and assisted.*

Proof. Proof is by contradiction. Suppose that a refinement $M \longrightarrow N$ exists. Since M is assisted, then by Definition 9 there must be some entity $x \in \mathit{Ent}_M$, different from the reproducer r, such that $x \in \mathit{Aff}_M(b)$ for some action b in the path. By Definition 11 we know that $h(\mathit{Aff}_M(b)) \subseteq \mathit{Aff}_N(g(b))$ and therefore $h(x) \in \mathit{Aff}_N(g(b))$. However, $\mathit{Aff}_N(b') = \emptyset$ for all actions b' in the path, because N is trivial and unassisted. Therefore, there can be no such function h, and therefore the refinement cannot exist, as desired. □

It is straightforward to demonstrate that refinements are allowed in all other directions, and therefore these proofs are omitted. The resulting structure of the space of reproduction models is summarised in Figure 3.4.

3.5 The Unassisted and Assisted Reproduction Theorems

In this section we show that the classification into assisted or unassisted models does not reflect an intrinsic property of the process being modelled, but rather reflects the decisions made in constructing a particular model. We show that every model can be refined by an unassisted model, and, dually, every model refines an assisted model.

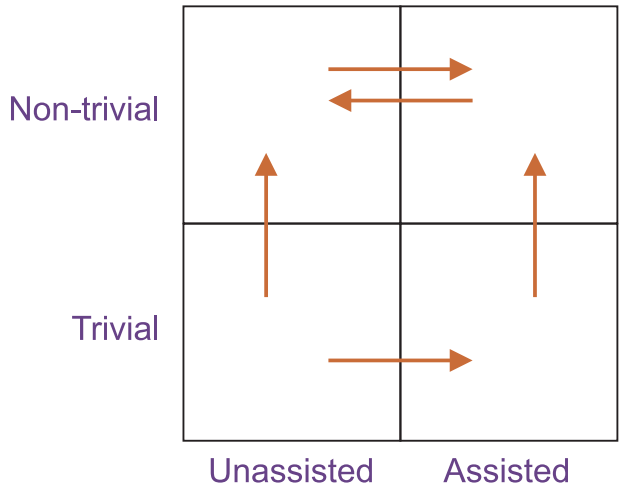


Figure 3.4: Allowed refinements between classes of affordance-based models.

3.5.1 The Unassisted Reproduction Theorem

One potential application of refinement is to identify the entities that assist the reproducer in the act of reproduction, and conglomerate them with the reproducer to form a "super-entity". Of course, the resulting aggrandised reproduction model will then be an unassisted model irrespective of whether the original reproduction model was assisted or unassisted. In the following Propositions 8–11, and culminating in Theorem 2, we will show how for any reproduction model, M, there is a corresponding unassisted reproduction model $M^{\#}$ — in which states, actions, action successions, and the reproducer's path remain unchanged — for which there is a refinement arrow $M \longrightarrow M^{\#}$. The first step is to define $M^{\#}$ in

Definition 12. *Given a reproduction model* $M = (S, A, \longmapsto, Ent, r, \varepsilon, p, Aff)$*, we define*

$$M^{\#} = (S, A, \longmapsto, Ent^{\#}, r, \varepsilon^{\#}, p, Aff^{\#})$$

where

1. $Ent^{\#} = (Ent \backslash \mathsf{E}(M)) \cup \{r\}$*;*
2. $r\ \varepsilon^{\#}\ s$ *iff* $e\ \varepsilon\ s$*, for some entity* $e \in \mathsf{E}(M) \cup \{r\}$*; and for all* $e \in Ent \backslash \mathsf{E}(M)$*,* $e\ \varepsilon^{\#}\ s$ *iff* $e\ \varepsilon\ s$*; and*
3. $Aff^{\#}(a) = h(Aff(a))$*, where* $h : Ent \to Ent^{\#}$ *maps* $c \in \mathsf{E}(M)$ *to* r*, otherwise* $h(e) = e$*.*

Proposition 8. *For any model* M*,* $M^{\#}$ *is an affordance-based reproduction model.*

Proof. By Definition 7, we require that for all $e \in \mathit{Aff}^{\#}(a)$, and for all states s where a is possible in s then $e \; \varepsilon^{\#} \; s$. Suppose that action a is possible in state s, and $e \in \mathit{Aff}^{\#}(a)$. By Definition 12(3), $e = h(e_0)$ for some $e_0 \in \mathit{Ent}$, and because M is an affordance-based reproduction model, it follows that $e_0 \; \varepsilon \; s$. If $e_0 \in \mathsf{E}(M)$, then $e = h(e_0) = r$ and $e \; \varepsilon^{\#} \; s$ by Definition 12(2); if $e_0 \notin \mathsf{E}(M)$, then $e = h(e_0) = e_0 \; \varepsilon \; s$ and so $e \; \varepsilon^{\#} \; s$ as desired. □

Now that we have established that both M and $M^{\#}$ are models, we must check that $M^{\#}$ is in fact unassisted.

Proposition 9. *For any model M, $M^{\#}$ is unassisted.*

Proof. The only entities which afford reproductive actions (i.e., those in p) to r in M are those in $\mathsf{E}(M)$. Therefore, for any a_i in p, if $e \in \mathit{Aff}^{\#}(a_i)$, then $e = h(c)$ for some $c \in \mathsf{E}(M)$ and so $e = r$. □

Next we show that $M^{\#}$ refines M.

Proposition 10. *For all models M, there is a refinement $M \longrightarrow M^{\#}$.*

Proof. The refinement $M \longrightarrow M^{\#}$ consists of the triple $(1_S, 1_A, h)$, where 1_S and 1_A are identities on states and actions, and $h : \mathit{Ent} \to \mathit{Ent}^{\#}$ is as defined in Definition 12. Clearly, transitions are preserved (Definition 11(1)), and preservation of occurrences (Definition 11(2)) follows immediately from Definition 12(2), and we need show only $h(\mathit{Aff}(a)) \subseteq \mathit{Aff}^{\#}(a)$ for all actions a, but this is immediate from Definition 12(3). □

This gives us our main result for this section:

Theorem 2 (Unassisted Reproduction Theorem)**.** *Every reproduction model can be refined by an unassisted reproduction model.*

In other words, for any reproduction model, be it assisted or not, there is another model which captures the same reproductive process but with modified entities, which is classified as unassisted. Therefore, all models of assisted reproduction can be refined so that they instead capture unassisted reproduction. This tells us that apparent assisted reproduction is simply a consequence of the way a particular reproductive process is modelled, since it can also be modelled as unassisted reproduction.

In order to demonstrate Theorem 2, we give an example of a model $M_v^{\#}$ that is a refinement of the copier computer virus model M_v (cf. Examples 5 and 7), and is an unassisted model.

Example 8. *Let $M_v^{\#}$ be constructed from the model M_v of Example 5, as in Definition 12. This gives us:*

- $S_{M_v^\#}$, $A_{M_v^\#}$, $\longmapsto_{M_v^\#}$, $r_{M_v^\#}(= v)$ *and* $p_{M_v^\#}$ *are identical to those in* M_v*;*
- $Ent_{M_v^\#} = \{v\}$*;*
- $v\ \varepsilon_{M_v^\#}\ s$ *for all* $s \in S$*; and*
- $Aff_{M_v^\#}(a) = \{v\}$ *for all actions* a*.*

The refinement $M_v \longrightarrow M_v^\#$ *consists of the identity functions on states and actions, and the function that maps all entities to* v*.*

If a model M is an unassisted model, then our construction of $M^\#$ just yields the original model M:

Proposition 11. *If* M *is unassisted, then* $M = M^\#$*.*

This is proved by inspection of Definition 12, noting that if M is unassisted, then $\mathsf{E}(M) \subseteq \{r\}$. A slightly stronger statement says that the construction of $M^\#$ is the smallest change to M that is needed to obtain an unassisted reproduction model. Consider the situation in Figure 3.5, where the arrow along the top is the refinement of Proposition 10. If there is some other unassisted model N that refines M, then it makes

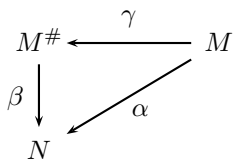


Figure 3.5: Refinement arrows between M, $M^\#$ and N.

a larger change than $M^\#$ does, and it refines $M^\#$ as well; moreover, it does so in a unique way. This is stated formally in the following

Proposition 12. *For all reproduction models* M*,* $M^\#$ *is the least unassisted refinement of* M*.*

Proof. Suppose $(\alpha_f, \alpha_g, \alpha_h)$ is a refinement of M by an unassisted reproduction model $N = (S_N, A_N, \longmapsto_N, Ent_N, r_N, \varepsilon_N, p_N, Aff_N)$.

We show that there is a unique refinement $\beta = (\beta_f, \beta_g, \beta_h) : M^\# \to N$ such that α is the composition of γ and β, where $\gamma = (1_S, 1_A, h)$ is the refinement $M \longrightarrow M^\#$ of Proposition 10. Since γ is the identity on states and actions, we clearly require $\beta_f = \alpha_f$ and $\beta_g = \alpha_g$, and all that remains is to define β_h. By the definition of refinement, we require $\beta_h(r_{M^\#}) = r_N$, and for all other $e \in Ent^\#$ (i.e., $e \in Ent \backslash \mathsf{E}(M)$), we set $\beta_h(e) = \alpha_h(e)$. This clearly satisfies the uniqueness constraint on β, and we

need only show that β does indeed exist; i.e., we need to show that $\alpha_h(e) = r_N$ for all $e \in \mathsf{E}(M)$. If $e \in \mathsf{E}(M)$, then there is some a_i in p_M with $e \in \mathit{Aff}_M(a_i)$, and so $\alpha_h(e) \in \mathit{Aff}_N(\alpha_g(a_i))$, but since N is unassisted, this must mean that $\alpha_h(e) = r_N$, as desired. □

3.5.2 The Assisted Reproduction Theorem

From Theorem 2 we know that all reproduction models can be refined by an unassisted reproduction model, and therefore all reproduction models can be viewed as unassisted. In this subsection we show that the converse is also possible, that for all reproduction models, there is a corresponding assisted reproduction model. We define this corresponding model in

Definition 13. *Given a reproduction model,* $M = (S, A, \longmapsto, \mathit{Ent}, r, \varepsilon, p, \mathit{Aff})$*, we define*

$$M_{\#} = (S, A, \longmapsto, \mathit{Ent}_{\#}, r, \varepsilon_{\#}, p, \mathit{Aff}_{\#})$$

where

- $\mathit{Ent}_{\#} = \mathit{Ent} \cup \{G\}$*;*
- $\mathit{Aff}_{\#}(a) = \mathit{Aff}(a)$ *if* $r \notin \mathit{Aff}(a)$*, and* $\mathit{Aff}_{\#}(a) = \mathit{Aff}(a) \setminus \{r\} \cup \{G\}$ *if* $r \in \mathit{Aff}_M(a)$*;*
- *for all states* s*,* $G \; \varepsilon_{\#} \; s$ *iff* $r \; \varepsilon \; s$*;*
- $e \; \varepsilon_{\#} \; s$ *iff* $e \; \varepsilon \; s$ *for all entities* $e \neq G$*.*

The model $M_{\#}$ introduces a new entity, G, and ascribes to it all the actions afforded by r in M. We might think of this new entity as the "Laws of Nature", which makes possible all of the actions afforded by the reproducer (in M); such a change of viewpoint might be seen in viewing, on the one hand, organisms as actively reproducing through their own actions, and viewing them, on the other hand, merely as phenotypes of genes that persist, or not, under the action of natural selection. Another example of such a change of viewpoint would be to view reproducers such as Langton's loops [89] as, on the one hand, reproducing by means of extending a process that loops back on itself, or, on the other hand, as just phenomena that emerge from the iterative application of the evolution rule of the cellular-automata grid in which the loops are realised. In this example, the entity G is the evolution rule; in the previous example, it is natural selection. We might say that the Unassisted Reproduction Theorem of the previous section represents an ecological view of reproduction, in which a number of separate entities that collaborate in a process of reproduction can be viewed as a entity in itself, whereas the construction of this section represents a reductionistic approach, in which

any behaviour (though especially reproduction for our purposes) can be viewed as a manifestation of physical or computational laws.

Proposition 13. *For all reproduction models M, $M_\#$ is a reproduction model and there is a refinement $M_\# \longrightarrow M$.*

Proof. It is clear from the construction that $M_\#$ is an affordance-based reproduction model whenever M is. The refinement $M_\# \longrightarrow M$ consists of the triple $(1_S, 1_A, h)$, where 1_S and 1_A are identities, and h is defined as follows: $h(G) = r$, and $h(e) = e$ for $e \in \mathit{Ent}$. We must check that the conditions from Definition 11 hold. Condition (1) holds trivially, because S, A and $\longmapsto$ are identical in M and $M_\#$. Condition (2) holds by construction of $M_\#$, as does condition (3). The final condition, that $h(\mathit{Aff}_\#(a)) \subseteq \mathit{Aff}(a)$, holds because $\mathit{Aff}_\#$ replaces r by G and h maps G to r. □

The Assisted Reproduction Theorem follows from this, with one proviso: that the original model M is *non-trivial* (cf. Definition 7).

Theorem 3 (Assisted Reproduction Theorem)**.** *Every non-trivial reproduction model M refines an assisted reproduction model.*

Proof. This follows directly from Proposition 13, noting that if $r \in \mathit{Aff}(a_i)$, then $\mathit{Aff}_\#(a_i) = \mathit{Aff}(a_i) \setminus \{r\} \cup \{G\}$, so $r \neq G \in \mathit{Aff}_\#(a_i)$, and therefore $M_\#$ is an assisted model. □

The requirement that M be non-trivial is from Definition 13, as $M_\#$ is assisted if M is non-trivial.

We illustrate this construction by revisiting the copier computer virus M_v of Example 5, an assisted reproducer that was rendered unassisted in Example 8, giving the model $M_v^\#$. We now apply the Assisted Reproduction Theorem to give

Example 9. *Let $(M_v^\#)_\#$ be constructed from $M_v^\#$ following Definition 13. We have*

- $\mathit{Ent}_{(M_v^\#)_\#} = \{v, G\}$;
- $G \; \varepsilon_{(M_v^\#)_\#} \; s$ *for all* $s \in S$; *and*
- $\mathit{Aff}_{(M_v^\#)_\#}(a) = \{G\}$ *for all actions* a.

In other words, G is omnipresent and omnipotent in that it alone affords all the actions of the copier virus's reproductive cycle.

These examples show that our approach does not say that assisted and unassisted models are the same thing: the constructions of Definitions 12 and 13 are not bijections,

as M_v and $M_\#$ are different. What our approach *does* say is that a reproductive process may be viewed qualitatively in different ways. Moreover, the notion of refinement serves to rank these qualitatively different approaches to modelling the same process: ecological approaches are more refined than reductionistic approaches, or, following Example 6, more *schematic*.

3.5.3 Further Classification Using Aspects

In Section 3.2, we classified reproducers into four categories, which we called Types I, II, III, and IV. These categories were based on identifying which actions in the model were concerned with obtaining a self-description (SD) of the reproducer, and which actions were concerned with the reproductive machinery (RM) for constructing a copy of the reproducer. Models where all actions in both groups were unassisted were categorised as Type I, which in Section 3.4 we simply called "unassisted"; the remaining types subdivide the assisted class of models. The case where SD was unassisted and RM was assisted we called Type II; the case where SD was assisted and RM was unassisted we called Type III; and the case where both were assisted we called Type IV.

As we argued in Section 3.2, a self-description and its use in the construction of a copy of the reproducer plays a central role in a great number of reproduction models. However, there may be applications where other aspects of the reproductive process play a more prominent role. For example, many computer viruses contain code that is designed to prevent the infection of already-infected files, or code that is intended to avoid detection by — or even to actively attack — anti-virus software, and the ecology or dependencies of these code fragments are therefore of interest to the producers of anti-virus software. As another example, in biological organisms, sexual reproduction, random mutations in DNA, and the interactions of the organism itself with its environment, including, for example, predators or potential mates, are the aspects of the reproductive cycle where natural selection plays a role: it would be hard to explain a peacock's tail without a notion of mate-selection. In this section we shall generalise this mode of classification to arbitrary predicates on the actions in the reproduction model, which we call *aspects*.

Formally, an aspect is just a name, such as "self-description", "detection-avoidance", or "mate-selection". As such, it has no intrinsic formal meaning, and its application to reproduction models depends on the intentions of the modeller. It applies to a particular model as a predicate on the actions of that model, saying which actions are concerned with that aspect. For example, the actions of the T4 bacteriophage life-cycle (see Section 3.4) that are concerned with reproductive machinery would be synthesis of the bacteriophage RNA and its maturation. The actions of the copier computer virus

(see Example 5) concerned with the same aspect would be the call of the `cp` function; the actions in this model concerned with obtaining a self-description would be the substitution of the virus's file name for the variable `$0` — but again, there is freedom for interpretation on the part of the modeller: it would be perfectly acceptable to also include the call of `cp` as concerned with obtaining a self-description, which would be reasonable, as this is where the actual viral code is accessed in the file store.

Of course, we may be interested in studying reproduction models with regard to a particular aspect, or with regard to several.

Definition 14. *Given an aspect α, an α-model consists of an affordance-based model M together with a predicate M_α on the actions of M. If C is a set of aspects, a C-model is a model with a predicate M_α for each $\alpha \in C$.*

We can relativise the model theory of the preceding sections to aspects or sets of aspects.

Definition 15. *For aspect α and α-model M, we define the* α-ecology of M, $\mathsf{E}_\alpha(M)$, *to be the union of all the sets $Aff_M(a_i)$ for a_i in the path of M for which M_α holds: i.e., all entities that afford α-actions in M.*

Definition 16. *We say M is* α-unassisted *iff $\mathsf{E}_\alpha(M) \subseteq \{r_M\}$, and M is* α-assisted *iff $\mathsf{E}_\alpha(M)$ contains some entity other than r_M. Similarly, M is C-unassisted iff it is α-unassisted for each $\alpha \in C$, and C-assisted iff it is α-assisted for some $\alpha \in C$.*

Moreover, it is clear that the constructions of the Assisted and Unassisted Reproduction Theorems can be relativised to arbitrary sets of aspects. Thus, for any α-model M, there is an α-unassisted model that refines M, and (for non-trivial M) an α-assisted model that is refined by M.

Any unassisted model will be α-unassisted, for any aspect α, and in this sense, aspects serve to subdivide the space of assisted models. The basic distinction between assisted and unassisted could be viewed as arising from an aspect that holds for all actions in a reproduction model's path, but in general, n aspects give 2^n categories of assisted reproduction. For example the four-fold categorisation (Types I–IV) described above arises from the two aspects of self-description and reproductive-machinery. Conversely, "enough" aspects serve to recapture the absolute notion of "unassisted":

Proposition 14. *If M is C-unassisted and the set of actions in p_M is covered by the aspects in C, i.e., each a_i in p_M satisfies some M_α for $\alpha \in C$, then M is unassisted.*

Proof. For any a_i in the path of M, there is at least one aspect $\alpha \in C$ for which M_α holds, and since M is α-unassisted, $Aff_M(a_i) \subseteq \{r_M\}$, and so $\mathsf{E}(M) \subseteq \{r_M\}$. □

In summary, aspects provide a generalisation of our basic distinction between assisted and unassisted models of reproduction, and allows finer-grained distinctions between assisted models.

3.6 Further Examples

In Section 3.4 we showed the applicability of affordance-based reproduction models to biological life forms (e.g., bacteriophage viruses) and computer viruses. In Chapter 4 we will explore the latter in much more detail. However it is useful to give explicit details about the application of affordance-based reproduction models to artificial life forms, e.g., the paradigmatic examples of Langton's loop, or the case of gliders in Conway's Game of Life. In the examples below we will demonstrate how models of artificial life forms at different levels of abstraction can be related using refinements, and how applications of the Unassisted and Assisted Theorems might aid in our understanding of artificial life.

3.6.1 Langton's Loop

Langton's loops reproduce on a two-dimensional cellular automaton grid. A loop consists of an outer "sheath" which contains the self-description: a series of symbols encoded in the states of the sheathed cells. The self-description causes an "arm" to be extended from one corner of the loop, which then turns perpendicularly, before repeating the process a further three times until a child loop is constructed after 151 time steps [89], as in Figure 3.1(a).

There are a number of ways in which the reproductive process of the loop could be modelled. For example, we could define the the labelled transition system so that it would model explicitly the states of the cellular automaton grid, e.g.,

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} \ldots \overset{a_{151}}{\longmapsto} s_{152}, \tag{3.2}$$

or we could define a more abstract labelled transition system modelling abstract actions which correspond to the **b**eginning, **m**iddle and **e**nd of the act of reproduction (see Figure 3.6).

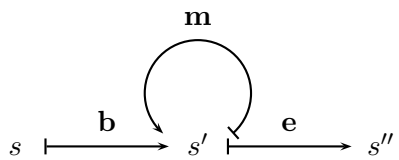


Figure 3.6: A possible labelled transition system for a model of Langton's loop.

In the next example we will show that there is a refinement of an affordance-based model with Equation 3.2 as its labelled transition system to another affordance-based model with Figure 3.6 as its labelled transition system.

Example 10. *Let L be a model in which S_L, A_L and $\longmapsto_L$ and the path p_L are defined by Figure 3.6. Let the set of entities $Ent_L = \{L, t\}$ in which L is Langton's loop (and also the reproducer in this model) and t is the transition rule of the cellular automaton that affords reproductive actions to L, i.e., $Aff_L(a) = \{L, t\}$ for all actions a. We know that $L \; \varepsilon_L \; s$ and $L \; \varepsilon_L \; s''$ by Definition 6, and $t \; \varepsilon_L \; s$ and $t \; \varepsilon_L \; s'$ by Definition 7.*

Then, let L' be a model in which $S_{L'}$, $A_{L'}$ and $\longmapsto_{L'}$ and the path $p_{L'}$ are defined by (3.2). Let the set of entities $Ent_{L'} = Ent_L$, and let all reproductive actions be afforded by the transition rule, i.e., $Aff_{L'}(a) = \{L, t\}$ for all actions a. We know that $L \; \varepsilon_{L'} \; s_1$ and $L \; \varepsilon_{L'} \; s_{152}$ by Definition 6, and we set $t \; \varepsilon_{L'} \; s_i$ for $1 \leq i \leq 151$.

Now we will define three functions refining L' to L. Let $f : S_{L'} \to S_L$ be defined as follows:

$$\begin{aligned} f(s_1) &= s, \\ f(s_i) &= s' \text{ for } 2 \leq i \leq 151, \text{ and} \\ f(s_{152}) &= s''. \end{aligned}$$

Let $g : A'_L \to A_L$ be defined as follows:

$$\begin{aligned} g(a_1) &= \mathbf{b}, \\ g(a_i) &= \mathbf{m} \text{ for } 2 \leq i \leq 150, \text{ and} \\ g(a_{151}) &= \mathbf{e}. \end{aligned}$$

Since we are specifying a refinement relating two models that differ only at their abstraction level, we set $h : Ent_{L'} \to Ent_L$ as an identity function.

In order to check that there is a refinement $L' \longrightarrow L$, we must check that the conditions from Definition 11 hold. The first condition, that $s \overset{a}{\longmapsto}_{L'} s'$ implies $f(s) \overset{g(a)}{\longmapsto}_L f(s')$ is readily checked exhaustively:

$$\begin{aligned} s_1 \overset{a_1}{\longmapsto}_{L'} s_2 &\Rightarrow s \overset{\mathbf{b}}{\longmapsto}_L s' \\ s_2 \overset{a_2}{\longmapsto}_{L'} s_3 &\Rightarrow s' \overset{\mathbf{m}}{\longmapsto}_L s' \\ &\vdots \\ s_{150} \overset{a_{150}}{\longmapsto}_{L'} s_{151} &\Rightarrow s' \overset{\mathbf{m}}{\longmapsto}_L s' \\ s_{151} \overset{a_{151}}{\longmapsto}_{L'} s_{152} &\Rightarrow s' \overset{\mathbf{e}}{\longmapsto}_L s'' \end{aligned}$$

Definition 11(2), that $e\ \varepsilon_{L'}\ s$ implies $h(e)\ \varepsilon_L\ f(s)$ is also easily checked. Since h is an identity function, we need only show that the following are true:

$$\begin{aligned} L\ \varepsilon_{L'}\ s_1 &\Rightarrow L\ \varepsilon_L\ s \\ L\ \varepsilon_{L'}\ s_{152} &\Rightarrow L\ \varepsilon_L\ s'' \\ t\ \varepsilon_{L'}\ s_1 &\Rightarrow t\ \varepsilon_L\ s \\ t\ \varepsilon_{L'}\ s_i &\Rightarrow t\ \varepsilon_L\ s' \text{ for } 2 \leq i \leq 151 \end{aligned}$$

It is simple to show that Definition 11(3) holds, as h is an identity function. Finally, the condition on affordance-based models that $h(Aff_{L'}(a)) \subseteq Aff_L(g(a))$ for all actions $a \in A_{L'}$ also holds, since h is an identity function and $Aff_{L'}(a) = Aff_L(a') = \{L, t\}$ for all actions $a \in A_{L'}$ and $a' \in A_L$.

Therefore $L' \longrightarrow L$ for the functions (f, g, h) defined above.

We know by Definition 9 that L is an assisted reproduction model, since there is an action a in the path p_L for which there is an entity other than L in $Aff_L(a)$. In the next example we will show an application of the Unassisted Reproduction Theorem, in which we construct a model $L^{\#}$ that is a refinement of L but is classified as unassisted.

Example 11. *We begin by constructing a tuple $L^{\#}$ from L according to the construction of $M^{\#}$ from M in Definition 12. Therefore $Ent_L^{\#} = Ent_L \setminus \mathsf{E}(L) \cup \{L\} = \{L\}$ since $\mathsf{E}(L) = \{L, t\}$ and L is the reproducer in model L. Since $L\ \varepsilon_L\ s$, $L\ \varepsilon_L\ s''$, $t\ \varepsilon_L\ s$, $t\ \varepsilon_L\ s'$ and $r, t \in \mathsf{E}(L)$, by Definition 12 we know that $L\ \varepsilon_L^{\#}\ s$, $L\ \varepsilon_L^{\#}\ s'$ and $L\ \varepsilon_L^{\#}\ s''$. Also, by Definition 12 we know that $Aff_L^{\#}(a) = \{L\}$ for all actions $a \in A_L^{\#}$, since $Aff_L(a) = \{L, t\}$ and $h(L) = h(t) = L$. By Theorem 2 we know that $L^{\#}$ is an unassisted reproduction model, and that there is a refinement $L \longrightarrow L^{\#}$.*

Looking at the definition of $L^{\#}$ above we can see that the Unassisted Reproduction Theorem allows for the refinement of L to a reproduction model that is classified as unassisted, through effectively conglomerating the entities L and t into a single entity (also named L) in $L^{\#}$ that is present in all states in which L and t were present, and affords all actions that were afforded by L or t. Therefore the refinements given by the Unassisted Reproduction Theorem effectively model the shift in observation of a reproduction system from being assisted to being unassisted. By the Theorem, all reproduction models have an unassisted refinement, which tells us that any model with an assisted classification can be recast as an unassisted reproduction model through the process described in Definition 12.

3.6.2 Conway's Game of Life Gliders

Conway's Game of Life, also known as "Life", is a cellular automaton with a much simpler transition rule than the cellular automaton for Langton's loop [51]. Here we will present an affordance-based model of a "glider" in Conway's Game of Life. The glider is a pattern within Life that reproduces over four steps (see Figure 3.1(c)), and is a simple example of artificial life.

In Example 10 we constructed two different affordance-based reproduction models of Langton's loops, one of which modelled the states of the reproductive as being the states of the cellular automaton grid, and the other modelled the reproductive path as much more schematic and abstract, with three actions corresponding to the beginning, the middle and the end of the act of reproduction. The glider is much simpler however, and whilst a schematic model would be possible, it would be only slightly less abstract than a concrete model in which the states of the model are the states of the cellular automaton grid. We construct this concrete model as follows.

Example 12. *Let H be an affordance-based model in which the states $s_1, \ldots, s_5$ are the five states of the cellular automaton grid corresponding to the reproduction process (see Figure 3.1(c)), and the actions $a_1, \ldots, a_4$ are the transition rule applications which update these states, so that the labelled transition system $(S_H, A_H, \longmapsto_H)$ and path p_H are given by*

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} \ldots \overset{a_4}{\longmapsto} s_5 \ .$$

The only entity present is the glider g which reproduces, and therefore $Ent_H = \{g\}$ and $r_H = g$. We define every action in the reproduction process to be afforded by the glider g, and so $Aff_H(a) = \{g\}$ for all actions a in the path p_H. By Definition 7 we know that $g \ \varepsilon_H \ s_i$ for $1 \leq i \leq 4$, and by Definition 6 we know that $g \ \varepsilon_H \ s_5$.

In Example 11 we demonstrated an application of the Unassisted Reproduction Theorem to an affordance-based model of Langton's loops. We wil now do the same for the Assisted Reproduction Theorem and the Game of Life glider.

Example 13. *We begin by constructing a tuple $H_\#$ from H according to the construction of $M_\#$ from M in Definition 13. Let $S_{H_\#} = S_H$, $A_{H_\#} = A_H$ and $\longmapsto_{H_\#} = \longmapsto_H$. Then, $Ent_{H_\#} = Ent_H \cup \{G\} = \{g, G\}$, and since $Aff_H(a) = \{g\}$ for all actions a in the path p_H we know that $Aff_{H_\#}(a) = \{G\}$ for all actions a in the path $p_{H_\#}$. Similarly, $G \ \varepsilon_{H_\#} \ s$ for all states $s \in S_{H_\#}$ since $G \ \varepsilon_H \ s$ for all states $s \in S_H$. By the Assisted Reproduction Theorem we know that $H_\#$ is an assisted reproduction model, and that there is a refinement $H_\# \longrightarrow H$.*

It is interesting to note that in the first affordance-based model H, the glider g is

the only entity present, and the only entity that affords actions in the reproductive process. However, the model $H_{\#}$ contains two entities, one of which (G) takes over the responsibilities of the reproducer, g. We can think of this new entity, G, as being a "god-like" entity that takes responsibility for each action in the path afforded by the reproducer. For example, in the context of cellular automata such as the Game of Life, we can think of G as being the transition rule, and the refinement relationship $H_{\#} \longrightarrow H$ as a recognition of the different views of the glider's reproduction: one in which the transition rule plays a part, and one in which it does not.

3.7 Summary

We started the chapter with an informal classification of reproducers based on notions of self-description and reproductive mechanism. Whilst this approach is useful[3], it appeared that there would be benefits from a full formal treatment of the classification. In order to highlight this, we posed a number of questions concerning the inadequacy of the informal classification in Section 3.2.5. The first two questions, which were conjectures based on assisted and unassisted reproduction have been proven in the forms of the Assisted and Unassisted Reproduction Theorems respectively. The third question, on whether there is a way to model varying degrees of assistance, has been answered in the formal definition of reproduction models, which permit different actions to be attributed to different entities. The last question, on means of classifying reproduction other than the self-description and reproduction mechanism, has been answered in the formal definition of aspects, which allow us to define arbitrary predicates in order to classify reproduction models.

In order to answer these questions, we have given the first full formal definition of affordance-based reproduction modelling, classification and refinement. We have shown that we can specify reproduction systems using models in which discrete-time processes are modelled using labelled transition systems. We can define the entities present in a model, and specify in which states they are present. When entities are present in a state, they can afford actions which move that state to another state. We can divide these models by classification as "unassisted" or "assisted" reproduction, the former describing a state of affairs where no actions in the reproducer's path are afforded by entities other than the reproducer, and the latter describing there is at least one action in the path that is afforded by an entity other than the reproducer.

We have proven the Unassisted Reproduction Theorem, which states that for every reproduction model M, there is another model $M^{\#}$ that is a refinement of M, and

[3]For instance, source code viruses can be distinguished from more conventional viruses using this approach (see Section 3.2.2.3).

therefore preserves the structure of the reproduction model described, but guarantees classification of $M^{\#}$ as an unassisted reproduction model. We have also given the converse to this, the Assisted Reproduction Theorem, which shows that for all non-trivial reproduction models M there is another model $M_{\#}$ of which M is a refinement. These theorems show essentially that any reproduction system specified using a reproduction model, which is initially classified as assisted or unassisted, can be essentially reclassified as the other through refinement.

In Section 3.2 we described how reproduction models could be classified according to two separate dichotomies based on whether the actions corresponding to the reproductive mechanism and self-description of a reproductive process were afforded by entities other than the reproducer, or not. In Section 3.5.3 we generalised classifications of this sort using aspects: predicates on the actions in a reproduction model. An aspect, once applied to a reproduction model, holds if and only if no external entities afford any actions for which the aspect is true. In this way, a number n of aspects gives up to 2^n different classes of reproduction models.

We have shown how these reproduction models can be defined for both computer viruses and biological viruses such as bacteriophages, and then classified and refined. Biological viruses are interesting phenomena at the boundary of accepted definitions of life [148], and computer viruses are a form of artificial life [135], and therefore we have shown the applicability of our approach to real-life examples of reproduction.

In Section 3.6 we demonstrated that affordance-based reproduction models can be applied to the modelling of reproduction in artificial life; that models of artificial life forms at different levels of abstraction can be related using refinements of models; that the conglomeration of entities can be related using an application of the Unassisted Reproduction Theorem; and that the inclusion of an entity that represents the underlying "laws" of the universe, e.g., a cellular automaton transition rule, is accurately modelled by an application of the Assisted Reproduction Theorem.

In the following section we give an overview of related work in the literature, including formal and informal reproduction models and classifications. We follow this with a comparison of the related work with our approach (as a critical appraisal of the novel contribution presented in this chapter), and further discussions and conclusions drawn from this contextual analysis.

3.7.1 Related Work

Reproduction is one of the fundamental characteristics of most definitions of life, and as a result there is an extraordinary wealth of information in the literature concerning reproduction. Indeed, reproduction is a central concept in biology, artificial life, com-

puter virology and kinematic reproducer engineering [50], and has been influential in fields as diverse as psychology [36] and economics [99]. Therefore, a formal model of reproduction such as the one presented in this chapter could be applicable and relevant to any of the aforementioned fields.

Many formal studies of reproductive systems already exist. There are examples within theoretical biology (see, e.g., [141, 110]), which is concerned primarily with the study of life as we find it, largely through the development of formal mathematical models of existing biological systems. However, the focus on biological life reduces the applicability to life and reproduction in general.

The work presented in this chapter differs from the above, however. Formal affordance-based reproduction models are defined with the aim of providing a class of reproduction models that can be related through refinements, and classified in a formal manner, e.g., as unassisted or assisted, trivial or non-trivial.

The work is sufficiently abstract that it can be readily applied to examples of reproduction from biology, computer virology and artificial life. However, the focus is not on reproduction as we find it (i.e., organic life), but rather on reproduction in general. This fundamental difference also distinguishes the fields of artificial life and biology, and therefore the approach to formal modelling of reproduction presented in this chapter is informed by the principles of the study of artificial life. Therefore the work in this chapter is most closely related to formal notions of reproduction in the field of artificial life, and it is these notions that we will focus on in this section.

3.7.1.1 Löfgren's Approach to Modelling Reproduction

Löfgren presents an axiomatisation of reproduction based on a comparison between explanation and reproduction [92]. The author explains that to fully understand a system or concept, that is, to be able to explain it, requires sufficient knowledge to reproduce that system or concept. For example, students in a university are tested on their understanding of a subject through their ability to reproduce details of the subject under exam conditions. For example, we could define π_2 as a function that explains (reproduces) entity π_1 such that $\pi_2(\pi_1) =< \alpha, \beta, \ldots, \pi_1 >$. When interpreted as a reproduction process, this equation can be read as "π_2 models the function of an automaton that produces a sequence of outputs $\alpha, \beta, \ldots$ culminating in the reproduction of π_1". When interpreted as an explanation process, this equation can be read as "π_2 is a function that produces a proof sequence $\alpha, \beta, \ldots$, ending with a proven theorem π_1". Therefore, explanations and reproduction are related.

Löfgren then presents two characterisations of different types of reproducer:

- A *symbiotically self-reproducing* pair of distinct entities π_1 and π_2 must have a

complete explicability chain of length 2, such that

$$\begin{aligned} \pi_1(\pi_2) &= <\ldots, \pi_2> \\ \pi_2(\pi_1) &= <\ldots, \pi_1> \end{aligned}$$

The length of the explicability chain increases if we have more than a pair of entities in the symbiosis.

- An *atomically self-reproducing* entity π must be a unit-length complete explicability chain, such that

$$\pi(\pi) =< \pi > . \tag{3.3}$$

Löfgren states that "atomic self-reproduction shall result from the symbiotic self-reproduction when all the distinct entities of the symbiotic case coalesce", that Equation 3.3 implies the existence of a function which is in its own domain and range, and that Wittgenstein argued [164] that function cannot be its own argument, and that Rosen argued [122] that a function cannot be a member of its range. Löfgren supports this idea with a proof that reproduction of this type is inconsistent with an ordinary set theory, namely that defined by von Neumann, Bernays and Gödel [10]. However, as the author proves, an axiomatisation of atomic self-reproduction is possible, and such functions do exist in this sense.

In later work, Löfgren proposed an axiomatic explanation of reproduction, based on formal definitions of "productive", "reproductive" and "self-reproductive" [93]:

- An object A is *productive* in a surrounding S if the configuration d of A forces the surroundings to produce some object B, denoted $A \rightarrow d \rightarrow_S B$.

- An object A is *reproductive* in a surrounding S if there are objects A_i with descriptions d_i such that $A \rightarrow d_1 \rightarrow A_1$ and $A_i \rightarrow d_{i+1} \rightarrow_S A_{i+1}$ for $i > 0$, i.e., A forces S to produce a reproductive object.

- An object A is *self-reproductive* in a surrounding S if A produces a copy of itself in S.

The importance of Löfgren's work is, therefore, in demonstrating that while some formal definitions of reproduction lead to contradictions [164, 122] (especially within well-founded set theory), reproduction is nevertheless describable within formal systems.

3.7.1.2 A Universal Framework for Self-Replication

Adams & Lipson give a formal framework for describing reproduction [3], with the aim of defining the difference between trivial and non-trivial reproduction on a continuous sliding-scale, rather than as a dichotomy based on Turing-completeness, or indefinite heredity, for example. The authors view reproduction as a property of a reproducer and its environment, rather than simply a property of a reproducer, and suggest their formal framework as a means of comparing a reproducer relative to different environments.

The authors define an *environment* E as a single state of a (presumably closed) system. The set of *E–configurations* $\overline{E}$ is the set of all possible states of the system that includes E, and a *time-development function* $T : \overline{E} \times X \to \overline{E}$ (where X is the set of non-negative real numbers $\mathbb{R}^+$ or natural numbers $\mathbb{N}^+$, depending on whether we are modelling a continuous- or discrete-time system), in which $T(E, x)$ returns the state E' that exists after the amount of time x has passed. Therefore, progression between states is deterministic. A *subsystem set* X^* is the collection of all subsystems of some system X, and the set of *possible subsystems* $\overline{E}^*$ is union of all F^* such that $F \in \overline{E}$. In order to distinguish the portion of a state corresponding to a reproducer, there is a *dissimilarity pseudometric* $d : \overline{E}^* \times \overline{E}^* \to \mathbb{R}^+$, such that $d(x, y) + d(y, z) \geq d(x, z)$, $d(x, y) \geq 0$ and $d(x, x) = 0$. The *presence* $P_\varepsilon(E, S)$ of a subsystem S in an environment E within tolerance ε is defined as the probability that a randomly selected subsystem $T \in E^*$ will satisfy $d(T, S) \leq \varepsilon$, i.e., the presence of S in E is a measure of how much of S can be found in E. When $P_\varepsilon(E, S) \geq 0$ it means that S is *ε-present*, and S is *ε-possible* when there is some time t such that S is ε-present in $T(E, t)$.

The *momentary relative replicability* of a system S in E_1 and relative to E_2 with tolerance ε at time t is:

$$R_M(S, E_1, E_2, \varepsilon, t) = \log \frac{P_\varepsilon(T(E_1, t), S)}{P_\varepsilon(T(E_2, t), S)}$$

The following special cases of R_M are defined: $R_M(S, E_1, E_2, \varepsilon, t) = 0$ when $E_1 = E_2$; $R_M(S, E_1, E_2, \varepsilon, t) = -\infty$ when $P_\varepsilon(T(E_1, t), S) = 0$ and $P_\varepsilon(T(E_2, t), S) \neq 0$; and $R_M(S, E_1, E_2, \varepsilon, t) = \infty$ when $P_\varepsilon(T(E_2, t), S) = 0$ and $P_\varepsilon(T(E_1, t), S) \neq 0$.

The *relative replicability over time* τ_0 to τ_1 of a state S in environments E_1 and E_2, in which S is ε-possible in E_1 and E_2 with tolerance ε is defined as:

$$R_T(S, E_1, E_2, \varepsilon, \tau_0, \tau_1) = \log \frac{\int_{t=\tau_0}^{\tau_1} P_\varepsilon(T(E_1, t), S)\, dt}{\int_{t=\tau_0}^{\tau_1} P_\varepsilon(T(E_2, t), S)\, dt}$$

In other words, replicability over time is based on the ratio of the sums of all momentary relative replicabilities for E_1 and E_2.

The *overall replicability* is given by the limit:

$$R_O(S, E_1, E_2, \varepsilon) = \lim_{t \to \infty} R_T(S, E_1, E_2, \varepsilon, 0, t)$$

Then, the *overall self-replicability* of some system S in an environment E is given by $R_S(S, E, \varepsilon) = R_O(S, E, E - S, \varepsilon$, where S is minimally-present in E, $d(E, E')$ is minimal for $E' = E - S$ and S is not ε-present in E' but is ε-possible in E'. In other words, R_S gives a value representing how present S becomes, given that it started as a single reproducer (environment E), compared to when it isn't present (environment $E - S$). Therefore if $R_S = 0$, then the inclusion of S does not affect the likelihood of finding S in some future state; if $R_S > 0$ then S is a reproducer as it increases in number over time, and if $R_S < 0$ then S reduces in number over time (i.e., it appears to be self-destructive).

The authors present examples in which they apply their framework: a cellular automaton, a fibre-optic ring and a case of crystal growth. In every case, the authors identify reproducing systems and identify their overall self-replicability empirically: negative, positive and positive respectively.

Adams & Lipson therefore provide a framework for describing reproductive systems within their environments, and a metric for tracking the reproductive success of those systems.

3.7.1.3 Autopoiesis

The theory of autopoiesis (which means literally "self-creation") was given by Varela et al [147]. The aim of autopoiesis is to describe a class of self-creating systems, of which biological life forms are examples, in order to describe the organisation of living systems. One of the main features of autopoiesis is a description of the functionality and structure of things that reproduce.

Autopoietic systems are described in terms of *unities*, which may be readily-analysable wholes with constitutive properties, or complex systems that can be identified through analysis of their components and their relationships. Given this context, the following conditions must be satisfied in order to identify an autopoietic system:

1. the unity must have identifiable boundaries;
2. the unity must be composed of describable constitutive elements;
3. the unity must be a mechanistic system, i.e., the component properties are capable of satisfying relations that determine their interactions and transformations within the unity;

4. the boundaries of the unity must be respected by the components at the boundary through preferential neighbourhood relations and interactions between themselves;

5. the components of the boundary of the unity must be produced by interactions of the components of the unity, either by transformation of previously produced components or by transformations and/or coupling of non-component elements which enter the unity through its boundaries;

6. all other components of the unity must either be produced as in (5), or if they are not produced must be necessary permanent constitutive components in the production of other components.

Varela et al demonstrated that autopoiesis is sufficient to describe living biological systems, and although the authors were clearly influenced by biochemical systems and existing biological life, the principles of autopoietic systems have subsequently been identified in fields as diverse as communication theory, management, economics, psychology, and sociological systems (pp. 145–6, [50]). In addition, computational autopoiesis continues to be an active and influential research area within the field of artificial life [103].

3.7.1.4 Reproduction in Cellular Automata

Cellular automata consist of a set of *cells*, where each cell has a *state* and a *neighbourhood* of other cells. The states of the cells in the neighbourhood determine the state of the cell in the next time step. The way in which a cell's state is determined from the states of the cells in its neighbourhood is called the *transition rule*. The transition rule is applied to all of the cells in parallel, so that the states of all the cells are updated simultaneously [133].

States, neighbourhoods and transition rules can be anything we wish them to be. The study of cellular automata is well-developed, and the apparent fascination by many researchers is most likely a result of the surprising emergent properties of even simple cellular automata.

Take, for example, a simple cellular automata developed by Conway known as the Game of Life [51]. The cells are arranged in a 2-dimensional grid in which the neighbourhood consists of the eight adjacent cells. The state of each cell is a Boolean value: it can either be on or off. The transition rule consists of four simple rules:

1. *Survival.* If a cell is on, and there are two or three cells in its neighbourhood whose states are on, then the state of the cell in the next time step is on.

2. *Death from overpopulation.* If a cell is on, and there are four or more cells in its neighbourhood whose states are on, then the state of the cell in the next time step is off.

3. *Death from loneliness.* If a cell is on, and there are less than two cells in its neighbourhood whose states are on, then the state of the cell in the next time step is off.

4. *Birth.* If a cell is off, and there are exactly three cells in its neighbourhood whose states are on, then the state of the cell in the next time step is on.

When animated using these rules, certain configurations of cells reproduce over a number of time steps (e.g., the glider in Figure 3.1). Other configurations quickly die out or reach a steady state, others manage to generate reproducers indefinitely.

Cellular automata are ideal for research into reproduction because they can be defined formally, they are generally simple to understand, and readily generate reproducing patterns [133]. The first application of cellular automata to this problem was by von Neumann, who defined a reproducing automaton that consisted of a configuration of cells within a 29-state cellular automaton. Von Neumann's automaton was a universal constructor, in that it could construct any configuration of cells given the right program. Reproduction therefore consisted only of creating a program (i.e., self-description) that would describe the automaton and another copy of the program [149, 133].

Von Neumann's automaton can be seen as a constructive proof of the existence of computational self-descriptive reproduction (i.e., reproduction of a machine with separate self-description and reproductive mechanism). However, the reproduction process of the automaton is not tractable, and is more complex than necessary for reproduction. Codd designed an 8-state cellular automaton consisting of a universal constructor in the form of a loop, inside which the self-description was stored [30, 89]. Langton subsequently simplified Codd's design by eliminating the requirement on universal construction, resulting in "Langton's loops" [89, 90]. The reproducing loop architecture of Langton was simplified even further by Byl [22] and Ludwig [97]. The study of reproduction in cellular automata is of significant interest in the field of artificial life; Sipper [133] gave a detailed overview in 1998. Notable further developments since then include Sayama's Evoloops [129], a variation of Langton's loops incorporating reproduction, and Oros & Nehaniv's Sexyloops [111], which model sexual reproduction and parasitic infection between loops.

3.7.1.5 Reproduction Classification by Dawkins

Dawkins describes an informal classification of reproducers based on five different criteria:

1. Longevity: how long does the reproducer live?
2. Fecundity: how frequently does reproduction occur?
3. Fidelity: how accurate is the reproductive process? How likely is it that the offspring of the reproducer will be able to reproduce?
4. Does the reproducer affect the probability of reproduction (active), or not (passive)?
5. Does the reproducer give rise to an unlimited number of successive generations (germ-line), or not (dead-end)?

Dawkins' classification, which appeared in his book *The Selfish Gene* [36], is naturally tailored towards the comparative analysis of the reproductive properties of genes. Nonetheless, the classification can be applied to a variety of reproducers, as the criteria are sufficiently abstract.

An interesting feature of this classification is that each criterion is a parameter that can affect how likely it is that the reproducer will succeed in a competitive environment populated with other reproducers, e.g., the most successful reproducers will have high longevity because they will be able to reproduce more than less long-lived reproducers, assuming that everything else remains constant. For other criteria, it is less clear whether what effect a change might have, e.g., a high-fidelity reproducer might produce lots of fertile offspring, but the rate of evolution will be lower, so the reproductive success might decrease over time.

3.7.1.6 Reproduction Classification by Taylor

In his Ph.D. thesis [142], Taylor describes the implementation of an artificial life system called Cosmos, somewhat inspired by Ray's Tierra artificial life system [117]. After a description of the experiments performed within the system, Taylor gives a reproducer classification system based on three criteria:

1. The degree to which the reproductive mechanism (algorithm) is **explicit**ly encoded within the reproducer, rather than being **implicit** in the physical laws of the universe.

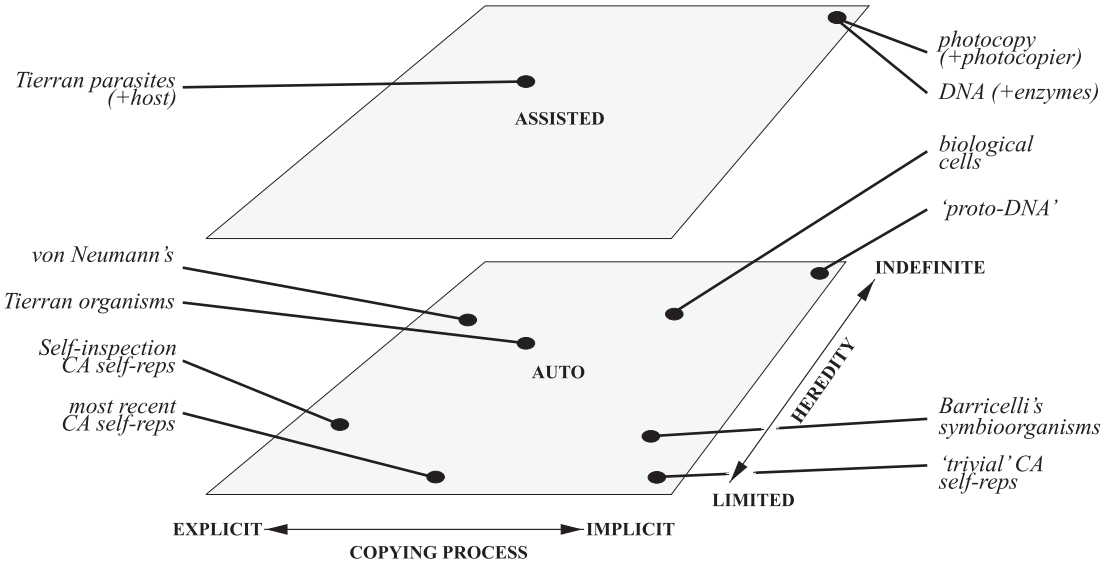


Figure 3.7: Taylor's classification of reproducers. Reproduced with permission from [142]. By Taylor's classification, this figure is therefore an assisted, indefinite-heredity reproducer with an implicitly-encoded algorithm.

2. Whether reproduction happens purely as the result of the physical laws of the universe (**auto-reproduction**), or whether it also requires additional physical and/or logical machinery (**assisted reproduction**).

3. The number of offspring of the reproducer that are capable of reproduction. i.e., whether the reproducer has **limited** or **indefinite heredity**.

Items 1 and 3 are continua, i.e., these scales are continuous, whereas item 2 is a dichotomy, i.e., this scale has two discrete classes. Taylor notes that item 3 does not classify a particular reproducer, but rather a lineage of reproducers. Also, the more explicit the encoding of the reproductive algorithm, the less likely it is (in general) that a reproducer has an indefinite heredity, and therefore criteria 1 and 3 are not independent. (For several examples of Taylor's classification of various reproducers, see Figure 3.7.) It is interesting to note that Taylor's distinction between limited and indefinite heredity is similar to Dawkins' comparison between germ-line and dead-end reproducers.

The classification presented by Taylor is informal, and is influenced by the findings of Taylor's study of the Cosmos system, in which reproducers are programs that reproduce and compete for survival within a virtual memory environment. The origins of the classification do not imply a loss of generality, however, as the classification criteria seem to be applicable to many different examples of reproduction.

3.7.1.7 Reproduction Classification by Luksha

Luksha presents two different formal classifications of reproducers [98] based on a formal model of reproduction by McMullin [102]. McMullin's formal definition of reproduction is as follows:

- Let M be a set of machine types. For any $m \in M$ let $O(m)$ denote the subset of machine types that are producible by m, i.e., $O(m)$ represents the offspring of m. Then, m is a constructor provided $O(m) \neq \emptyset$, and m is a reproducer provided $m \in O(m)$.

Luksha extends McMullin's framework to include a difference metric such that $d(x, y)$ denotes the difference between reproducers x and y, $d(x, y) = 0$ denotes that y is an exact copy of x, and $d(x, y) \leq D$ if y is considered an imitation of y, where D is some level of acceptable variation. Three sub-classes of reproduction can then be identified (let s_n denote the nth-generation descendent of the ultimate ancestor s_0):

1. Exact reproduction, in the case where $d(s_0, s_t) = 0$.
2. Near reproduction of a parent, in the case where $d(s_t, s_{t+1}) < D$.
3. Near reproduction of an ancestor, in the case where $d(s_0, s_t)$.

McMullin also describes a measure of complexity $c(m)$ that assigns a value for all $m \in M$ that describes the complexity of m by some arbitrary measure. This complexity measure is used by Luksha's second classification, which compares the relative complexity of the reproducer and the environment in which it reproduces:

1. Quasi-self-reproduction where $c(R) < c(E)$, in which reproducers are strictly dependent on a system of higher complexity which is external to the reproductive process, e.g., biological viruses and genes.
2. Semi-self-reproduction, where $c(R) \cong c(E)$, in which reproducers are of similar complexity to their environments, e.g., organisms with parasitic reproduction.
3. True self-reproduction, where $c(R) > c(E)$, in which reproducers are complex autonomous systems that reproduce in an environment consisting of basic elements, e.g., cells and self-reproducing societies (e.g., [99]).

Whilst this reproduction classification is interesting, it assumes that (i) there exists fair, accurate, and computable functions c and d, which is not guaranteed. At the very least, definition of reliable functions c and d would seem to be a difficult task.

3.7.2 Comparisons with Other Approaches

3.7.2.1 Comparison with Löfgren's Approach

The axiomatisation of self-reproduction given by Löfgren involves two possible characterisations of reproduction: symbiotic and atomic. Both cases are defined relative to certain entities. There is an obvious correlation with our approach, which is also based on entities. Furthermore, both symbiotic and atomic reproduction can be interpreted using affordance-based reproduction models.

Symbiotic self-reproduction could be interpreted as follows. Where Löfgren specifies that entity π_1 produces entity π_2, we could specify the reproduction model M_{π_2} with $\pi_1, \pi_2 \in Ent_{\pi_2}$ and $\pi_1 \in Aff_{\pi_2}(a)$ for all actions $a \in A_{\pi_2}$. Likewise, we could specify another reproduction model M_{π_1} that shows how π_2 produces entity π_1, in which the roles of π_1 and π_2 in M_{π_2} are reversed. Therefore symbiotic self-reproduction corresponds to mutually assisted reproduction of two reproducers π_1 and π_2.

Atomic reproduction could be interpreted even more simply. For some atomically self-reproducing entity π, we specify a reproduction model M_π with $Ent_\pi = \{\pi\}$. Therefore, atomic self-reproduction corresponds to unassisted reproduction of one reproducer.

Symbiotic and atomic self-reproduction can therefore be specified using affordance-based models, but the translation is not exact. For instance, affordance-based models must have the reproducer in start and end states, whereas Löfgren's definitions make no such requirement. Also Löfgren specifies reproduction functionally, without any mention of the state transition systems which are specified by those functions. As state transition systems are a necessary part of affordance-based models, the only sensible assumption to make when creating the affordance-based model is that the transition system is a trivial one (e.g., $s \overset{a}{\longmapsto} s'$). Much of the affordance-based models, which are best suited for ecological modelling of reproduction across any number of time steps, is therefore redundant.

Another interesting parallel is that Löfgren notes that "atomic self-reproduction shall result from the symbiotic self-reproduction when all the distinct entities of the symbiotic case coalesce." The idea of coalescence is also used throughout this chapter, and particularly in the proof of the Unassisted Reproduction Theorem, to show how assisted models are related to unassisted models. In this sense, the Unassisted Reproduction Theorem captures formally Löfgren's statement.

Löfgren's definitions of "reproductive" and "self-reproductive" can also be specified using affordance-based models:

- *An object A is reproductive in a surrounding S if there are objects A_i with descriptions d_i such that $A \to d_1 \to A_1$ and $A_i \to d_{i+1} \to_S A_{i+1}$ for $i > 0$, i.e., A*

forces S to produce a reproductive object.

We can specify that all objects $A, A_1, \ldots$ are members of a self set (see Section 3.4) α. Then we specify a model M_α with reproductive path $s \overset{a}{\longmapsto} s'$ and $\{\alpha, S\} \subseteq Ent_\alpha$. The notion of A forcing S to produce a reproductive object implies that A and S are active in the act of reproduction, and therefore and $\{\alpha, S\} \subseteq Aff_\alpha(a)$. We could even form a model in which d_n are entities themselves.

- *An object A is self-reproductive in a surrounding S if A produces a copy of itself in S.*

  Since the only requirement here is that A is reproductive, we form a simple affordance based model M_A with A as the reproducer, and A affording the action(s) in the reproductive path.

Löfgren also defines "production", in which an object forces the surroundings to produce another object. However, it appears to be more difficult to define this in terms of affordance-based reproduction models, since reproduction is not occurring.

The translation between Löfgren's formalisations of "reproductive" and "self-reproductive" and the affordance-based models is not exact. However, there is a correlation: Löfgren's work centers around a notion of agency to distinguish different types of reproduction, and affordance-based models are designed to specify agency within the reproductive act.

3.7.2.2 Comparison with A Universal Framework

The framework proposed by Adams & Lipson has several similarities with affordance-based reproduction models. The framework identifies reproduction as an interaction between system and environment, rather than simply a property of the system itself. This implies that viewing a system (reproducer) within a different environment (model) may provide a different perspective of an act of reproduction. The system–environment view therefore fits neatly with affordance-based models, which alter their classification through refinement, depending on differing views of the environment. A proper analysis of an act of reproduction therefore comes through an analysis of the affordance-based models that are related by refinements, rather than through a focus on any one particular model.

Further similarities between the two approaches include the framework's dissimilarity pseudometric, which gives a functional description of how the membership of a self set (see Section 3.4) might be determined; and that both approaches have a notion of presence of an entity (system) within a state (environment).

There are also several differences between the two approaches. The framework by Adams & Lipson is able to model continuous- as well as discrete-time processes, whereas affordance-based reproduction models are discrete. In principle, affordance-based reproduction models can be applied to continuous-time reproduction, as long as we give an abstract discrete-time description of the process. (For an example see the definition of the affordance-based model of the bacteriophage virus presented in Section 3.4.2.) The framework allows for the identification of systems within environments, however, there is no description of collaborative action between entities as there is with affordance-based reproduction models. The time-development function identifies future states that can result deterministically from a current state, whereas affordance-based models give a precise formulation of the state-transition system that can be non-deterministic in nature. There are other minor differences, e.g., in the specification of a reproducer and a path in an affordance-based model.

Overall, the differences between the two approaches are a result of a differing perspective on the role of a model of an act of reproduction: Adams & Lipson's system–environments give a way to track the reproductive success of a system within an environment, resulting in a model that can be used for empirical analysis of deterministic reproduction systems such as cellular automata. The aim of affordance-based models, in contrast, is to highlight the ecology of the act of reproduction, and how these ecologies and their classifications can be affected by viewing the act of reproduction differently, i.e., refining the reproduction model. The similarities of the approaches suggest perhaps that the idea of reproduction as the interaction between a reproducer and its environment, i.e., an ecological view of reproduction, has explanatory power.

3.7.2.3 Comparison with Cellular Automata

In the previous section, cellular automata were included as an example of a formal system in which reproduction can be specified and researched. However, there is a crucial difference between systems of this type and the formal reproduction models presented in this chapter. Cellular automata are formal systems, but their use as reproduction models comes as a result of the emergent reproductive behaviour of certain configurations of cells within the automaton. In other words, cellular automata are a formal system in which reproduction may occur, but our reproduction models are a formal system in which the reproduction systems of cellular automata, biological life, or any other reproducer, can be modelled and compared. Therefore, our approach can be compared to work in the fields of cybernetics and systems theory, which study the organisation of systems independent of the substrate in which they are embodied [163, 85].

3.7.2.4 Arbitrariness of Assistance

Classification of affordance-based reproduction models as assisted or unassisted is a natural application of an affordance-based model, which captures the ecological characteristics and the collaborative nature of an act of reproduction. Assistance as a classification mechanism also appears in Taylor's classification, as the auto/assisted dichotomy can be seen as analogous to the assisted/unassisted reproduction model classification presented in this chapter.

In our approach to reproduction modelling, we do not define any requirements or heuristics for identifying reproduction. Rather, we assume that a case of reproduction has already been identified[4], and that a reproduction model can then be constructed to represent this process. Our formal definitions of assistance and triviality, then, are not for the purposes of identifying reproduction, but rather to explore the structure of the space of reproducers, and show how differences in viewing what is essentially the same system can result in re-classifications.

For example, the Unassisted Reproduction Theorem states that once we have identified reproduction within a reproduction model that is classified as assisted or unassisted, then we can relate this model to another model in which a similar act of reproduction occurs (i.e., the labelled transition system is the same), but which is guaranteed to be classified as unassisted. The converse of this is the Assisted Reproduction Theorem.

One conclusion that can be drawn from the two theorems is that classification as unassisted or assisted appears not to be based upon any intrinsic property of a reproduction system, but rather is dependent upon the way in which we choose to view the system. To take the example from before, the photocopy could be classified as a reproducer in an assisted reproduction model (e.g., the photocopier is another entity which affords the actions in the path). However, the Unassisted Reproduction Theorem guarantees a re-classification (through refinement) to a related model which is classified as unassisted, which results from a "conglomeration" of the photocopy and photocopier entities into a reproductive whole. In other words, if we take this conglomerate entity as a reproducer, then it is classified as unassisted. One might argue that the conglomerate entity is not a reproducer, as the act of reproduction (i.e., photocopying) does not reproduce the photocopier. However, viewed from a more abstract perspective, the act of reproduction does preserve the presence of a photocopy and a photocopier in the start and end states of reproduction, so the conglomerate entity, at least on this level, is a reproducer.

Therefore, if we accept affordance-based reproduction models as a reasonable model of reproduction, then we must accept that classification as assisted or unassisted is only

[4]The identification of reproducing systems has been studied in detail elsewhere, e.g., [94, 69].

a result of the way in which we choose to view a particular example of reproduction, that is, classification as unassisted or assisted is arbitrary.

One could question whether the related models of the Unassisted and Assisted Reproduction Theorems, whilst they are proven to exist, describe a real-life reproduction system in a meaningful way. In other words: we have described a mathematical structure which conforms to the formal definition of an affordance-based reproduction model, but is it simply a mathematical structure, or is it indicative of some truth about reality? We will attempt to show here that it is the latter.

In the case of the Unassisted Reproduction Theorem, we prove that for every assisted reproduction model M, there is a related model $M^{\#}$ (with the same labelled transition system) that is classified as unassisted. We define this related model based on $\mathsf{E}(M)$, which denotes the set of entities that afford action in the path of model M. Using the function $h : Ent_M \rightarrow Ent_{M^{\#}}$ we map each entity in $\mathsf{E}(M)$ to the reproducer $r_{M^{\#}}$ of model $M^{\#}$. We can view this refinement as a conglomeration of the entities which collaborate in the act of reproduction. Refinement, at least in this case, describes an *ontological shift* in which our perception (denoted by a reproduction model) is altered, so that we come to view all of the entities in $\mathsf{E}(M)$ as the same entity in the related model $M^{\#}$. For example, we could have a reproduction model in which a T4 bacteriophage virus is assisted by a bacterium in its act of reproduction. The conglomerate entity, consisting of the T4 and the bacterium, is the reproducer in the model $M^{\#}$. As is the case with the conglomeration of the photocopy and photocopier described above, one could question whether a T4 virus and a bacterium actually reproduce, as copies of the virus appear to be produced, but not copies of the bacterium. However, if we consider the cyclical nature of reproduction, in which the T4 bacteriophages go on to find other bacteria to infect, then at this point the T4–bacterium conglomerate entity is reproduced. (This idea is expressed in more detail in Section 3.7.4.) Whilst this is just one possible case of an assisted model and its unassisted counterpart, we believe that conglomerate entities do make sense in other cases, and at least in all the cases that we have come across.

In the case of the Assisted Reproduction Theorem, we showed that for any non-trivial reproduction model M which is classified as unassisted, there is a refinement from an assisted reproduction model $M_{\#}$. In this case, refinement is achieved through the introduction of an entity G into model $M_{\#}$, which affords all of the actions afforded by r in M. In other words, we can see G as an entity which takes on the "reproductive responsibilities" from the reproducer. Again, we take the example of the bacteriophage virus in which there is a non-trivial model which is classified as unassisted. Then, the introduction of G which affords all of the actions that the bacteriophage used to

afford can be seen as the introduction of the physical laws of the Universe, which are ultimately responsible for any activity of the bacteriophage virus. Therefore, the ontological shift, which results in re-classification as assisted, can be seen as a "demotion" of the bacteriophage virus, in which its activity is attributed to some more fundamental entity, G. Again, in all of the examples we have come across, the generation of a "more fundamental entity" makes sense, as we can always attribute acts of reproduction to the more fundamental laws of the universe in which reproduction takes place.

3.7.2.5 Sexual Reproduction

An obvious criticism of our reproduction models is that there is just one entity that is identified as a reproducer. Therefore, this seems to be at odds with the idea of sexual reproduction, in which the genetic material from two parents comes together in the offspring. However, we can avoid this contradiction if we use the idea that the entity that reproduces in the act of sexual reproduction is the male ♂ and the female ♀, but rather the set $\{♂, ♀\}$, which we can denote $♂ + ♀$. The reproducer $♂ + ♀$ will reproduce itself over time. The requirement that the reproducer is present in start and end states of reproduction is fulfilled, even if only a single male or a female is produced during reproduction:

$$♂, ♀ \longrightarrow ♂, ♀, ♂$$
$$♂, ♀ \longrightarrow ♂, ♀, ♀$$

In other words, a complete reproductive unit composed of a male and female are present in the start state, and they reproduce and generate a male or a female. In the end state, a complete reproductive unit consisting of a male and female is still present (assuming incest is not a concern, of course). Reproductive units of three or more members can be modelled in a similar manner.

3.7.2.6 Triviality and Non-triviality

Reproduction can be identified in many different systems, from biological life forms and viruses, to artificial life forms like cellular automaton gliders, von Neumann's reproducing automaton, Tierra organisms, seeding crystals, fire, photocopies, fixed points of mathematical functions, etc. A common concern in the theoretical study of reproduction is to determine which of these reproducers are trivial, and which are non-trivial. The rationale behind this approach is that some examples (e.g., seeding crystals, photocopies) do not seem to have the same order of complexity or organisation as the others (e.g., biological life, von Neumann's reproducing automaton). One possible solution is

to place life forms on a sliding scale of complexity, with the more trivial examples at one end of the scale, and the increasingly less trivial examples further up the scale. For example, this is the approach taken by Adams & Lipson and Luksha. However, this always leaves the question of how we determine the measure of complexity of organisation.

Another approach is to divide the space of reproducers with a dichotomy, by stating some predicate which must be satisfied by all non-trivial, or trivial examples of reproduction. For example, Varela et al give a number of predicates which must be satisfied by all autopoietic systems. Autopoiesis seems to be associated with non-trivial examples of reproduction, e.g., a photocopy does not seem to satisfy at least some of the requirements. Therefore, the notion of autopoiesis can be used to divide the classification space of reproduction examples.

A combination of the above can be seen in the classifications of Dawkins and Taylor, who both use a combination of continuous and dichotomous dimensions, for which we could determine that the presence of a reproducer at a certain point within the multi-dimensional classification space indicates triviality.

Within our approach, trivial reproduction models are those in which the reproducer does not afford any of the actions in the reproductive path, i.e., it plays no part in reproduction. Non-trivial reproduction models, then, are those in which there is an action in the reproductive path which is afforded by the reproducer. Therefore, our notion of triviality versus non-triviality captures whether the reproducer is active or passive in its act of reproduction. This is similar to the dichotomy in Dawkins' classification, in which reproducers are classified according to whether they affect the probability of their own reproduction, or not. Notionally, a reproducer that affords one of its reproductive actions assists its own reproduction, and therefore reproduction would not be possible in that reproduction model without the reproducer, so in this sense it has increased the likelihood of its reproduction.

The relationship between trivial and non-trivial reproduction models as presented in this chapter is not as strong as the relationship between assisted and unassisted reproduction models presented above, as we do not yet have any formal results along the lines of the Unassisted and Assisted Reproduction Theorems. We have shown that it is possible to refine trivial models to non-trivial models, though we do not yet know whether this is possible for all or a specific sub-class of trivial and/or non-trivial models. Therefore the pursuit of this knowledge is a topic for future work.

3.7.2.7 Comparison with Multiagent Systems

Wooldridge [165] defines a software agent as "... a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives." A multiagent system is a system which "... contains a number of agents, which interact with one another through communication." An affordance-based model, amongst other things, is a model of the collaborative behaviour of different entities within a reproduction system. Alternatively, we could say that the affordance-based model describes a multiagent system in which various entities assist in different actions. Multiagent systems are of particular interest to artificial intelligence researchers, because of the emergent intelligent behaviours of communities of competing and collaborating simple software agents [165].

The recent interest in multi-agent systems has resulted in a resurgence of interest in the notion of agency, and several formal models of agency have been proposed [119]. For example, Reed & Norman describe a formal characterisation of Hamblin's action–state semantics, in which a notion of agency is made explicit [119]. The authors describe two modalities, S and T, which denote "being responsible for the achievement of a state of affairs" and "being responsible for the achievement of an action", respectively. For example, we write $\mathsf{S}_x A$ to show that agent x is responsible for bringing about state of affairs A, $\mathsf{T}_x \alpha$ to show that agent x is responsible for the action α, and $\mathsf{T}_x \alpha^y$ to denote that an agent x is responsible for action α being performed by agent y. It is the modality T that bears a resemblance to the affordance-based model function Aff.

By Definition 7, an affordance-based model has a function $\mathit{Aff} : A \to \mathcal{P}(\mathit{Ent})$ such that, for all states s, if a is possible in s then $e \; \varepsilon \; s$ for all e in $\mathit{Aff}(a)$. In other words, Aff maps an action to a set of entities which must be present if that action is possible in a state. Reed & Norman's modality T is somewhat similar in meaning, except that $\mathsf{T}_x a^y$ specifies that x is responsible for y doing a, i.e., we specify that x somehow is responsible for an action that is performed by some other agent y (assuming $x \neq y$). In contrast, within affordance-based models we say only that $\mathit{Aff}(a) \subseteq \mathit{Ent}$ is a set of entities that are collectively responsible for a, i.e., in affordance-based reproduction models the notions of responsibility and doing are not separate.

Another research area relevant to multiagent systems in which notions of agency appear is cooperation logics [146]. A formula $[1, 2]p \wedge q$ states that the coalition of agents 1 and 2 can act in such as way as to make the formula $p \wedge q$ true. Clearly, this can be applied to affordance-based models: $[e_1, e_2, r]\rho$ might denote that entities e_1, e_2 and the reproducer r might collaborate in (i.e., afford) the act of reproduction, in which ρ is true iff reproduction of r has occurred. In terms of affordance-based models we could say that $\mathit{Ent} = \{e_1, e_2, r\}$ and $\mathit{Ent} \subseteq \mathit{Aff}(a)$ for all actions a in the reproductive

path. Furthermore, the same formula might denote that the same entities collaborate on a particular action α, if ρ is redefined so that it is true iff the action α has occurred.

Whilst the similarities between affordance-based modelling and formal definitions of agency are interesting, it is obvious that they are not the same thing, as the former is defined with the aim of specifying the ecological conditions under which reproduction can take place, and the latter are concerned with modelling the behaviour of communities of agents. For example, significant changes and additions would need to be made to the formalisation of action–state semantics in order to allow for classification and refinement of models, which is one of the main contributions of the work presented in this chapter. Likewise, affordance-based reproductions models would need to undergo major changes in order to mimic the sophisticated concepts defined using modal logics.

The considerable interest in formal notions of responsibility is indicative of the relevance of affordance-based reproduction models beyond domains in which reproduction is a concern, and holds promise of a cross-fertilisation of ideas between formal reproduction modelling and multiagent systems[5].

3.7.2.8 Comparison with Formal Methods for Concurrent Systems

Notions of system and environment are also important within formal methods, specifically in the specification and verification of concurrent systems. Concurrent systems are those in which programs execute concurrently. These programs may interact with each other, making the task of specification and verification more difficult than for single programs. There is an obvious parallel with our affordance-based models of reproduction, in which we focus on a single "program" (the reproducer) and its environment of other "programs" (entities other than the reproducer). The interactions of these programs are laid out in their affordance relationships.

One such formalism for specification and verification of concurrent systems is the rely/guarantee method proposed by Jones [76], which works as follows. A program acts in an environment. Actions by the program must satisfy a *guarantee* condition. During its execution the program can trust that the *rely* condition, which specifies the behaviour of the environment, will hold. The rely/guarantee method therefore enables software developers to develop systems of concurrent programs.

A similar notion to rely/guarantee can be found in Definition 7, in which affordance-based models are defined. The function Aff is defined such that if an action a is possible in a state s, then $e \; \varepsilon \; s$ for all entities $e \in Aff(a)$. If we think of the reproducer as the

[5]Perhaps the cross-fertilisation may be a necessity should reproductive software and/or robotic agents, e.g., for the purposes of space exploration [49], become a practical reality. The burgeoning research area on kinematic self-replicating machines indicates that this could soon be the case [50, 39].

"program", then this definition specifies a rely condition that the environment must satisfy: all affording entities must be present in the necessary states of the program's execution (i.e., reproductive process). Conversely, the affording entities can be thought of as programs which guarantee their presence in necessary states.

However, there are significant dissimilarites between affordance-based models and formal methods for concurrent programs, such as rely/guarantee, due to the fact that they are designed to solve different problems. For instance, formal methods for concurrent systems are designed to have a rich vocabulary for specifying the interactions between concurrently executing programs, where as affordance-based models state only that there are interactions between entities (using the *Aff* function), and not what form they take.

3.7.3 Comparison with Rosen's Ideas on Life

3.7.3.1 Life Itself

Rosen describes the "machine metaphor" as the view of systems in terms of states and transitions between states, and traces the lineage of this reductionistic idea to Newtonian mechanics. In his book, *Life Itself* [123, 124], Rosen posits that scientific reductionism[6]is unfit for modelling life. Life, Rosen says, is organisational and requires a *relational* model consisting of interacting *components* which acquire their definition of function from the system of which they are a part.

There are at least two overlaps between Rosen's ideas and the ideas presented in this chapter. The first is Rosen's criticism of reductionist models of biological systems. Reproduction is, amongst other things, a biological phenomenon, and therefore affordance-based reproduction models are descriptions of the biological world. The reproduction models presented in this chapter are not reductionistic, however, as we do not suggest that reproduction can be understood through a reduction to some process at a lower level of abstraction, e.g., biochemical reactions. Rather, we model reproduction at a natural level of abstraction, viewing reproduction at an ecological level at which entities can cooperate in the act of reproduction. The second overlap is Rosen's use of interacting components to describe a relational model of a biological system. Within affordance-based reproduction models, we can identify the entities as being analogous to Rosen's components, and the affordance function as a model of the functional relationships between models. Therefore, the approach to modelling of

[6]Reductionism in science is encountered when the problem of understanding a complex system is reduced to the problem of understanding its constituent parts. An example of reductionism is the idea that an understanding of physics, which is concerned with the physical properties of the Universe, is all that is needed to understand biological life.

reproduction presented in this chapter is in accordance with Rosen's argument against reductionism in biology.

3.7.3.2 Rosen's Paradox

Rosen described a paradox that arises when defining reproduction in terms of certain mathematical functions, which disappears when a different kind of function is used [122]. Whilst our reproduction models contain no information about the mechanism of reproduction other than abstract information on states, entities, and so on, it is interesting that there is a parallel between our notions of unassisted versus assisted reproduction, and Rosen's description of the conditions under which the logical paradox arises. Specifically, Rosen's paradox comes into play when the mathematical function we use appears to describe unassisted reproduction, and disappears when we introduce a notion of assistance.

Suppose we have a unassisted reproduction model M_u. Although our model does not describe functionally the mechanism by which reproduction takes place, we might suppose what is happening within such a model. The model is unassisted, so we know that no entity other than the reproducer can afford actions in the reproducer's path to the reproducer. Therefore, we might think that reproduction is taking place only as a result of the reproducer's agency, e.g., for all actions a in the path p_{M_u}, $r \in Aff_{M_u}(a)$. We might characterise this using a function $f : A \to B$ which maps admissible inputs from A to admissible outputs in B. We know that f is reproducing, that is, $f \in B$. However, Rosen showed that if this is the case then we reach a paradox because the definition of f depends on the proper definition of A and B, and therefore neither the mapping f not the range B can be defined until the other is defined.

However, Rosen also showed that there is no paradox if we define a mapping

$$F : A \times B \times (A \times B) \to (A \times B) \times (A \times B)$$

which takes a construct ab and components a and b, and produces a copy of ab, i.e., $F(a, b, ab) = (ab, ab)$. In other words, ab is the reproducer, which is "assisted" in reproduction by the function F.

If f was the reproduction method used in M_u, then as a consequence of the Assisted Reproduction Theorem, there must be a refinement of M_u, which we shall call $M_{u\#}$, such that $M_{u\#}$ models an assisted reproduction system. Since $M_{u\#}$ gives a description of assisted reproduction, we might characterise this abstractly using the function F instead of f.

It is logical to assume that any formulation of reproductive behaviour along the

lines of f is incorrect, and that reproduction that may be correctly characterised as F may be errantly described using f. In the context of affordance-based reproduction, the function f reproduces itself, and is therefore an example of unassisted reproduction, whereas F can be seen as an entity that assists ab in reproduction.

For example, we consider a model of Langton's loops. One model, M_l, is unassisted, and another M_l' is assisted. For example, M_l' could correspond to a model of Langton's loop in which the transition rule (corresponding to Rosen's function F) enters, and necessarily assists in every action in the reproductive path of the loop. Such a model is logical, as the transition rule is indeed essential to the loop's reproductive behaviour. Suppose that the Assisted Reproduction Theorem tells us that $M_l' \longrightarrow M_l$. We know that in order to characterise M_l might result in an invocation of a paradoxical function such as f, whereas M_l' does not, since f is not an accurate characterisation of assisted reproduction. Therefore, the Unassisted and Assisted Reproduction Theorems relate the different models of the same reproductive process, and where we have a paradoxical unassisted model analogous to the case where f was a reproducer, we are able to refine this model to another assisted model which is non-paradoxical.

3.7.4 Reproduction as Preservation of Information Over Time

In the same way that reproduction models of the same reproducer can be specified at different levels of abstraction, the entities within the model can be specified at different levels of abstraction. When we denote that an entity e is present in a state s ($e \; \varepsilon \; s$), it is possible we are talking about an abstract entity. For example, e could be a member of a species, meaning that $e \; \varepsilon \; s$ denotes that any individual organism from that species is present in the state s. We described this idea in Section 3.4 as a "self set", e.g., all of the various individuals i in a species are members of a self set I , and all members of the self set are identifiable as e within a reproduction model[7].

The ideas of self sets and abstract entities make sense if we think of reproduction as the preservation of information over time. By Definition 6, to say that some entity r is a reproducer is to say that it is present in the start and end states of a reproductive path. When combined with the concept of self sets, these models allow for an entity of one species to reproduce and create a different entity of the same species, or even of a different species, depending on how we define the self set. We also have room for evolution, in that we can identify all possible progeny of a given reproducer as a self set, and we do not necessarily specify that a self set is finite.

Another interesting property of reproduction models is that we specify that the reproducer is present in the start and end states. There is also an implicit assump-

[7]Of course, self sets are inspired by Cohen's viral sets [32].

tion that the reproductive path might be executed an unlimited number of times, for example, if the path describes the reproduction of a glider on an unbounded cellular automaton grid, there is no limit to the number of times that the glider can reproduce. We assume that the end state of reproduction can be updated to result in the start state, and therefore reproduction can happen once more. In the simplest possible scenario, the end state *is* the start state, and the reproductive path becomes a cycle.

This raises an interesting philosophical question: if we can portray a reproductive process as a linear path or as a cycle, then are the start and end states for the linear representation arbitrary? Clearly, if we design a labelled transition system as a cycle then there is no start or end state. If we take this information and apply it to real-life reproduction examples, does this imply that there the start and end states of reproduction are unidentifiable also?

It is possible that this is the case. Take, for example the case of the bacteriophage reproduction described in Section 3.4.2. In Equation 3.1 we specified the reproductive path as follows:

$$s_1 \overset{\mathbf{a}}{\longmapsto} s_2 \overset{\mathbf{i}}{\longmapsto} s_3 \overset{\mathbf{s}}{\longmapsto} s_4 \overset{\mathbf{m}}{\longmapsto} s_5 \overset{\mathbf{r}}{\longmapsto} s_6$$

If this path is cyclical, then we can also represent the above as

$$s_2 \overset{\mathbf{i}}{\longmapsto} s_3 \overset{\mathbf{s}}{\longmapsto} s_4 \overset{\mathbf{m}}{\longmapsto} s_5 \overset{\mathbf{r}}{\longmapsto} s_6 \overset{\mathbf{X}}{\longmapsto} s_1 \overset{\mathbf{a}}{\longmapsto} s_2,$$

for example. In this example, $\mathbf{X}$ is the action described above, that takes the "end" state s_6 back to the start state s_1. (Again, it may be the case that $\mathbf{X}$ does not affect the state at all, making $s_1 = s_6$.) Naturally, we might think of s_1 as being the start state of bacteriophage reproduction, as this is the state preceding the **a**ttachment of the virus to the cell. However, this need not be the case: we could think of s_2 as the start state of this reproduction process. The state s_2 is where the bacteriophage is attached to the wall of the host cell, and is ready to inject its genome into the host cell. However, this raises another question: if s_2 is the start state and s_1 is the end state, which entity is the reproducer? We know by Definition 6 that the reproducer must be present in start and end states. However, in the example given in Section 3.4 we said that the bacteriophage need only be present in s_1, s_5 and s_6. However, we can resolve this contradiction if we suppose that the reproducer is perhaps not the virus as it is pictured in Figure 3.2, but rather the real reproducer is the information corresponding to the bacteriophage virus, which is present at every point in the reproductive process. In states s_1 and s_2 there is some information corresponding to the bacteriophage, stored as genotype and phenotype within the mobile, pre-infection stage virus. In state s_3 this version of the virus is gone, as after infection we are left with a deflated bacteriophage husk

attached to the outside of the cell, and inside the cell we have an RNA bacteriophage genome floating around, waiting to be synthesised by the cell's reproductive machinery. However, the information needed to make the mobile virus from s_1 and s_2 is still present in the potentiality of the genome and the reproductive machinery of the host cell[8]. Similarly, we can see that for each step in the reproductive process the information corresponding to the bacteriophage remains present. Therefore, this example supports the idea of reproduction as preservation of information over time.

This discussion of reproduction and information seems reminiscent of Dawkins' book *The Selfish Gene* [36], in which the reproducers in an act of reproduction are not organisms, but rather the genes contained within them. The organism's phenotype is simply an expression of those genes, and it is "designed" with only the aim of reproducing the genes that created it. However, the notion of information described above is not necessarily genetic information, and therefore the idea of reproduction as preservation of information over time cannot be reduced to the idea of selfish genes. For example, a computer virus might reproduce without any self-descriptive information (i.e., a genome), as it obtains one during its act of reproduction by self-analysis.

Another consequence of reproduction as the preservation of information over time is an explanation of trivial "fixed point" reproducers like the pen on the desk described in Section 3.1, or a fixed point of a mathematical function. These examples of reproduction are essentially conditions which, by the laws of the universes in which they exist, are stable and remain unaffected from one moment to the next. If the preservation of information over time is reproduction, then any condition which is stable over time is a reproducer. Perhaps then, the more complex reproducers are those which progress through a series of states (a reproductive path/cycle) in order to preserve that information. These non-trivial reproducers include biological and artificial life, as well as non-steady state trivial reproducers that exhibit change, e.g., growing crystals and fire.

3.7.5 Further Application to Artificial Life

We have shown by examples that it is possible to model the reproduction of computer viruses, biological viruses and cellular automaton-based artificial life forms such as

[8] We can think of the reproductive machinery as some function f, and the genome g as the function's argument. Then, $f(g)$ is the application of f to g, which results in the reproduction of the mobile bacteriophage b, so we could say that $f(g) = b$. The function application $f(g)$ always denotes b, and if we take f to be a function which is computed, then there is some period of time between $f(g)$ being specified, and b, the result, being computed. However, the progression from $f(g)$ to b is inevitable, and all of the information needed to compute b from f and g is present at the stage of function application $f(g)$. It is interesting to note that at this point in the bacteriophage's reproductive cycle, we are back to von Neumann's reproducing automaton, which contained an input tape (genome) and constructor (reproductive machinery) sufficient to reproduce itself.

gliders and Langton's loop. These examples were chosen for their clarity; in general, computer and biological viruses are well-understood real-life phenomena. Also, these are interesting applications of reproduction model classification and refinement; both computer viruses and biological viruses have many obvious entities, and are therefore ideal illustrations of the Unassisted and Assisted Reproduction Theorems.

It is obvious, but relevant, to state that models and refinements of other classic artificial life examples are also possible; e.g., von Neumann's reproducing automaton [149]. Virtual machine-based artificial life environments such as Tierra [117, 118], Core War [37] or Cosmos [142], have obvious characterisations of multiple dissimilar entities, and these would provide interesting examples of our approach to reproduction modelling and classification. These systems are based on programming language instructions at a similar level of abstraction to assembly language, and therefore were not included as examples in this chapter as they would not have provided sufficient succinctness or clarity for the demonstration of our reproduction models. However, there is ample evidence, in our own work (see Chapter 2) and the Rewriting Logic Semantics Project [105], that these kinds of programming languages can be formalised using systems like Maude, and therefore these could be used to create affordance-based reproduction models.

Chapter 4: Formal Affordance-based Models of Computer Viruses

4.1 Introduction

In this chapter we present a new approach to the classification of computer viruses based on Gibson's theory of affordances [55, 56]. This approach arose from work on the related problem of affordance-based reproduction model classification, described in the previous chapter, in which reproduction models can be classified as assisted or unassisted, trivial or non-trivial, or using general-purpose predicates known as aspects. In this chapter, we will classify computer viruses on the unassisted–assisted axis, and we will show that a single computer virus can be classified differently depending on the construction of the affordance-based computer virus reproduction model. We will demonstrate that the difference in construction between reproduction models is analogous to viewing the same computer virus with different anti-virus behaviour monitors. Therefore we are able to describe formally the difference between behaviour monitors with respect to a particular computer virus, thus giving a practical application of the affordance-based reproduction models presented in the previous chapter.

The approach presented here differs from other models and classifications of computer viruses, e.g., those described in Section 4.4.1, in that it is constructed upon a formalised abstract ontology of reproduction based on Gibson's theory of affordances. Using our ontology we can classify computer viruses at different abstraction levels, from behavioural abstractions in the vein of Filiol et al [46], to low-level assembly code semantical descriptions in the vein of the work on metamorphic computer virus detection in Chapter 2.

As was described in Chapter 1, computer viruses can be detected in a variety of ways, which can be divided into those based on static or dynamic analysis. Behaviour monitoring is a form of dynamic analysis, which involves observing the behaviour of programs to discover suspicious behaviour. If a suspicious behaviour is observed, then the behaviour monitor can flag that program or process for further action or investiga-

tion, as it is likely to contain malware.

The capabilities of different behaviour monitors will vary, and therefore it is possible that a computer virus might be detectable using one behaviour monitor, but not another. In fact, a recent study by Filiol et al has demonstrated the inconsistency of the capability of behaviour monitors in different anti-virus software [46]. In this chapter we will show how the method of classifying viruses as invisible or visible to behaviour monitoring software can be equivalent to classification of formal computer virus reproduction models as unassisted or assisted respectively. Classification, as we will show, is also possible "by hand", but automation is advantageous given the frequency of malware occurrence and the laboriousness of manual classification. To this end we describe the automation of affordance-based computer virus reproduction model classification.

One possible application of this approach is to increase the efficiency of anti-virus software. Suppose that an anti-virus software system has a variety of computer virus detection strategies, including behaviour monitoring, signature scanning and so on. If system resources are limited, and a full search for viruses using a non-behaviour monitoring method is therefore not practical, then it is logical for anti-virus software to try to prioritise the detection (by non-behavioural monitoring means) of those viruses that are invisible to behaviour monitoring software. Our approach is therefore useful in this regard, as it can be used to automatically classify computer viruses as visible or invisible to anti-virus software. This may be of particular use on systems where resources are limited, such as mobile computing systems.

It is possible that similar approaches have been used previously to improve the efficiency of anti-virus software. Therefore, the main novel contribution of this chapter is not a new means of increasing the efficiency of anti-virus software, but rather a theoretical explanation of the effects of viewing the same computer virus with different software, and a proof of the relevance to, and applicability of, affordance-based reproduction models to computer viruses and other forms of reproducing malware.

4.1.1 Chapter Overview

In Section 4.2 we present our formal computer virus reproduction models, and describe formally the difference between unassisted and assisted classifications of reproduction models. We give several examples of formal reproduction models of real-life computer viruses, and construct these based on low and high levels of abstraction. We then show how decisions made in the formal model can result in different classifications of the same computer virus.

In Section 4.3 we show how this flexibility of classification can be exploited, by using it to tailor automatic computer virus classifications to the capabilities of different

anti-virus behaviour monitoring software. We also discuss a potential application to computer virus detection: the development of an automatic classification system that separates viruses that are detectable at run-time by behaviour monitoring from those that are not. We show how ad hoc reproduction models can be generated and classified automatically using static and dynamic analysis. By defining the notion of external agency (on which we base the automatically-generated reproduction models) in accordance with the capabilities of different anti-virus behaviour monitoring software, the classifications of computer viruses are tailored to suit different anti-virus behaviour monitors. We demonstrate this with the formal executable language Maude, which we use to give a specification of an automatic computer virus classification system based on dynamic analysis. We specify the various capabilities of behaviour monitoring software using Maude, and show that they result in different classifications of the same computer virus. We show how it is possible to develop metrics for comparing those viruses that depend on external entities, so that viruses that rely on external entities can be assessed for their potential difficulty of detection at run-time by behaviour monitoring.

Finally, in Section 4.4 we give an extensive overview of other computer virus classifications in the literature, and compare them with our affordance-based computer virus reproduction model classification.

4.2 Computer Virus Reproduction Models

4.2.1 Formal Models of Computer Virus Reproduction

Our formal models of computer virus reproduction are based on the formal affordance-based reproduction models presented in Chapter 3. Our classification of reproducers is based on the ontological framework given by Gibson's theory of affordances. Originally Gibson proposed affordances as an ecological theory of perception: animals perceive objects in their environment, to which their instincts or experience attach a certain significance based on what that object can afford (i.e., do for) the animal [55, 56]. For example, for a small mammal, a cave affords shelter, a tree affords a better view of the surroundings, and food affords sustenance. These relationships between the animal and its environment are called affordances. Affordance theory is a theory of perception, and therefore we use the affordance idea as a metaphor: we do not suggest that a computer virus perceives its environment in any significant way, but we could say metaphorically that a file affords an infection site for a computer virus, for example.

Our affordance-based computer virus reproduction models are, essentially, a subclass of affordance-based reproduction models. In the case of a particular computer virus, it is natural to specify the virus as an entity, with the other entities composed

of those parts of the virus's environment which may assist the virus in some way. Therefore, we could include as entities such things as operating system application programming interfaces (APIs), disk input/output routines, networking APIs or protocols, services on the same or other computers, anti-virus software, or even the user. We are able to include such diverse entities in our models since we do not impose a fixed level of abstraction; the aim is to be able to give a framework that specifies the reproductive behaviour of computer viruses in a minimal way, so that classifications can be made to suit the particular circumstances we face; we may wish to tailor our classification so that viruses are divided into classes of varying degrees of difficulty of detection, for example.

We assume that any model of a reproductive process identifies the states of affairs within which the process plays itself out. For computer viruses, these states of affairs may be very clearly and precisely defined: e.g., the states of a computer that contains a virus, including the files stored on disk, the contents of working memory, and so forth. Alternatively, we can use abstract state transitions corresponding to abstract behaviours of the computer virus. We will demonstrate how these models can be constructed, and how they are used in computer virus classification. Reproduction models are usually based on a sense of how the computer virus operates and interacts with its environment; different points of view can result in different reproduction models of the same virus, each with its own classification.

This, of course, allows for both abstract reproductive systems where we have identified abstract actions which correspond to the virus's behaviour, as well as low-level modelling at the assembly code or high-level language statement level. As is the case with affordance-based reproduction models, we assume that the relation $v \ \varepsilon \ s$ can be made abstract enough to accommodate an appropriate laxity in the notion of entity: i.e., we should gloss $v \ \varepsilon \ s$ as stating that the entity v, or a copy of v, or even a possible mutation of v by polymorphic or metamorphic means, is present in the state s. In computer virology, such an abstraction was explicit in the pioneering work of Cohen [31], where a virus was identified with the "viral set" of forms that the virus could take. This approach is useful for polymorphic or metamorphic viruses that, in an attempt to avoid detection, may mutate their source code.

This discussion is summarised in

Definition 17. *An* affordance-based computer virus reproduction model *is an affordance-based reproduction model*

$$(S, A, \longmapsto, Ent, v, \varepsilon, p, Aff) \ ,$$

where

- $(S, A, \longmapsto)$ *is a labelled transition system, describing the states and state transitions of the computer executing the virus;*
- *Ent is a set of entities present during the virus's execution, with* $v \in$ *Ent the particular computer virus that reproduces in the model;*
- p *is a path through the transition system representing the reproduction of the virus* v*, i.e., a particular path which culminated in an offspring of the virus being generated.*

Note that every aspect of affordance-based models, as given on page 74, is preserved — we simply apply affordance-based models to the problem of specifying computer virus reproduction. As is the case with affordance-based models, the states and transitions in the model may be low-level (e.g., states of the memory and file system) or high-level (e.g., states corresponding to abstract actions, such as opening a file for reading and writing).

As a result of this definition there are some interesting questions that arise. First, we know that polymorphic and metamorphic computer viruses can vary syntactically, so which of these variants is the one specified in this reproduction model? Second, if a polymorphic or metamorphic virus is able to alter its syntax, then how do we define the path of the virus's reproduction model?

Both questions can be answered by invoking Cohen's viral sets, discussed above, and generalised to "self sets" in Chapter 3. In evolutionary systems, there is variation both in the genome and the phenome, so the first question is equivalent to "which of all the possible xs of species y are we specifying in the model?" In other words, the reproducers specified in our reproduction models are abstract. In biology we are able to identify dissimilar entities as being of the same species by their behaviour, or physiology, for example. In a similar way, we can identify the various allomorphs of a metamorphic computer virus through definition of a viral set that enumerates the possible generations, or even by using an abstract description of the virus's behaviour.

The second question is related to the first. The generations of a given polymorphic or metamorphic virus are not semantically equivalent, but must remain behaviourally equivalent. Therefore, we can define a typical execution run as any sequence of instructions that leads to the behaviour we expect of the virus. Since all generations of the virus will share this behaviour, we can use a description of this behaviour to define the reproductive path. Another way might to be to abstract from the particular instructions used by the metamorphic computer computer virus, and use these abstract actions to construct the path.

We shall see below how these formal models of computer virus reproduction can be used to classify computer viruses and other forms of reproducing malware.

4.2.2 Classifying Computer Viruses

The key distinction in our classification of computer viruses is the ability to distinguish between computer viruses which require the help of external entities, and those that do not. We call the former *unassisted* computer viruses, and the latter *assisted* computer viruses.

As was the case in Chapter 3, it is not the reproducer itself that is classified as unassisted or assisted, but its reproduction model. In fact, it is possible to create affordance-based reproduction models of the same computer virus that are classified differently. In Section 4.3, we use this flexibility to tailor our reproduction models to the particular abilities of different anti-virus behaviour monitors, and classify as assisted only those viruses detectable by the behaviour monitor.

In addition, our reproduction models do not enforce a particular level of abstraction. For example, we could create a reproduction model of a computer virus in which the states are the states of the processor executing the virus, and the actions are the assembly language instructions which the processor executes. Alternatively, we could view the virus as an abstract entity with a certain number of abstract behaviours, e.g., "opening a file" or "copying data". As we shall see later in this section, the ability to model viruses at different levels of abstraction is advantageous, because it can make the modelling and classification process much simpler.

For example, the reproduction models of the Unix shell script virus and the Archangel virus presented in Sections 4.2.3 and 4.2.4 are of a low abstraction level, in that there is one action in the path for every statement of the virus's code. However, for the sake of simplicity in Section 4.2.5 we will present a more abstract model of computer virus behaviour, in which the individual statements which compose the Strangebrew virus are abstracted to generalised actions that correspond to abstract reproductive behaviours such as "open host file" or "search for a file to infect". These abstract models are efficient means of classification "by hand", as computer viruses often contain thousands of lines of code. However, in Section 4.3 we will show how classification using "concrete" models (i.e., one action per instruction/statement) can be achieved by automated, algorithmic means.

Regardless of the level of abstraction of a reproduction model, the overall distinction between unassisted and assisted computer virus reproduction models remains the same.

Definition 18. *A computer virus reproduction model can be classified as* unassisted *iff there is no entity e, different from the computer virus v, in* $\mathsf{E}(M)$*. Conversely, a*

```
    ...
4   echo st=$sq${st}$sq > .1;
5   echo dq=$sq${dq}$sq >> .1;
6   echo sq=$dq${sq}$dq >> .1;
7   echo $st >> .1;
8   chmod +x .1
```

Figure 4.1: Statements from the Unix shell script virus showing use of `echo` and `chmod`.

computer virus reproduction model can be classified as assisted *iff there is some entity* e *different from* v *in* $\mathsf{E}(M)$.

Since affordances are actions in the labelled transition system that are not possible without the presence of some entity, we say that if there are any actions in the computer virus's reproduction path (which could be abstract actions such as "open file" or less abstract examples like a specific instruction `mov eax,ebx`) that are afforded by entities other than the virus itself, then the virus's reproduction is assisted in some way, and therefore the reproduction model is classified as assisted. In the converse scenario, where there are no actions in the reproduction path of the computer virus that are afforded by entities other than the virus, then the virus's reproduction is not assisted in any way, and therefore the resulting classification of the reproduction model is unassisted.

4.2.3 Modelling a Unix Shell Script Virus

The virus given in Figure 4.1 is a Unix shell script virus which runs when interpreted using the Bourne-again shell (Bash). The first three lines of the virus define three variables that contain the program code and aliases for single and double quotation marks. The next three statements of the program code output these data into a new file called `.1`. The seventh statement of the program appends the program code to `.1`, and the final statement of the program changes the file permissions of `.1` so that it is executable. At this point the reproductive process is complete.

We consider a typical execution run of the Bash virus, i.e., we neglect any anomalies which might prevent the reproductive process from completing, such as the hard disk crashing or the user terminating an essential process. We define a model of the Bash virus's reproduction M_B as follows.

We base the labelled transition system on the statements of the Bash virus, so that each statement corresponds to an action in the path. Therefore, we define nine states, $S = \{s_1, s_2, \ldots, s_9\}$ and eight actions $A = \{a_1, a_2, \ldots, a_8\}$, where statement i of the virus code (see Figure 4.1) corresponds to the transition $s_i \overset{a_i}{\longmapsto} s_{i+1}$. Therefore each

statement in the shell script virus is an action, and the states therefore correspond to the states of the shell which runs the script. The reproductive path is therefore

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} \ldots \overset{a_8}{\longmapsto} s_9$$

from starting state s_1 to final state s_9.

Next we must consider which entities are present in the reproduction model. The virus uses the `echo` and `chmod` commands, which are actually programs within the Unix file system, and are called by the shell when the virus executes. Therefore, we can model `echo` and `chmod` as entities, and since the Bash virus reproduces, we can specify that the set of entities $Ent = \{v, \texttt{echo}, \texttt{chmod}\}$, where v is the Bash virus.

The computer virus could not execute without `echo` and `chmod`, and we can model this using affordances, i.e., `echo` and `chmod` afford certain actions to the virus. These actions are the actions in which the `echo` and `chmod` commands are used. For example, we can say that the actions a_4, a_5, a_6 and a_7 are afforded by the `echo` entity to the virus because these actions correspond to statements in which the command `echo` appears. Similarly, a_8 is afforded by the `chmod` entity to the virus, because a_8 is the action corresponding to the eighth statement, which contains the `chmod` command. Formally, we say that

$$\mathit{Aff}(a_4) = \mathit{Aff}(a_5) = \mathit{Aff}(a_6) = \mathit{Aff}(a_7) = \{\texttt{echo}\}$$

and

$$\mathit{Aff}(a_8) = \{\texttt{chmod}\} \ .$$

Since we know that these actions are afforded by other entities to the reproducer, these entities must be present in the states preceding these actions, in line with condition 4 of Definition 7. Therefore, `echo` ε s_3, `echo` ε s_4, `echo` ε s_5, `echo` ε s_6 and `chmod` ε s_7. In addition, we know that the Bash virus reproduces, and by Definition 7, it must be present in the start and end states of the reproduction path, i.e., $v \ \varepsilon \ s_1$ and $v \ \varepsilon \ s_9$.

Classification as unassisted or assisted depends upon whether there are any entities other than the reproducer which afford actions in the reproduction path. Actions a_4, a_5, a_6, a_7 and a_8 are actions in the path that are afforded to the virus by entities other than the viruses, and therefore by Definition 18 reproduction model M_B is classified as assisted.

This is just one way to model the reproduction of the Bash virus, however. For example, we could consider no entity other than the reproducer itself. Let us call this reproduction model M'_B. Let the labelled transition system of M'_B be the same as M_B, i.e., $S_{M'_B} = S_{M_B}$, $A_{M'_B} = A_{M_B}$ and $\longmapsto_{M'_B} = \longmapsto_{M_B}$, and let the reproducer be the Bash virus, as before, i.e., $r_{M'_B} = r_{M_B} = v$. Let the Bash virus's reproduction path and

start/end states be as before, and so $p_{M'_B} = p_{M_B}$ and $s_{s_{M'_B}} = s_{s_{M_B}}$ and $s_{e_{M'_B}} = s_{e_{M_B}}$. Our model M'_B differs from M_B in that we assume that `chmod` and `echo` are given. So, the only entity present is the virus itself, and therefore $Ent_{M'_B} = \{v\}$. Here, affordances are not needed, and so $Aff_{M'_B}(a) = \emptyset$ for all $a \in A_{M'_B}$. Again, we know that the Bash virus reproduces and therefore is present in the start and end states of the reproduction path, so $v \; \varepsilon_{M'_B} \; s_1$ and $v \; \varepsilon_{M'_B} \; s_9$. Since there are no actions in the path that are afforded to the virus by another entity, we know that by Definition 18, M'_B is classified as an unassisted reproduction model.

4.2.4 Modelling Virus.VBS.Archangel

Archangel (see Figure 4.2) is a Visual Basic script virus written for the Microsoft Windows platform. Archangel starts by displaying a message box, and declaring some variables. In line 5 the virus obtains a handle to the file system in the form of an object `fso` of the `FileSystemObject` class. A new folder is created, and then Archangel uses the `CopyFile` method of the `fso` object to create a copy of itself called `fun.vbs`. This method call uses a variable from the `WScript` class called `ScriptFullName`, which contains the name of the Visual Basic Script file containing the Archangel virus. The Archangel virus uses this method to reproduce a further five times. In addition to its reproduction behaviour, Archangel executes its payload and attempts to run one of its offspring via a Windows script called `autoexec.bat`.

We define a computer virus reproduction model for the Archangel virus called M_A. The labelled transition system is constructed in a similar way to the Bash virus in the previous section, with one action corresponding to one statement. However, the flow of control is more complex, as Archangel uses two conditional if–then statements to execute lines 6 and 12 conditionally. As a result, the labelled transition system branches at each of these points (see Figure 4.3). One possible reproduction path corresponds to the case where the guards of the two conditional statements are true, and we specify this path in our reproduction model:

$$p_{M_A} = s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} \ldots \overset{a_{32}}{\longmapsto} s_{33}$$

There are two different objects which assist the Archangel virus at different points in its reproduction: `fso` and `WScript`. We could define these as two different entities which afford the virus certain actions. Alternatively, we could consolidate them into one entity representing the Windows Script Host which provides library classes to Visual Basic scripts. The Archangel virus is also an entity which must appear in the model, and therefore we define $Ent_{M_A} = \{v_A, \texttt{wsh}\}$, where v_A is the Archangel virus and `wsh`

```
    ...
   If Not fso.FolderExists(newfolderpath) Then
6    Set newfolder = fso.CreateFolder(newfolderpath)
   End If
7  fso.CopyFile Wscript.ScriptFullName, "C;\WINDOWS\SYSTEM\fun.vbs", True
8  fso.MoveFile "C:\WINDOWS\SYSTEM\*.*","C;\WINDOWS\MyFolder\"
9  fso, newfolder, newfolderpath
10 newfolderpath = "c:\WINDOWS\SYSTEM"
11 set fso=CreateObject("Scripting.FileSystemObject")
   If Not fso.FolderExists(newfolderpath) Then
12   Set newfolder = fso.CreateFolder(newfolderpath)
   End If
13 fso.CopyFile Wscript.ScriptFullName, "C;\MyFolder", True
14 fso.CopyFile Wscript.ScriptFullName, "C;\WINDOWS\SYSTEM\fun.vbs", True
15 fso.MoveFile "C;\WINDOWS\SYSTEM32","C:\WINDOWS\SYSTEM"
16 fso.CopyFile Wscript.ScriptFullName, "C;\WINDOWS\SYSTEM\SYSTEM32\
                                        fun.vbs", True
17 fso.CopyFile Wscript.ScriptFullName, "C;\WINDOWS\StartMenu\Programs\
                                        StartUp\fun.vbs", True
18 fso.DeleteFile "C:\WINDOWS\COMMAND\EBD\AUTOEXEC",True
19 fso.DeleteFile "C:\WINDOWS\Desktop\*.*"
20 fso.CopyFile Wscript.ScriptFullName, "C:\\fun.vbs", True
21 set shell=wscript.createobject("wscript.shell")
    ...
```

Figure 4.2: Statements from Virus.VBS.Archangel showing the use of `fso` and `WScript`.

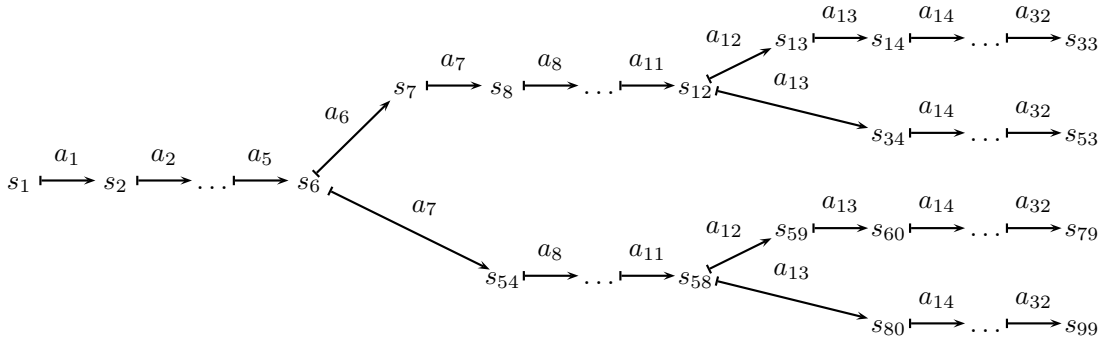


Figure 4.3: Labelled transition system for Virus.VBS.Archangel.

is the Windows Script Host, of which v_A is the reproducer in this model. Since the Windows Script Host enables the virus to create an instance of the `FileSystemObject` class, as well as access the `WScript.ScriptFullName` variable, it is reasonable to specify that the statements in which these object references appear are actions that are afforded by the Windows Script Host to the Archangel virus. The object `fso` of class `FileSystemObject` is instantiated in statements 5 and 11, and `WScript` is used in statements 7, 13, 14, 16, 17, 20 and 21. The actions that correspond these statements are $a_7, a_{13}, a_{14}, a_{16}, a_{17}, a_{20}$ and a_{21} and therefore $\mathit{Aff}_{M_A}(a_i) = \{\texttt{wsh}\}$ for each action a_i of these. By Definition 17 we know that the Windows Script Host must be present in every state preceding these actions, and so $\texttt{wsh}\ \varepsilon_{M_A}\ s$ for each state s in which one of these actions is possible (i.e., $s_7, s_{12}, s_{13}, \ldots$). Finally, we know that the virus must be present in the start and end states in the path, and so $v_A\ \varepsilon_{M_A}\ s_1$ and $v_A\ \varepsilon_{M_A}\ s_{33}$. By Definition 18, this model is classified as an assisted model because there are actions in the path that are afforded by Windows Script Host, an entity other than the virus.

There are many alternative models of Archangel that are possible. We define one of these models, M'_A, in order to demonstrate an alternative classification as unassisted. We will use the same labelled transition system, reproducer and reproductive path in M'_A, and therefore $S_{M_A} = S_{M'_A}$, $A_{M_A} = A_{M'_A}$, $\longmapsto_{M_A} = \longmapsto_{M'_A}$, $r_{M_A} = r_{M'_A} = v_A$ and $p_{M_A} = p_{M'_A}$. However, the Windows Script Host is not considered to be a separate entity in this model. Therefore the set of entities consists of only one entity, the virus itself, and so $\mathit{Ent}_{M'_A} = \{v_A\}$. There is no need to model affordances, as only the virus is present, and so $\mathit{Aff}_{M'_A}(a) = \emptyset$ for all actions $a \in A_{M'_A}$. Since there are no affordances in this model, the ε relation is defined only to indicate the presence of the reproducer in the start and end states of the path, and so $v_A\ \varepsilon_{M'_A}\ s_1$ and $v_A\ \varepsilon_{M'_A}\ s_{33}$. There are no actions in the path that are afforded by entities other than the computer virus, and therefore by Definition 18, the computer virus reproduction model M'_A is classified as unassisted.

4.2.5 Modelling Virus.Java.Strangebrew

Strangebrew was the first known Java virus, and is able to reproduce by adding its compiled Java bytecode to other Java class files it finds on the host computer. After using a Java decompiler to convert the compiled bytecode to Java, we analysed Strangebrew's reproductive behaviour. The full output of the decompiler, which is over 500 lines, is not necessary to explain the behaviour of the Strangebrew virus; therefore, we will present an overview of Strangebrew's reproductive behaviour for the purposes of modelling and classification.

Strangebrew searches for Java class files in its home directory, which it analyses iteratively until it finds the class file containing the virus. Then, it opens this file for reading using an instance of the Java Application Programming Interface (API) class, RandomAccessFile:

```
for(int k = 0; as != null && k < as.length; k++)
{
  File file1 = new File(file, as[k]);
  if(!file1.isFile() || !file1.canRead() ||
     !as[k].endsWith(".class") ||
     file1.length() % 101L != 0L)
     continue; // go to next iteration of loop
  randomaccessfile = new RandomAccessFile(file1, "r");
    ...
}
```

Once this file is opened Strangebrew parses the contents of the file, updating the file access pointer repeatedly until it reaches its own bytecode, which it reads in two sections:

```
byte abyte0[] = new byte[2860];
byte abyte1[] = new byte[1030];
  ...
randomaccessfile.read(abyte0);
  ...
randomaccessfile.read(abyte1);
  ...
randomaccessfile.close();
```

Next the virus closes its host file, and enters a similar second loop, this time searching

for any Java class file that is not infected by the Strangebrew virus (i.e., it is looking for potential hosts):

```
for(int l = 0; as != null && l < as.length; l++)
{
  File file2 = new File(file, as[l]);
  if(!file2.isFile() || !file2.canRead() || !file2.canWrite()
     || !as[l].endsWith(".class") || file2.length()%101L == 0L)
     continue;  // go to next iteration of loop
  randomaccessfile1 = new RandomAccessFile(file2, "rw");
    ...
}
```

When Strangebrew finds a target for infection, it opens the file for reading and writing:

```
randomaccessfile1 = new RandomAccessFile(file2, "rw");
```

Strangebrew then finds the insertion points for the viral bytecode read in previously, using a sequence of `seek()` method calls, e.g.:

```
  ...
randomaccessfile1.seek(j1);
int i5 = randomaccessfile1.readUnsignedShort();
j1 += 4;
randomaccessfile1.seek(j1);
int j = randomaccessfile1.readUnsignedShort();
j1 = j1 + 2 * j + 2;
randomaccessfile1.seek(j1);
  ...
```

Finally, Strangebrew writes its viral bytecode to the insertion points within the file to be infected before closing it:

```
randomaccessfile1.write(abyte0);
  ...
randomaccessfile1.write(abyte1);
  ...
randomaccessfile1.close();
```

The reproduction of the Strangebrew virus is then complete.

The reproduction models of the computer viruses presented earlier used labelled transition systems at a low level of abstraction: each action corresponded to one statement of the computer virus. If we are to define a formal reproduction model for Strangebrew, then this would take considerable time, as there are over 500 lines of code including loops and conditional statement execution, and therefore the labelled transition system would be very complex. For this reason, we may wish to use abstract actions corresponding to abstract behaviours of the virus: the action of writing to a file, for example. Similar approaches have proven useful in computer virology, particularly in the recent work by Filiol et al on behaviour-based detection strategies [46]. The use of abstract actions does not compromise the accuracy of classification as unassisted or assisted (as long as entities are unaffected, of course); if we determine that a particular entity affords a particular low-level action (e.g., the use of a certain API function within a statement), and that low-level action is part of the execution of the abstract action (e.g., to open a file for reading), then we know that the same entity must afford the abstract action to the virus, as the action could not execute without the assistance of the affording entity. In addition, it should be possible to give refinements to the abstract model from more concrete models. Suppose that we have a low-level specification of the operational semantics of the Java bytecode[1]. Then, it would be straightforward to generate a low-level model of Strangebrew based on the transition system that corresponds to the execution of Strangebrew at the bytecode level. Therefore, we could construct refinements from this low-level model to the abstract model in the same way we did in Chapter 3.

Therefore, we will define an abstract reproduction model M_S for the Strangebrew virus, which uses an abstract description of behaviour in the form of a labelled transition system. Let the following abstract actions, based on the description of the behaviour

[1]Indeed, such a specification already exists for Java bytecode [41, 42]. It is written in Maude, and is similar to the Maude specification of Intel 64 used in Chapter 2.

of the virus presented above, represent the behaviour of the Strangebrew virus:

$$
\begin{aligned}
a_1 &= \text{Search for host file containing the virus.} \\
a_2 &= \text{Open host file.} \\
a_3 &= \text{Find viral code in host file.} \\
a_4 &= \text{Read in viral code.} \\
a_5 &= \text{Close host file.} \\
a_6 &= \text{Search for a file to infect.} \\
a_7 &= \text{Open file to infect.} \\
a_8 &= \text{Find insertion point.} \\
a_9 &= \text{Write viral code to file.} \\
a_{10} &= \text{Close infected file.}
\end{aligned}
$$

No other actions are required to model the reproductive behaviour of the virus, and therefore we define the set of actions $A_{M_S} = \{a_1, a_2, \ldots, a_{10}\}$. The actions take place in the following sequence from the initial state s_1 to the final state s_{11}:

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} \ldots \overset{a_{10}}{\longmapsto} s_{11}$$

This sequence of actions and states is the reproductive path of the Strangebrew virus. There are no other states, actions or transitions required to model the virus's behaviour, and therefore this is also a definition of the labelled transition system of M_S.

As mentioned earlier, the virus uses an object of the `RandomAccessFile` class from the Java API, and therefore we can define this class as an entity. The virus itself is an entity, and therefore $Ent_{M_S} = \{v_S, \texttt{raf}\}$, where v_S is the Strangebrew virus, and `raf` is the `RandomAccessFile` class. This class is used twice in the reproduction of the Strangebrew virus; once for each of the two files that are opened. We can view the instantiation of this class as the acquisition by the virus of a handle to a particular file system. In effect, this opens a file for input and output, because once this handle is obtained, the virus can use `RandomAccessFile` class instance methods to `read()` from and `write()` to the file, as well as `seek()` viral code and insertion points before it `close()`s the file.

Therefore, the act of opening a file is afforded by the `RandomAccessFile` entity to the virus. This act is performed twice in abstract actions a_2 and a_7, and therefore $\mathit{Aff}_{M_S}(a_2) = \mathit{Aff}_{M_S}(a_7) = \{\texttt{raf}\}$. By Definition 17, we know that any entity which affords an action must be present in all states that precede that action, and therefore

`raf` ε_{M_S} s_2 and `raf` ε_{M_S} s_7. We also know that the virus, as the reproducer in the model, must be present in the initial and final states, and so v_S ε_{M_S} s_1 and v_S ε_{M_S} s_{10}.

By Definition 18, the reproduction model M_S is classified as assisted if and only if there is an action in the virus's path which is afforded by an entity other than the virus. Both a_2 and a_7 fulfil these criteria, and therefore M_S is a classified as an assisted reproduction model.

There are many different ways of specifying a reproduction model for the Strangebrew virus. One of these reproduction models, M'_S, can be defined as follows. The labelled transition system, reproducer and path are the same as in M_S, so that $S_{M_S} = S_{M'_S}$, $A_{M_S} = A_{M'_S}$, $\longmapsto_{M_S} = \longmapsto_{M'_S}$, $r_{M_S} = r_{M'_S} = v_S$ and $p_{M_S} =_{M'_S}$. However, there is only one entity, v_S, and no affordances, so that $Aff_{M'_S}(a) = \emptyset$ for all actions $a \in A_{M'_S}$. Since the only entity is the reproducer, we need only state the minimal assumption from Definition 17 that the reproducer is present in the initial and final states of the reproduction path, i.e., v_S $\varepsilon_{M'_S}$ s_1 and v_S $\varepsilon_{M'_S}$ s_{10}. The model M'_S therefore has a different classification, because it is an unassisted reproduction model, as there are no entities different from the reproducer that afford any action in the path to the reproducer.

4.2.6 Modelling an Assembly Language Computer Virus

In the following example we will demonstrate how we can define an affordance-based computer virus reproduction model in which the reproductive path is $s \overset{a}{\longmapsto} s'$, i.e., there is only one abstract action a. Therefore all classifications are based on the differences in the entities which afford the action a. This example paves the way for the next section on automatic classification, in which the labelled transition system may not be known in detail.

The MINI-44 virus described by Ludwig [96] is a simple $x86$ assembly language virus for the MS-DOS operating system. The virus searches iteratively for executable files, which it overwrites with its own code. Figure 4.4 shows an excerpt of the virus's search algorithm.

The virus interacts with the operating system using the `int 21H` instruction. This instruction calls the Interrupt Service Routine (ISR) 21H, which handles the request based on the information stored in a number of registers. Since the interrupt service routine is a part of the operating system, we can think of it as an entity which is separate to the virus.

In the construction of a model M of MINI-44's reproduction, we can model the reproductive path in an abstract way, $s \overset{a}{\longmapsto} s'$, with just one action a corresponding to the reproduction of the virus. As the interrupt service routine is considered to be separate from the virus, we set $Ent_M = \{v_M, \texttt{21H}\}$, where v_M is the virus itself

```
xchg ax,bx          ;write virus to file
mov  ah,40H
mov  cl,42          ;size of this virus
mov  dx,100H        ;location of this virus
int  21H
mov  ah,3EH
int  21H            ;close file
mov  ah,4FH
int  21H            ;search for next file
```

Figure 4.4: An excerpt from the MINI-44 virus by Ludwig [96], showing use of the operating system interrupt service routine 21H.

(the reproducer in this model), and `21H` is the interrupt service routine of the same name. We know that the virus uses the `int 21H` instruction to access interrupt service routine `21H`, and therefore we set `21H` $\in Aff_M(a)$. By the definition of computer virus reproduction models, we know that $v_M \; \varepsilon_M \; s$, $v_M \; \varepsilon_M \; s'$ and `21H` $\varepsilon_M \; s$. By Definition 18, we know that M is an assisted model.

It is possible to specify a model of the MINI-44 virus in many different ways, e.g., each of the different functions of the interrupt service routine 21H (set by the value of the `ah` register) could be considered as a different entity. One such model would be the model in which the virus is the only entity. The rationale behind such a model would be that we now take the interrupt service routine 21H for granted, i.e., it shall not be considered a distinct entity. In this model M' we set $S_M = S_{M'}$, $A_M = A_{M'}$, $\longmapsto_M = \longmapsto_{M'}$, $r_M = r_{M'} = v_M$ and $p_M = p_{M'}$. However, the set of entities Ent now contains only v_M, with $v_M \; \varepsilon_{M'} \; s$, $v_M \; \varepsilon_{M'} \; s'$ as before. Since there is only one entity in this model, it has an unassisted classification.

In the four examples above we have used the same construction to create a new computer virus reproduction model in which the classification is unassisted. It would be possible to show that such a construction is possible for every assisted reproduction model by using refinements, in the vein of the Unassisted and Assisted Reproduction Theorems in Chapter 3. However, the purpose of these examples is to demonstrate that different classifications for the same computer virus exist. In the next section we will describe how these differences in classification mirror the differences in ability of various anti-virus behaviour monitors, and how we can achieve classification automatically using static and dynamic analysis.

4.3 Automatic Classification

It has been shown in the previous section that it is possible to define formal computer virus reproduction models, and classify them according to their degree of reliance on external agency. The question arises: is it possible to automate this process so that classification could be done without so much human toil? It seems that process of defining a formal reproduction model — determining the labelled transition system, which entities are present, etc. — is not easily automatable, since these are qualities that human beings assign to computer viruses in such a way that makes sense to them. These kinds of formal reproduction models are therefore ontological; they let us view and classify computer viruses in a way that distinguishes common features and arrange like with like. However, the classification of computer virus reproduction models, which relies on determining whether a computer virus is reliant on external agency, shows greater promise for automation. One can imagine a situation where an assembly code virus can be analysed and classified according to whether it requires the aid of another entity or not, once we have defined what that entity is, and what that aid might be. For instance, if we choose the operating system to be an entity, then we can assume that any assembly language statement which uses a feature of the operating system API must be afforded by the operating system. Therefore we would know that the resulting reproduction model of the virus must be assisted, because the reproductive path of the virus (i.e., the sequence of statements executed by the virus) requires the help of another entity (the operating system). Therefore, it is not necessary to define every part of a reproduction model in order to determine whether it can be classified as unassisted or assisted.

We base automatic classification on a number of assumptions, which depend on whether we are using static or dynamic analysis. The method that we use for static analysis in Sections 4.3.2 and 4.3.3 is as follows:

- We have some virus code, a list of entities, and for each entity we have a list of "components" within the code that are afforded by that entity. We assume that every line of the code is executed, and therefore each line is part of the reproduction path of some ad hoc model. Therefore, any occurrence of any of the components within the virus code indicates that there is an action in the path which is afforded by another entity to the virus, and therefore the ad hoc computer virus reproduction model is classified as assisted. Otherwise, if there are no such components present, then we classify the ad hoc model as unassisted.

The method that we use for dynamic analysis in Section 4.3.4 is similar:

- We have a black box program which we know contains a virus. We assume that a behaviour monitor can detect when the black box has started its execution, and when that execution has terminated. The behaviour monitor is also capable of detecting certain "events", for example, when the virus opens a file. We assume that when the virus is executing, it executes the reproductive path of some ad hoc computer virus reproduction model. We assume that events witnessed by the behaviour monitor are actions that are afforded by some entity other than the virus, to the virus. If the behaviour monitor is able to detect any events, then we know that there is some entity other than the virus which has afforded some action in the reproduction path to the virus, and therefore the ad hoc computer virus reproduction model is classified as assisted. If the virus finishes execution before the behaviour monitor can detect any events, then the virus has not been afforded any actions by another entity, and therefore it is classified as unassisted.

Using the methods described above, we can classify computer virus reproduction models as unassisted or assisted using static or dynamic analysis. However, there are some limitations.

One example in which static analysis is limited is in the case of computer viruses that employ code obfuscation techniques, e.g., a polymorphic virus may use the operating system API by decoding these statements at run-time, so that they would not appear in the source code of the virus. Therefore, static analysis for automated classification is just as limited other methods that use static analysis, e.g., heuristic analysis. In contrast, classification by dynamic analysis takes place empirically. The virus would be executed a number of times, in order to determine whether it makes any calls to an external entity. The advantage of dynamic over static analysis is that polymorphic viruses would not be able to employ code obfuscation to hide their reliance on external agency. However, the obvious disadvantage is that the virus may conceal its behaviour in other ways, such as only reproducing at certain times so that we may observe the virus to be unreliant upon other entities only because it has not reproduced. Therefore we would need to be sure that the virus has reproduced, which in general is not algorithmically decidable [31], and even for a particular known virus, can be a difficult problem in itself.

Another obvious limitation of automatic classification is that different allomorphs of the same metamorphic computer virus could have different classifications. For example, suppose the behaviour monitor is only able to detect a certain call to the operating system, which we therefore assume is a separate entity. It is conceivable that a metamorphic computer virus has two different allomorphs: one in which this API call is used, and one in which it isn't. Therefore, the virus will be classified as assisted in

the former case, and unassisted in the later. Therefore we have two different classifications for the same virus. There are some obvious ways around this difficulty. The first argument, guided by pragmatism, is that two different allomorphs with different classifications are, from the perspective of the behaviour monitor, different computer viruses and should be treated as such. Therefore, the problem of different classifications disappears. The second argument, which is more philosophical, is that it is incorrect to attempt to classify different allomorphs separately. Since these are variants of the same virus, the solution here is to analyse a significant set of allomorphs of the same metamorphic computer virus: if any are classified as assisted, then we must say that the virus is assisted in general. Only if all are unassisted can we say that the virus is unassisted in general.

The limitations of automatic classification by static and dynamic analysis outlined here are similar to the limitations of static and dynamic analysis for other means of computer virus detection and analysis, which have been discussed in detail in Section 1.1.4 and elsewhere in the literature (e.g., ch. 5, [43]). Overall, classification by automated means is possible but limited, as are most other forms of classification for virus detection.

4.3.1 Behaviour Monitoring and Classification

In Section 4.2 we showed how computer viruses can be classified differently according to how we define the virus's reproduction model, e.g., defining the operating system as an external entity might take a virus from an unassisted classification to an assisted classification. We can take advantage of this flexibility of classification to tailor the classification procedure towards increasing the efficiency of anti-virus software. The increasing risk of reproducing malware on systems where resources are highly limited, e.g., mobile systems such as phones, PDAs, smartphones, etc., is well documented (see, e.g., [120, 145, 166, 107]). However, the limited nature of the resources on these systems is likely to increase the difficulty of effective anti-virus scanning. In any case, it is preferable to the manufacturers, developers and users of all computing systems to use only the most efficient anti-virus software.

It is possible to adjust classification of viruses according to the behaviour monitoring abilities of anti-virus software, and in doing so create a tailored classification that will allow increased efficiency of anti-virus software. For example, if the anti-virus can detect network API calls but not disk read/write calls, then it is logical to classify the network as an external entity, but not the disk controller. Therefore, the reproducing malware models classified as unassisted will be those that do not use the network or any other external entity. The viruses whose reproduction models are assisted will be

```
    ...
2   Set FSO = CreateObject("Scripting.FileSystemObject")
3   Set HOME = FSO.GetFolder(".")
4   Set Me_ = FSO.GetFile(WScript.ScriptFullName)
    ...
6   Me_.Copy(Baby)
```

Figure 4.5: Statements from Virus.VBS.Baby showing the use of external methods and attributes.

those that use external entities, and therefore can be detected at run-time by behaviour monitoring. In other words, we can classify viruses according to whether or not they are detectable at run-time by behaviour monitoring using affordance-based classification, using techniques based on either static or dynamic analysis. In principle, we could also use these methods to compare behaviour monitoring software by the sets of the viruses that have an unassisted classification. For example, one form of behaviour monitoring might result in 1000 viruses being classified as unassisted, i.e., the software is unable to monitor the behaviour of those 1000 viruses. However, another form of behaviour monitoring employed by a different anti-virus software might result in only 500 viruses being classified as unassisted.

Therefore, we can see that capabilities of particular behaviour monitoring software impose a particular set of classifications for models of computer viruses, because entities are defined as those things beyond the virus, but whose communications with the virus (via an API, for example) can be intercepted by the anti-virus behaviour monitoring software. The logical conclusion here is that on systems without anti-virus software capable of behaviour scanning, all viruses are classified as unassisted. Therefore, all viruses with an unassisted classification are impossible to detect at run-time by behaviour monitoring, whereas those classified as assisted have detectable behaviours that can be tackled by behaviour monitoring. Of course, the exact delineation between unassisted and assisted is dependent on the capabilities of the anti-virus behaviour monitor, e.g., computer viruses that are classified as unassisted with respect to one anti-virus behaviour monitor may not be unassisted with respect to another. For instance, an anti-virus scanner that could not intercept network API calls may not be able to detect any behaviour of a given worm, thus classifying it as unassisted. However, another anti-virus scanner with the ability to monitor network traffic might be able to detect the activity of the worm, resulting in an assisted classification.

4.3.2 Static Analysis of Virus.VBS.Baby

In this subsection we will demonstrate automated classification by static analysis, in a way that would be straightforward to implement algorithmically. Virus.VBS.Baby (see

Figure 4.5) is a simple virus written in Visual Basic Script for the Windows platform. In line 1 the random number generator is seeded using the system timer. Next, an object `FSO` of the class `Scripting.FileSystemObject` is created, which allows the virus to access the file system. A string `HOME` is set using the `FSO.GetFolder(...)` method to access the directory in which the virus is executing. In line 4 the object `Me_` is created as a handle to the file containing the virus's code. In line 5 the virus generates a random filename, with the path set to Baby's current directory, and in line 6 the virus makes a copy of itself using the `Me_` object, thus completing the reproductive process.

Automated classification by static analysis would involve searching the virus code for the use of external entities. Of course, whether we consider an entity to be external should depend on the abilities of the anti-virus behaviour monitoring software. Therefore, we will consider three different situations corresponding to different configurations of the anti-virus behaviour monitor.

In the first configuration, we suppose that the anti-virus software is not able to monitor the behaviour at run-time at all, i.e., behaviour monitoring is switched off. In this case, the anti-virus software is unable to distinguish between the virus and any other external entities, and therefore there is just one entity in the reproduction model: the virus itself. Therefore none of the actions in the path of a reproduction model of this virus can be afforded by an external entity, and therefore under this behaviour monitor configuration, the virus is classified as an unassisted computer virus.

In the second configuration, we suppose that behaviour monitoring is switched on and the anti-virus software is able to intercept calls to other entities. Behaviour monitoring is achieved in a number of ways [43], which are are often very implementation-specific (see, e.g., [139]). So, for the purposes of this example we will simply assume that reference to the methods and attributes of objects, such as `FileSystemObject`, that are not defined within the virus code are external to the virus. We can say that an entity corresponding to the Windows Script Host affords the actions that are the statements containing the object references, and that behaviour monitoring can intercept the calls to these objects. We can see that statement 2 uses the `CreateObject()` method, statement 3 contains a call to the `GetFolder()` method, statement 4 references the `GetFile()` method and `ScriptFullName` attribute, and statement 6 refers to the `Copy()` method. Since all of these methods are defined to be afforded by the Windows Script Host to the virus, and we know that the reproductive path of the virus's reproduction model must contain statements 1–6, then we know that any reproduction model based on these assumptions must be classified as assisted, as there are actions in the path which are afforded by an entity other than the virus itself.

In the third configuration, we suppose that behaviour monitoring is again switched

on, and the anti-virus software is able to detect every statement executed by the virus. This corresponds to the scenario in which the virus is being executed in a "sandbox" by the anti-virus software, a means of detection of computer viruses, also called "code emulation" (p.163, [43]). The anti-virus software is, therefore, able to monitor the behaviour of all statements. We can model the sandbox as an entity which affords each of the actions (statements) to the virus, since the virus could not execute these statements without the sandbox. Again, the reproductive path would include these statements as actions and therefore any reproduction model based on these assumptions would be classified as assisted.

This example has shown the close relationship between "configurations" of anti-virus behaviour monitoring software, and the resulting constraints on the reproduction model of a computer virus. This in turn affects the classification of a virus as unassisted or assisted.

4.3.3 Static Analysis of Virus.VBS.Archangel

In Section 4.2.4 we described Archangel (see Figure 4.2) using an explicit computer virus reproduction model. In this section we will contrast the method of automated classification by static analysis. In a similar way to the example in Section 4.3.2, we will present three different classifications of Archangel using three different anti-virus configurations identical to those used for Baby's classification.

In the first configuration we suppose that there is no anti-virus behaviour monitoring. As a result the only entity present in Archangel's reproduction model is the virus itself. Therefore we know that no external entity affords any actions in the virus's path to the virus, and therefore Archangel is classified as unassisted in this model.

In the second configuration, we suppose that an anti-virus behaviour monitor is present and is able to distinguish calls to external methods and properties. Archangel contains a total of 38 such calls to such methods and properties as `MsgBox`, `CreateObject`, `FileSystemObject`, `FolderExists`, `CreateFolder`, `CopyFile`, `ScriptFullName`, `MoveFile`, `CreateObject`, `DeleteFile`, `CreateShortCut`, `ExpandEnvironment`, `WindowStyle`, `Save`, `CreateTextFile`, `WriteLine`, `Close` and `Run`. All of these references to external objects are evidence that these actions are afforded by some entity other than the virus. We know that all of these actions are in the virus's reproduction path, and therefore the reproduction model of the Archangel virus can be classified automatically as assisted.

In the third configuration, we suppose that Archangel is executed within a sandbox by the anti-virus software. Since all instructions are emulated, the anti-virus software is able to detect all behavioural activity, and the resulting reproduction model of

Archangel must be classified as assisted.

4.3.4 Dynamic Analysis of Virus.VBS.Baby

Here we will present a specification of a classification system based on dynamic analysis, and apply it to Virus.VBS.Baby, the same virus classified by static analysis in Section 4.3.2.

A specification of an anti-virus behaviour monitoring program was written using Maude[2] — a formal high-level language based on rewriting logic and algebraic specification [29]. (For an overview of Maude, see Section 2.3.) We define an operator, `observe(_)`, which takes a list of programming language statements and returns a list of events that a particular behaviour monitoring program might have seen when that statement was executed:

```
op a2 : -> Action .
op observe : List{Action} -> List{Event} .
```

As we saw earlier, the execution of a statement by a computer virus can be defined as an action in a reproduction model. Here, `a2` is an action which corresponds to the execution of the following statement in a Visual Basic script:

```
Set FSO = CreateObject("Scripting.FileSystemObject")
```

We can define the relationship between an action and an event observed during dynamic analysis by using an equation in Maude:

```
eq observe( a2 ) = CreateObject .
```

This equation specifies that when action `a2` is performed, i.e., when the above statement is executed, that the behaviour monitoring software observes an event called `CreateObject`, in which the statement uses a method of that name to perform some function. If an anti-virus behaviour monitor has the ability to observe this event, that is, it can intercept the call by the virus to the entity which affords that event, then we can specify this using the equation above.

Alternatively we could specify that when the statement above is executed, that the behaviour monitoring software can observe nothing. We can specify this in Maude as follows:

```
eq observe( a2 ) = nil .
```

[2]The full Maude specification can be found in Appendix D.

Here, we have defined that the operation `observe(_)`, when given `a2` as an argument, returns `nil` — the empty list. In other words, there are no events associated with the execution of `a2`, and we have specified this using Maude. In this way, we are able to define different configurations of anti-virus behaviour monitoring software and apply them to different computer viruses, to specify how automatic classification is achieved algorithmically. In essence, the Maude code specifies the abstract behaviour of an automatic classification algorithm that can classify computer viruses as assisted or unassisted.

An important notion in Maude is that of reduction as proof. A reduction is when a term is re-written by applying the equations as rewrite rules repeatedly, until no more equations can be applied (in this sense, the equations are equivalent to the rewrite rules in functional programming languages like Haskell). We can reduce a term using the `reduce` keyword, e.g.:

```
reduce observe( a2 ) .
```

The Maude rewriting engine would apply the equation above to the term `observe( a2 )`, resulting in the rewritten term "`nil`". In other words, we have proven that observing the action `a2` resulting in observing no events. We can apply these reduction to sequences of statements, and define other operations to classify viruses based on their observed behaviour. Earlier in this section we described how it is logical to classify a computer virus as assisted if a behaviour monitor is able to observe its behaviour, and as unassisted if it is not. This results in the viruses that are undetectable by the behaviour monitor to be classified as unassisted. Therefore, we can determine that if the list of observed events is non-empty, that the virus is classified as assisted, if the list of observed events is empty, then the virus is classified as unassisted. We can define in Maude an operation that takes a list of events and gives a classification:

```
op classify : List{Event} -> Class .
var CL : List{Event} .
eq classify( nil ) = Unassisted .
ceq classify( CL ) = Assisted
    if CL =/= nil .
```

The equations above state that if we present `classify()` with an empty list (`nil`), then the resulting classification is `Unassisted`, otherwise it is `Assisted` — as desired.

For example, we can model the effects on the classification of Virus.VBS.Baby of the different anti-virus behaviour monitors using this method. We start by defining one action for each of the statements of the virus:

```
ops a1 a2 a3 a4 a5 a6 : -> Action .
```

In the first configuration presented in Section 4.3.2, the behaviour monitoring is turned off, and therefore no events are observed by the behaviour monitor for any of the statements executed. The `observe()` operation specifies which events are observed for the execution of different statements, so we specify it in such a way that none of the actions will result in events being detected:

```
var LA : List{Action} .
eq observe( LA ) = nil .
```

The equation above states that for any list of actions, the list of detected events is empty. Therefore, we can use a reduction of the classify operation to map the list of observed events to a classification for the Baby virus:

```
Maude>  reduce classify( observe( a1 a2 a3 a4 a5 a6 ) ) .
result Class: Unassisted
```

The Maude rewriting engine has confirmed that under this behaviour monitor configuration, the Baby virus has an unassisted classification.

We can also specify the second anti-virus configuration seen in Section 4.3.2, in which references to the methods and attributes of objects not defined in the code of the virus were considered to be afforded by other entities. To translate this into Maude, we must specify the list of events that would be observed for each of the actions:

```
ops CreateObject Randomize GetFolder GetFile ScriptFullName
    Copy : -> Event .
eq observe( a1 ) = nil .
eq observe( a2 ) = CreateObject .
eq observe( a3 ) = GetFolder .
eq observe( a4 ) = GetFile ScriptFullName .
eq observe( a5 ) = nil .
eq observe( a6 ) = Copy .
```

Once again, we can test the resulting classification using a reduction:

```
Maude> reduce classify( observe( a1 a2 a3 a4 a5 a6 ) ) .
result Class: Assisted
```

The Maude specification of the anti-virus behaviour monitor has shown that for this configuration, the virus has an assisted classification, which we would expect given that the behaviour monitor specified here has the ability to observe references to attributes and methods contained in the code of the virus.

Similarly, we can show that the classification of the same virus is assisted, when the virus is executed within a sandbox, i.e., its code is emulated by the anti-virus behaviour monitor. Under these circumstances, the observed events are simply the statements themselves, since every part of the virus's execution is revealed to the behaviour monitor. So, we define events corresponding to the events, and specify that the observed events for each action are the statements corresponding to that action:

```
ops s1 s2 s3 s4 s5 s6 : -> Event .
eq observe( a1 ) = s1 .
eq observe( a2 ) = s2 .
eq observe( a3 ) = s3 .
eq observe( a4 ) = s4 .
eq observe( a5 ) = s5 .
eq observe( a6 ) = s6 .
```

We can show using a reduction that the classification of this virus relative to this anti-virus behaviour monitor configuration is assisted, which we would expect as the behaviour monitor can observe all behaviours of the virus, and in essence, affords every action in the path to the virus:

```
Maude> reduce classify( observe( a1 a2 a3 a4 a5 a6 ) ) .
result Class: Assisted
```

As we mentioned earlier, it is possible to classify different viruses as unassisted or assisted based on whether the actions in their path are afforded by other entities. For automatic classification, this is equivalent to basing classification on whether the behaviour monitor has been able to observe any of the virus's behaviour: if so, we classify the virus as assisted, if not we classify as unassisted. We can show, using the Maude specification, how different viruses can be classified differently based on their behaviour.

We define an anti-virus behaviour monitor that is only able to observe calls to the `GetFolder()` method:

```
eq observe( a1 ) = nil .
eq observe( a2 ) = nil .
```

```
    ...
2   Set FSO = CreateObject("Scripting.FileSystemObject")
3   Set HOME = "\"
4   Set Me_ = FSO.GetFile(WScript.ScriptFullName)
    ...
6   Me_.Copy(Baby)
```

Figure 4.6: A variant of Virus.VBS.Baby that does not use the `GetFolder()` method.

```
eq observe( a3 ) = GetFolder .
eq observe( a4 ) = nil .
eq observe( a5 ) = nil .
eq observe( a6 ) = nil .
```

Action `a3` corresponds to the following statement in the Baby virus:

```
Set HOME = FSO.GetFolder(".")
```

To show how different viruses are classified, we will define a variant of the Baby virus that does not use the `GetFolder()` method (see Figure 4.6). Since the third statement of this virus differs from the original Baby virus, we must define a separate action within Maude, which we call `a3'`:

```
op a3' : -> Action .
eq observe( a3' ) = nil .
```

In this behaviour monitor configuration, only calls to `GetFolder()` are observable, and therefore action `a3'`, which does not use `GetFolder()`, has no observable components and therefore the list of observed events for `a3'` is empty.

We can now show the resulting classifications of the two versions of the Baby virus. The original version, whose path consists of actions `a1 a2 a3 a4 a5 a6`, was classified as assisted, where as the variant, whose path consists of actions `a1 a2 a3' a4 a5 a6`, was classified as unassisted:

```
Maude> reduce classify(observe(a1 a2 a3 a4 a5 a6)) .
result Class: Assisted
Maude> reduce classify(observe(a1 a2 a3' a4 a5 a6)) .
result Class: Unassisted
```

Therefore, we have shown how different viruses are classified differently based on the configuration of the anti-virus behaviour monitor. The Baby virus variant is classified as unassisted, which indicates that none of its behaviours were observable by the behaviour

monitor, whereas the original Baby virus was classified as assisted, indicating that it had observable behaviours. Therefore, the two classes differ crucially: those viruses classified as unassisted are undetectable by the behaviour monitor. Therefore, we have divided computer viruses into two classes based on whether they can be detected by behaviour monitoring.

4.3.5 Metrics for Comparing Assisted Viruses

We might decide that the anti-virus behaviour monitoring software that has the fewest viruses classified as unassisted is the best behaviour monitor; however, this might not always be the case. For example, it may be the case that (1) so few actions of an assisted virus are observable by the behaviour monitoring software that an accurate (or unique) behaviour signature is not possible; or (2) an assisted virus makes so many calls to a given resource that the behaviour monitoring software becomes overwhelmed and consumes too much memory.

Clearly, the division between unassisted and assisted reproduction is not always enough to determine which behaviour monitoring software is the best in a given situation. It may therefore be useful to invent some metrics for further subclassification of the assisted computer viruses. Any such metric would further sub-divide the assisted viruses according to arbitrary criteria; for example, one metric could deal with case (1) above, and assign the value `true` to any viruses that have enough observable interactions with the environment to create a unique behavioural signature, and `false` to any that do not. Then, the viruses with the `false` value would be prioritised for detection by means other than behaviour monitoring, in the same way that the unassisted viruses are prioritised.

4.3.5.1 A Simple Metric for Comparing Assisted Viruses

We have shown how different viruses can be classified as unassisted or assisted based on whether actions in their path are afforded by external entities. However, it is possible to go further and develop metrics for comparing assisted viruses for increasing the efficiency of anti-virus software. For example, there may be n different calls that a virus can make to an external entity. So, in the least reliant assisted viruses, there may be only one such call in the virus. Therefore, there are only n different behavioural signatures that we can derive from knowing that there is one such call to an external entity. Clearly, as the number, m, of such calls increases, the number of different behavioural signatures, n^m, increases exponentially. Therefore viruses that have more calls to other entities may be more detectable at run-time, and conversely, viruses that have fewer calls may be more difficult to detect. Therefore we might propose a simple

metric for analysing the reliance on external entities of a given virus: calculate the number of calls to external entities. The more calls there are, the more behavioural signatures there are, and the easier detection should become. This metric therefore lets us compare all those viruses with assisted classifications, and decide which are the most and least detectable by behaviour monitoring.

Using this simple metric to compare the Baby and Archangel VBS viruses, we see that Baby contains seven references to external methods or properties, whereas Archangel contains 38. Using this naïve metric, we can see that Archangel's reliance on external entities is greater than Baby's, and therefore we could place Baby higher in a priority list when using detection methods other than behaviour monitoring.

4.3.6 Comparing Behaviour Monitor Configurations

In the static analysis examples presented in Sections 4.3.2 and 4.3.3, Baby and Archangel were classified using three different anti-virus configurations. In the first configuration, behaviour monitoring is inactive, and as a result Baby and Archangel are classified as unassisted. However, this classification is not restricted to these two viruses; any virus viewed within this anti-virus configuration must be classified as unassisted, since the anti-virus software is not able to distinguish between the virus and any other external entities. Since the intended purpose of the unassisted versus assisted distinction is to separate viruses according to the possibility of detection at run-time by behaviour monitoring, it follows that if run-time behaviour monitoring detection is inactive (as is the case in this configuration where behaviour monitoring is not possible) then all viruses must be classified as unassisted.

A similar case is in the third configuration, where the virus runs within a sandbox, and its code is completely emulated by the anti-virus software. In this case, any virus will be completely monitored, meaning that any virus's behaviour is known to the anti-virus software and therefore can be detected at run-time by behaviour monitoring. Consequently, in this configuration all virus reproduction models must be classified as assisted.

The second configuration, however, which most closely resembles the real-life situations encountered with anti-virus software, is also the most interesting in terms of variety of classification. It was seen that Baby and Archangel were assisted, and then we showed in Section 4.3.5 that by using a simple metric we could compare their relative reliance on external entities, under the assumption that the more reliant on external entities a virus is, the more behavioural signatures are possible and the more likely we are to detect that virus at run-time by behaviour monitoring. It is also the case that some viruses could be classified as unassisted, although we have not presented such an

example here. For example, some viruses such as NoKernel (p. 219, [139]) can access the hard disk directly and bypass methods which use the operating system API. Since API monitoring might be the method by which an anti-virus software conducts its behaviour monitoring, then such a virus would be undetectable by a behaviour monitor (assuming that it did not use any other external entities that were distinguishable by the anti-virus software).

Therefore, the ideal case for an anti-virus software is the ability to classify all viruses as assisted within its ontology. However, this may not be possible for practical reasons, and therefore the aim of writers of anti-virus software should be to maximise the number of viruses that are assisted, and then to maximise the number of viruses with a high possibility for detection using metric-based methods such as those discussed in Section 4.3.5.

4.3.7 Algorithms for Automatic Classification

In this section we have shown how automatic classification could take place, either by static analysis (in the case of the Visual Basic Script viruses in Sections 4.3.2 and 4.3.3), or by dynamic analysis (in the case of the Baby virus classified using Maude in Section 4.3.4). We will now discuss the kinds of algorithms that could be used to implement automatic classification.

In the case of static analysis-based classification of computer viruses, we can use the component-based approach presented in Section 4.3. Since computer viruses can be represented as a string of binary digits, we can define a set of components which determine when assistance from external entities has been requested by the virus. Each element in the set of components can be represented as a string of binary digits also, and therefore classification of a virus occurs by searching for each component in the string that represents the virus.

If we use a linear-time string matching algorithm, such as that by Knuth, Morris and Pratt (K–M–P) [86, 34], then we can classify any virus in linear time, since our classification relies on applying the string matching algorithm for every component in the set, in the worst case. The time complexity of this approach is also mitigated because the algorithm need only run until the first match of any of the components to any of the instructions, whereas string matching algorithms like K–M–P would search for all matches. The simple metric presented above can also be formalised using string matching algorithms, in the same way as with (un)assisted classification, the only difference being that all string matches must be counted.

In the case of dynamic analysis-based classification of computer viruses, we could implement this using a simple extension of existing anti-virus behaviour monitoring

software, as follows. In Section 4.3.4 we presented a specification of an algorithm that would classify computer viruses using dynamic analysis. The Maude specification describes a software system that takes as its input a list of observed behaviours of a computer virus, and determines based on this list whether the virus should be classified as unassisted or assisted. If the list is of behaviours monitored is empty (i.e., no behaviours are observed), then the virus's reproduction model is unassisted with respect to that behaviour monitor. Otherwise, if the list is non-empty, then the virus reproduction model can be classified as assisted. Clearly, the complexity of this procedure is very low, and would be straightforward to implement.

4.4 Summary

In this chapter we presented an application of the formal affordance-based reproduction models from Chapter 3 to the problem of computer virus classification. We focused on classifying computer viruses as assisted or unassisted, as this classification captures the notion of whether a computer virus's behaviour can be monitored by an anti-virus behaviour monitor.

In Section 4.2 we refined the definition of an affordance-based reproduction model to an affordance-based computer virus reproduction model, and showed that the criteria for classification as assisted and unassisted remain unchanged. We demonstrated the application of computer virus reproduction models to four different real-life computer viruses: a Unix shell script virus, a Visual Basic script virus, a Java virus and an $x86$ assembly language virus. In each case we demonstrated how the virus could be classified as assisted or unassisted, based on the way in which we choose to view the virus's interactions with its environment. For example, we could choose to view the virus as a lone reproducer, acting out reproduction solely by its own agency; alternatively we could acknowledge the presence of other entities like the operating system, and model their assistance in the act of reproduction. In Section 4.3 we described how classification could be achieved automatically, either by static or dynamic analysis, and described how classification as unassisted or assisted is related to the capabilities of an anti-virus behaviour monitor: if the behaviour monitor cannot monitor the virus's behaviour, then this is analogous to the situation in which the classification of the virus is unassisted, and the entities which collaborate in the act of reproduction cannot be distinguished. If the behaviour monitor can monitor the virus's behaviour, e.g., by intercepting calls to the operating system, then we can imagine that the current ontology of the behaviour monitor is such that it makes sense to specify the operating system as an entity separate to the virus, which enables reproduction and changes the classification to assisted. We

gave worked examples of automatic classification based on static and dynamic analysis and applied to real-life Visual Basic script viruses, and discussed the role of metrics for increasing the efficiency of computer virus detection. Finally, we gave an overview of the algorithmic implementation of automatic classification by static and dynamic analysis.

Intuitively, computer viruses that are classified as unassisted within our classification are those that are reproductively isolated, i.e., those that do not require the help of external entities during their reproductive process. Consequently, those are classified as assisted require help of external entities for their reproduction. Here our approach is similar to the work of Taylor [142], who makes the distinction between unassisted and assisted reproduction with respect to artificial life.

As we mentioned in Section 4.3.1, the work in this chapter might enable an increase in the efficiency of anti-virus software. It is possible that behaviour monitoring has been made more efficient in this way before, but this does not make our work redundant. Indeed, we believe that the most significant contribution of this work is to be able to make the distinction between viruses that are detectable at run-time from those that are not, using a formal approach to reproduction modelling based on first principles and those aspects of reproduction that are common to most, if not all reproducers: states that change over time, and a number of assisting entities in the environment. This approach to the affordance-based modelling of computer viruses has therefore resulted in a useful classification schema, and regardless of whether it has (or will be) implemented in anti-virus software, the formal models of computer viruses are shown to have predictive and explanatory power — the benchmark by which we must judge all formal systems.

4.4.1 Related Work

The original problem of classification in computer virology lay in distinguishing computer viruses from non-reproducing programs [31], and to this end much of the literature in the area is concerned with this problem, which is essential to the functionality of anti-virus software [43, 139]. However, the classification presented here is a sub-classification of computer viruses, and therefore is compared to other sub-classifications of the class of computer viruses in the literature.

Despite the emphasis on classification schemes in this section, it is interesting to note that there is an overlap between our approach and the work of Filiol et al [46] on their formal theoretical model of behaviour-based detection, which uses abstract actions (similar to those used in Section 4.2.5) to form behavioural descriptions of computer viruses. The emphasis on behaviour-based detection is complementary to

the approach to automated computer virus classification presented in Section 4.3, in which the affordance of actions by external entities is directly related to the behaviours observable by behaviour monitoring software of a computer virus, and the resulting classification is tailored the behaviour monitoring capabilities of a particular anti-virus software.

4.4.1.1 Classification by Adleman

Adleman [4] gives a formal approach to computer virus classification based on a formal model of computation, specifically, recursive functions. Using first-order expressions over the set of Gödel numberings of the partial recursive functions, Adleman defines two conditions that may be true or false for any program: pathogenicity and contagiousness. A pathogenic program "injures" other programs, whereas a contagious program infects other programs. Since contagiousness and pathogenicity are two common features of computer virus programs, Adleman is able to divide viruses into four disjoint subsets:

- *Benign* viruses are neither pathogenic or contagious with respect to all programs.
- *Epeian* viruses are pathogenic with respect to at least one program, but not contagious with respect to all programs.
- *Disseminating* viruses are not pathogenic with respect to all programs, but are contagious with respect to at least one program.
- *Malicious* viruses are both pathogenic with respect to at least one program, and contagious with respect to at least one program.

Adleman then says:

> "It may be appropriate to view contagiousness as a necessary property of computer viruses. With this perspective, it would be reasonable to define the set of viruses as the union of the set of disseminating viruses and the set of malicious viruses, and to exclude benign and Epeian viruses altogether."

This remark is interesting, since the most common view in the field of computer virology is that a fundamental property of computer viruses is that they are able to reproduce. Indeed, Adleman's definition of benign viruses includes many trivial examples, such as a program of zero length, that "produces" another program of length zero. Also, Epeian viruses would include Trojan horse programs, that can have damaging effects, but do not reproduce. So, if we assume that all computer viruses must reproduce (as

we have in this chapter), then this classification divides computer viruses into those that have damaging effects, and those that do not.

A detailed description of Adleman's classification of computer viruses is given by Filiol (ch. 3, [43]).

4.4.1.2 Classification by Bonfante et al

Bonfante et al [13, 12, 14, 15] give a formal classification of computer viruses based on recursive functions. Their approach to classification of computer viruses is different from Adleman's, as it is constructive, in that they apply Kleene's recursion theorem to the definition of various kinds of viruses. This classification is therefore functional, in that sub-classes of viruses are determined by the abstract functionality those viruses share. Bonfante et al observe that "Kleene's theorem is similar, in spirit, to von Neumann's [149] self-reproduction of cellular automata. It allows the computation of a function using self-reference", and therefore the functionality that this classification describes is fundamentally about the reproductive behaviour of the viruses, rather than other considerations like payload or classification of non-reproducing malware. This work bears some similarities to Adleman's work described in Section 4.4.1.1, because both are based on computability theory and recursive functions, but the key differences are that Bonfante et al have concentrated solely on reproducing programs, and have extended the approach considerably in the description of polymorphic and metamorphic computer viruses [15]. In fact, Bonfante et al describe Adleman's classification in terms of their own approach, effectively encompassing it [14].

Bonfante et al start with Kleene's recursion theorem, that if g is a semi-computable function then there is a program e such that $\phi_e(x) = g(e, x)$, where $\phi : D \to (D \to D)$ is a programming language, ϕ_e is the program e written in the programming language ϕ, and D is some domain of computation. They then define a computable function that captures the notion of virus reproduction. Using this apparatus, various types of viruses can be classified:

- *Overwriting viruses*: let v be a virus, and let $(p_1, p_2, \ldots, p_n)$ be a list of programs on a computer system. The virus v is overwriting if $\phi_v(p_1, p_2, \ldots, p_n) = (v, v, \ldots, v)$. In other words v overwrites every program p_i with its own code.

- *Ecto-symbiotic viruses*: let v be a virus and let $(p_1, p_2, \ldots, p_n)$ be a list of programs, as before. The virus v is ecto-symbiotic if

$$\phi_v(p_1, p_2, \ldots, p_n) = (\delta(v, p_1), \delta(v, p_2), \ldots, \delta(v, p_n))$$

where $\delta(x, y)$ is a concatenation of x and y. Ecto-symbiotic viruses, therefore, are viruses that do not change the behaviour of the programs they infect, but rather add their own code parasitically.

- *Organisms*: let v be a virus and p be any program. Then, the virus v is an organism if $\phi_v(p) = (v, p)$. Therefore organisms are viruses that reproduce without modifying any programs, i.e., they simply make copies of themselves.

- *Companion viruses*: Let v be a virus and p be some program. Then, the virus v is a companion virus if $\phi_v(p) = v'$ where for all $x \in D$, $\phi_{v'}(x) = \phi_p(\phi_v(x))$. Therefore, when a companion virus is executed, it reproduces before executing the program that it has infected.

4.4.1.3 Phylogenetic Classifications

One of the most well-explored computer virus classification methods is phylogenetic classification, in which viruses are related if they share some common feature, such as a sequence of code that has been used in both viruses, or a pattern of behaviours at run-time. Phylogenetics is the field of biology that deals with relations between groups of organisms, and so these classifications are based on an application of techniques from the field of phylogenetics to the domain of computer viruses.

The broad aim of several of these phylogenetic classification methods is automatic classification of malware. For example, anti-virus software companies commonly classify malware based on abstract families, where a group of viruses have an abstract name, and each virus within the family is a variant. For example, in October 2007, the website of anti-virus software vendor F-Secure listed over 90 variants of the Bagle family of worms, called Bagle.A, Bagle.B, and so on. One application of phylogenetic classification would be to take an unknown worm, and by comparison to other worms, classify it within a family of related viruses.

Goldberg et al [61] give a formal definition of computer virus phylogeny trees, and assuming that computer viruses use common code blocks which can be used to identify "families" of related computer viruses, show how phylogeny trees can be constructed algorithmically. The problem of the construction of phylogenetic directed acyclic graphs ("PhyloDAGs") is tackled, and theoretical results on the hardness of phylogenetic graph construction are obtained. The approach is not applied to any real malware, as the aim of the paper seems to be to provide a sound theoretical underpinning to the problem of malware phylogeny.

Carrera & Erdélyi [24] describe a practical approach to malware phylogeny graph generation based on an analysis of the functions used within the malware. A func-

tion call flow graph is generated for each malware example, and algorithms are used to analyse the similarities between the graphs. A close similarity is suggestive of a relationship, and so phylogeny graphs can be generated. The approach is shown to be useful in classifying some known related malware as similar, but for other malware such as the Bagle worm, for which there are variants lacking a reproductive mechanism, it was not possible to show relatedness to a high degree using the technique.

Wehner [162] uses a method based on Kolmogorov complexity to determine the complexity of various worms. Every pair of worms in the test set was assigned a value, based on a technique called the normalized compression distance (NCD), which represented the similarity between the pair. Then, a tree graph was constructed for the test set which reflected the similarities between the different worms. This classification is very similar to the phylogenetic classifications if we suppose that the reason that two worms might be similar is that they are related as ancestor and progeny.

Karim et al [78, 79] present an approach to malware phylogeny using a technique based on maximal π-patterns, a technique first used in bioinformatics to analyse the occurrence of substrings within a string. Malware programs in the test set are concatenated together into a single string, and the maximal π-patterns are extracted, which can then be used for feature or distance-based comparisons. These comparisons are then used to construct a phylogeny tree for the malware.

4.4.1.4 Classification by Spafford

Spafford classification is based on an artificial life view of computer viruses [135][3]. The author presents an "evolution" of viruses, in which successive generations increase in complexity. Of course, Spafford does not suggest that natural evolution is taking place; although there is a refinement process at work that resembles an evolutionary system. The first computer viruses were relatively simple; early personal computers did not have anti-virus software, and so computer viruses did not need to ensure their survival by code obfuscation, for example. The first anti-virus software was developed, and in order to ensure the reproductive success of their creations, the virus writers had to increase the complexity of their viruses in order to evade detection. This was the beginning of the "arms race" between virus writers and anti-virus researchers.

Spafford's classification gives a coarse-grained model of five generations of this "evolutionary" process:

- Generation 1: Simple viruses, which do nothing more complicated than reproduce. They are simple to detect by signature-based means, and their activity is easily

[3]It is interesting to compare Ludwig's work on a similar theme (see Section 1.4.1).

observed by an increase in file size, for example.

- Generation 2: Self-recognition viruses, which avoid repeated infection of the same host file. An example of this kind of virus is the Strangebrew virus presented in Section 4.2.5, which can detect whether it has already infected a file, and if so, will not re-infect the same file. The advantage that self-recognition viruses have over simple viruses is that they can evade detection for longer periods of time because their presence is less obvious, and since they will only infect files a limited number of times, they are unlikely to cause the computer system to run out of disk space and become unusable — since this would most likely result in erasure of files containing the virus, or even a re-install of the operation system, self-recognition viruses ensure their survival by being more conservative in their reproduction strategy.

- Generation 3: Stealth viruses, which are memory-resident, and are able to intercept attempts by anti-virus software to read the contents of infected executable files. Once the call is intercepted, the stealth virus returns information to the anti-virus software that will appear innocuous.

- Generation 4: Armoured viruses. In order to understand and create detection strategies for new virus threats, anti-virus researchers must attempt to reverse-engineer computer virus code. Armoured viruses attempt to delay this process through logical code obfuscation, in which unnecessary or confusing code is introduced in order to obstruct the reverse-engineer.

- Generation 5: Polymorphic viruses, which refers to computer viruses whose code is obfuscated using metamorphism and/or encryption. This obfuscation helps to prevent reverse-engineering, as is the case with Armoured viruses, but also helps to prevent detection by static analysis means such as signature scanning, heuristic analysis, file integrity checking and spectral analysis, because the viral code can have a different syntax for every virus offspring. This virus type is known to be undetectable in general. Metamorphic computer viruses were discussed in more detail in Chapter 2.

Spafford's classification of computer viruses, like Adleman's, is ontological, since it presents a way of thinking about the domain of computer viruses and other forms of reproducing malware. However, Spafford did not formalise his ideas, probably because the primary goal of his research was to describe computer viruses in terms of artificial life, rather than to develop a complete theory of computer virus classification.

4.4.1.5 Classification by Weaver et al

Weaver et al [151] have given an informal taxonomy of computer worms based on several criteria including target discovery (the means by which the worm finds new hosts to infect), carrier (the means by which the worm sends its code to new hosts), activation (the means by which the worm's code is executed on host machines), payloads (the non-reproductive code that a worm may execute once infection has taken place) and attacker motivation (the sociological and criminological reasons for the attack). This classification is not formal, and includes a non-computational criterion (attacker motivation), and therefore can be seen, perhaps, as a classification of the security threats posed by worms, rather than a classification of the worms themselves. The taxonomy is summarised as follows:

- *Target Discovery.* A worm can scan for potential hosts on the network (**network scanning**), use **pre-generated target lists** (a list of IP addresses stored as literal values, for example), **externally-generated target lists** (e.g., a list of targets stored on a server), **internal target lists** (where information stored on the host computer is used to find targets, e.g., an email address book), or **passive** target discovery (the worm does not automatically seek new targets, e.g., it waits for the user to send an email before attaching the worm code to the email).

- *Propagation Carriers and Distribution Mechanisms.* A worm can be actively transmit its code as part of the infection process (**self-carried**), open a second channel of communication for transmission of the worm code (**second channel)**, or embed itself in a normal communication channel (**embedded**).

- *Activation.* A worm can be activated by the user (**human activation**), by a trigger such as rebooting the machine (**human activity-based activation**), by a scheduled process such as a daemon program or auto-updater (**scheduled process activation**), or through self-execution in which the worm can executed the code of its offspring in order to continue the reproduction process (**self activation**).

- *Payloads.* A worm might not have a payload at all (**none**). If there is a payload, then it could create a security vulnerability so that the machine can be accessed by an attacker over the network (**internet remote control**), or it could create an open mail relay for use by spammers (**spam relays**), or it could create an HTTP proxy through which illicit or illegal websites can be routed (**HTML-proxies**), or it could form part of a denial of service (DOS) attack by helping to bombard another node on the network with traffic (**Internet DOS**), or it could perform

espionage and spy on user data or activity (**data collection**), or it could offer internet remote control or data collection payloads for sale (**Access for Sale**), or it could damage data on the host computer (**data damage**), or it could induce behaviour in human users through extortion or take control of physical objects that are controlled over the network (**physical-world remote control**), or perform malicious denial of service acts on physical systems such as mail order companies (**physical DOS**), or use devices connected to the physical world to perform reconnaissance (**physical-world reconnaissance**), or damage physical objects like erasable hardware BIOS circuity (**physical-world damage**), or modify the worm's behaviour by checking for updates from servers (**worm maintenance**).

- *Attacker Motivation.* The attacker may be motivated by **experimental curiosity**, **pride and power**, **commercial advantage** (e.g., a worm may be designed to perform a denial of service attack on a competitor), **extortion and criminal gain** (e.g., holding a user's data to ransom), malevolence without rational motivation (**random protest**), political activism (**political protest**), **terrorism**, or **cyber warfare** (e.g., disrupting essential systems such as those used by emergency services).

4.4.1.6 Industrial Classifications

Most anti-virus software companies have their own schemes for malware naming, which involve some implicit classification, e.g., names like "W32.Wargbot" or "W97M/Trojan-Dropper.Lafool.NAA" give some information about the platform (e.g., 32-bit Microsoft Windows) and/or the primary reproductive mode of the virus (e.g., "trojan dropper"). Many naming schemes are ad hoc, and for good reason: anti-virus software engineers have to reverse engineer viruses as quickly as possible, in order to generate a means of detection and removal, and because of this, the naming schemes for computer viruses are generally not standardised or consistent [64]. As a result it is very common for a computer virus to be known under several different names.

An example of an industrial virus classification is that made by the Symantec Corporation [137] — vendors of a very well-established and popular anti-virus product. Their malware naming syntax is given in Extended BNF notation as follows:

```
<malware-name> ::= <prefix>.<name>[.<variant>][<suffix>]
```

Examination of the malware naming scheme given by Symantec reveals that the `<prefix>` can indicate any of the following:

- The platform that the viruses requires to reproduce, e.g., "W32" (32-bit Windows) or "Linux".
- The scripting language used by the virus, e.g., "BAT" (for viruses that infect MS-DOS batch files).
- That the malware is a Trojan horse program, e.g., "Trojan".
- The payload of the malware, e.g., "PWSTEAL" for Trojan horses that steal passwords.
- The real-world effects of the malware behaviour, e.g., "DoS" for malware that conduct Internet-based denials of service.

The `<name>` can be anything, and is presumably determined by the anti-virus researcher(s) responsible for identifying the virus. For example, the name may be inspired by the payload of the malware, e.g., "LoveLetter" for an email worm that spreads by making the user believe they have been sent an amorous message. The `<variant>` is usually some modifier used to denote a variant of the virus whose `<name>` precedes it, e.g., "A", "B", and so on. The `<suffix>` denotes miscellaneous information about the virus, including but not limited to the following:

- The reproductive behaviour of the malware, e.g., "@m" for email worms.
- The malware facilitates the reproduction of other malware, e.g., ".dr" for programs that "drop" other malware onto the computer system.
- The malware is a worm, e.g., ".Worm".

For example, one piece of malware is denoted as `W32.Beagle.AV@mm`, indicating that it reproduces using the 32-bit Windows platform, is a variant of the "Beagle" malware family, and sends emails out en masse ("@mm") in order to reproduce.

Recently there have been efforts to standardise the many and varied malware naming schemes, e.g., the Common Malware Enumeration (CME) project [87] and the Computer Antivirus Research Organization (CARO) virus naming convention[4]. CME is still at an early stage, and the current status of CARO is unclear. However, it is clear that we are far from uniformity with respect to malware naming schemes, as is revealed in recent surveys [65, 54].

[4]See `http://www.caro.org/`.

4.4.2 Comparisons with Related Work

This chapter is concerned with an application of the formal affordance-based reproduction models, first presented in Chapter 3, to the specific problem of computer virus classification. Therefore, the purpose of the following comparisons is not to critically assess affordance-based reproduction models, which has already been done in the previous chapter, but rather to critically-assess the classification scheme presented here.

The variety of computer virus classification schemes is fascinating. Adleman classifies computer viruses according to abstract definitions of their behaviour. Bonfante et al use a functional description of behaviour to classify computer viruses according to their means of reproduction. Phylogenetic classification arises from the need to group computer viruses automatically according to their apparent heredity. Spafford presents a taxonomy of the "evolution" of computer viruses over their short history. Weaver et al classify worms according to their behaviour, social effects and author motivation. Industrial classifications are the result of computer virology in the trenches; assumptions and decisions about heredity, reproduction method and payload are reflected in an ad hoc naming system.

Most classifications arise from some insight into the universe of objects being classified, and therefore the only requirement upon a classification being considered worthy of the title is that it should have some explanatory power. Therefore, the task of comparing different computer virus classifications becomes a relative one, as all of the classifications presented above have some basis in rational thought and have some explanatory power. Consequently it is, perhaps, difficult to justify the use of one computer virus classification over another. Therefore, we will instead draw qualitative comparisons between our approach and the other approaches in the literature, with the intention of illustrating the novelty and insight of our approach.

4.4.2.1 Comparison with Formal Classifications

The approaches taken by Adleman and Bonfante et al are similar in that they are both based on a specific model of computation — recursive functions. Computer viruses are defined within the model, and then variants on that class are defined in order to determine sub-classes. For example, Adleman uses first-order logic to define abstract features of viruses based on recursive functions, such as contagiousness and pathogenicity, that are then used as the basis of a classification; Bonfante et al's use recursive functions to determine specific virus behaviours, such as companion viruses. Another similar approach is given in our earlier work [153, 154], in which Abstract State Machines (themselves a model of computation) are used to define viruses, and then sub-classes

are defined which satisfy the requirements of being a virus, together with some interesting features (in the form of additional restrictions) which separate them from other viruses.

These methods contrast sharply with the approach presented in this chapter, which is based on an algebraic notion of reproduction, in the form of formal reproduction models. An algebra is simply a set, together with operations on that set, and is analogous to other concepts in computer science, such as "abstract data types" and "objects" in object-oriented programming. Therefore, formal reproduction models can be seen as a many-sorted set S (consisting of states, actions, entities and so on), and a set of operations on that set (such as $\mathit{Aff} : S \times S \to S$ and $\varepsilon : S \times S \to \{0, 1\}$). We do not base our classification on a single model of computation, but rather specify the circumstances in which an act of reproduction takes place, e.g., a labelled transition system which describes how the system evolves over time, or the set of entities that are present during the reproductive act.

Since it is a fundamental assumption within our models that reproduction has taken place, our models do not describe the computation involved in a reproductive process. However, this is the aim of the computability-based classifications, which lay out the formal computational nature of computer viruses, e.g., the mechanics of their reproductive behaviour. In contrast, the aim of our approach is to describe formally the interactions between the computer virus and its environment based on a formal description of affordance theory, and therefore come to a more "ecological" view of computer virus reproduction in terms of the reproductive reliance of the computer virus on external agency.

The phylogenetic approach to classification, particularly as it is described by Goldberg et al, makes significant use of formal tools such as algorithmic complexity theory to describe a possible means of automatic computer virus classification. This approach is fundamentally different to our approach, as it is based on an observed low-level feature of computer viruses: that different viruses tend to share similar code blocks, and these similarities can be used to create algorithmically-generated phylogenetic trees based on the varying degrees of similarity. Our approach, in contrast, is based upon a high-level feature of computer viruses: that they are part of an environment, from which they may receive assistance during the act of reproduction. It is not necessarily the case that computer viruses must share common code blocks, in fact, it is the case in certain examples such as Gödel numbering of programs, that there are no shared code blocks at all. However, it is arguable that our approach to computer virus classification is applicable to computer viruses programmed in any language, since the language is not strictly relevant within our reproduction models, as the reproductive process is

modelled using a labelled transition system.

4.4.2.2 Comparison with Informal Classifications

Our approach to computer virus classification is formal, and therefore is very different in implementation to the informal approaches of Spafford, Weaver et al and industry. However, it may be interesting to compare our approach with these others in terms of the intention of the classification.

The aim of Spafford's classification is to question whether computer viruses are a form of artificial life; the five generations are of increasing complexity, and are suggestive of an evolutionary process. Weaver et al describe a multi-dimensional classification of computer worms, incorporating reproductive, payload and social features of various worms. The authors do not apply their approach to any real-life examples of worms, and state that "in order to understand the worm threat, it is necessary to understand the various types of worms, payloads, and attackers." Therefore, it appears that their classification is meant as an illustration of the variety of worm behaviours, effects etc. The classification of computer viruses by anti-virus software vendors is essentially an ad hoc naming system, and is meant primarily to establish a common, descriptive name for a recurring threat.

Therefore, the primary intentions of two of the informal classifications are illustrative and rhetorical — Spafford is showing the life-like characteristics of computer viruses, and Weaver et al are showing the wide variety of present-day computer worms. The primary intention of the third classification is to decide upon names for certain reproducing programs, so that effective detection methods can be found as quickly as possible.

The primary intentions of our approach are (i) a practical application of formal affordance-based reproduction models, i.e., a demonstration of their practical relevance; (ii) a formal modelling system for computer viruses, so that viruses may be compared and classified in a formal way; and (iii) an investigation into how different classifications of the same computer virus can be formed, and how this relates to observable behaviour of a computer virus. Therefore, our approach differs from the informal approaches not just in its formality, but also in its motivation and intended application.

4.4.2.3 Classification of Models versus Classification of Computer Viruses

An important distinction between our classification presented in this chapter, and the related work is that we do not classify computer viruses, but rather models of computer viruses. In every other approach, a particular computer virus is classified, based on observation of its characteristics. For instance, Adleman's classification hinges on the

observation of a computer virus as contagious (or not) and pathogenic (or not). Bonfante et al classify computer viruses using a list of abstract computational behaviours, expressed using recursive functions, that we can identify real-life viruses as having (or not). The phylogenetic classifications are essentially algorithmic methods for constructing graphs which display the likely lineage of a computer virus — it's classification is therefore its relationship to other nodes on the graph. Spafford classifies computer viruses by different levels of observed sophistication. Weaver et al classify computer viruses based on observed characteristics. The industrial classification is essentially an ad hoc naming scheme, with different classes (e.g., "Worm") denoting observed behaviours.

Therefore, it is possible to classify computer viruses directly, or first make a model of their behaviour which we then classify (as in our approach). The first approach is a one-step process from computer virus to classification

$$\text{virus} \longrightarrow \text{classification} \tag{4.1}$$

whereas the second approach takes two steps — modelling, then classification:

$$\text{virus} \longrightarrow \text{model} \longrightarrow \text{classification} \tag{4.2}$$

Actually, these two approaches are different descriptions of the same classification process. In the case where we appear to classify computer viruses directly, classification is only possible thanks to an implicit model of the virus's behaviour in the mind of the classifier. For example, to classify a computer virus using Adleman's classification, we must prove that its Turing machine satisfies certain properties. Therefore, we need to know to which Turing machine(s) a computer virus maps. We might formulate this as a function $m : P \rightarrow TM$, where P is the set of programs in some language and TM is the set of Turing machines. The function m essentially specifies the syntax and semantics of a programming language. However, there is no canonical form of m. A computer virus exists within a state of a real-life computing machine. Even if we have a formal definition of the syntax and semantics of a programming language, we cannot be sure that the computing machine executing the virus actually satisfies the formal specification[5]. Therefore, there is no canonical function m, and whenever we try to

[5]This statement might appear to be controversial. It is common in computer science to prove that a given computer can be proven to satisfy a formal specification. However, this statement is actually more philosophical: how can we ever be sure that we have an exact formal model of any real-life system? The best that we can ask of any formal models is that it fits the observed data; however there may be extremes of circumstance that permit anomalous behaviour, e.g., one can imagine a scenario in which a non-reproducing program could be made to reproduce if the machine it were running on were struck by lightning!

formulate it, we are simply creating a model of what we think the relationship between P and TM ought to be. Therefore, the one-step classification shown in (4.1) is the same as the two-step classification in (4.2) — the only difference is that the modelling step is implicit.

Therefore, the key difference between our computer virus classification and the related work is that our modelling step is made explicit. First, an affordance-based computer virus reproduction model is generated, and then it is classified. The classification process is purely formal, as classification is based on the properties of the model, rather than the properties of the computer virus. Therefore, the potential inaccuracy of classification is limited to the modelling stage only.

It is possible to take one-step classification systems and turn them into two-step systems by making the model explicit. For example, in the case of the classification of Bonfante et al, we could we could establish some procedure Q for converting computer viruses into recursive functions. Then, classification of the recursive functions could take place formally and according to the authors' definitions. The procedure Q would therefore corresponding to the modelling stage, and the classification would therefore become a two-step classification system.

Another example of a two-step classification system would be to construct a formal specification of the operational semantics of a programming language, similar to the Maude specification of Intel 64 used in Chapter 2. This would effectively give an explicit mapping from programs to models. For example, the transition rule could be derived from the state transitions of a Maude reduction corresponding to the execution of the program. It would be possible to further develop the specification so that automatic classification would be possible through a representation of a behaviour monitor along the lines of Section 4.3.4.

4.4.2.4 General Comments on Affordance-based Classification

Intuitively, computer viruses that are classified as unassisted within our classification are those that are reproductively isolated, i.e., those that do not require the help of external entities during their reproductive process. Consequently, those classified as assisted require the help of external entities for their reproduction. Here our approach is similar to the work of Taylor [142], who makes the distinction between unassisted and assisted reproduction with respect to artificial life. This similarity is discussed in more detail in Section 3.7.1.

Our classification of computer viruses is a special case of the construction and classification of reproduction models from the previous chapter, which places computer viruses within the broader class of natural and artificial life forms. This relationship

between computer viruses and other forms of life has been explored by Ludwig [97] and Spafford [135], in their descriptions of computer viruses as artificial life, and by Cohen's treatise [33] on living computer programs. The comparison between computer viruses and other reproductive systems has resulted in interesting techniques for anti-virus software such as computer immune systems [83, 134, 70], and in that sense we hope that the formal relationship between computer viruses and other life forms has been further demonstrated in this chapter, and could assist in the application of concepts from the study of natural and artificial life to the problem of malware control. In addition, we believe our description of computer viruses within a formal theoretical framework also capable of describing natural and artificial life systems further supports the ideas of Ludwig, Spafford and Cohen: that computer viruses are not merely a security problem or a computational curiosity, but a life form in their own right.

Chapter 5: Conclusion

In Chapter 1 we explained that the structure of this thesis is such that the chapters are (almost) self-contained, each with their own introduction, literature review and summary. Therefore, we conclude this thesis by summarising the novel contributions of each chapter, and by giving a number of directions for future research.

5.1 Novel Contributions

The novel contributions of this thesis are listed in order of presentation:

Chapter 2

- A formal algebraic specification of a subset of the Intel 64 assembly programming language has been developed. Intel 64 is the processor language of the majority of personal computers worldwide, and since our specification is general-purpose, it can be applied to proving properties of programs written in Intel 64.

- The formal algebraic specification of Intel 64 has been shown to be applicable to the practical problem of metamorphic computer virus detection. The specification is made executable by using the Maude term rewriting engine, and can be used for dynamic analysis of metamorphic computer viruses. We showed how it is possible to prove the equivalence of different metamorphic code fragments using this approach, and described how this could be applied to metamorphic computer virus detection.

- Formal definitions of the equivalence and semi-equivalence of programs were developed, and were used to prove the Equivalence in Context theorem. This theorem states that a semi-equivalent store can be made equivalent through the execution of an instruction sequence that makes all non-equivalent variables equivalent.

- Equivalence in Context has been shown to be applicable to the practical problem of detection of metamorphic computer viruses, which can generate syntactic

variants that are semi-equivalent. We have shown how it is possible to apply Equivalence in Context using static analysis in order to prove the contextual equivalence of semi-equivalent metamorphic computer virus variants.

Chapter 3

- Gibson's theory of affordances has been applied to the problem of formal reproduction modelling and classification. A reproduction system can be described in terms of a labelled transition system, with entities (of which the reproducer is one) in various states. The entities required for a particular action are the entities that afford that action. The affordance metaphor is useful as it captures notions of agency and collaboration in the reproductive act, enabling classification as trivial or non-trivial, assisted or unassisted. We have shown how assisted classification can be sub-classified further using aspects, which isolate certain actions (e.g., the payload of a computer virus) and test whether they are assisted or not.

- We have proven that refinements exist only between certain classes of reproduction models, i.e., the space of formal affordance-based reproduction models is structured under refinement. In particular, we proved that there are no refinements from non-trivial to trivial models, and there are no refinements to models that are trivial and unassisted.

- We have proven the Unassisted and Assisted Reproduction Theorems. The Unassisted Reproduction Theorem says that every affordance-based reproduction model can be refined by an unassisted reproduction model. The Assisted Reproduction Theorem says that every non-trivial reproduction model refines an assisted reproduction model. Together, the two theorems show that for any non-trivial reproduction model — classified as assisted or unassisted — there is always another model, related under refinement, that has the opposite classification.

- We have shown that formal affordance-based reproduction models can be used in the domains of biology, computer virology and artificial life. Such models can then be classified and refined, e.g., to demonstrate the relationships between models that have the same reproductive process, but have different classifications. We have constructed affordance-based reproduction models for biological viruses, computer viruses and artificial life forms, and have sketched how models might be constructed for more complex reproduction systems, such as sexual reproduction.

- We have discussed the philosophical implications of our formal affordance-based reproduction models, including the idea that classification based on assistance

is arbitrary, the idea of reproduction as preservation of information over time and the relationship between our work and Rosen's work on the irreducibility of biological systems and a paradox implicit in the notion of unassisted reproduction.

Chapter 4

- We have applied our formal affordance-based reproduction models to the practical problem of computer virus modelling and classification. We have shown that the affordance metaphor is apt within computer virology, as computer viruses frequently use resources of the host computer in order to reproduce, e.g., the operating system or a network.

- We have shown how the classification of affordance-based models of computer viruses as unassisted or assisted mirrors the classification of computer viruses as undetectable or detectable by behaviour monitors. Different behaviour monitors employ different means of observation of computer virus behaviour. In cases where a monitor can intercept a communication between a computer virus and a resource, we can model this resource as a separate entity which affords some action in the computer virus's affordance-based reproduction model. In order to demonstrate this technique, we have constructed models of computer viruses programmed in a Unix shell script language, Visual Basic script, Java and $x86$ assembly language, and showed how they can be classified as unassisted or assisted.

- We have shown how the classification of affordance-based computer virus reproduction models can be achieved by automatic means, either by static or dynamic analysis. We gave case studies of how this might be achieved for real-life Visual Basic script viruses, and discussed the possibility of metrics for further subclassification of assisted models. In addition, we showed that these techniques can be implemented using well-known string pattern matching and behaviour monitoring techniques.

5.2 Directions for Future Research

The work in this thesis has highlighted a number of possible avenues for future research. We describe these as follows.

5.2.1 Complexity of Detecting Metamorphic Computer Viruses

In Chapter 2 we gave two different ways to detect metamorphic computer viruses using equivalence checking: equivalence in context and dynamic analysis using the Maude specification of Intel 64. However, the emphasis of this chapter was on establishing the methods for detection and providing illustrative examples. For both detection methods it would be desirable to extend this investigation to include algorithms for implementing these procedures. An obvious benefit would be an analysis of the computational time and space complexity of the algorithms.

In the case of equivalence in context, which is based on formal definitions and results, the development of an algorithm to prove equivalence in context — one which takes two semi-equivalent stores $s_1 \equiv_W s_2$ and an instruction sequence i (of length n) that equalises them — would seem to be straightforward and natural. One has to define an iterative (or recursive) process in which the first statement i_0 in i is tested to see whether $V_{in}(i_0) \subseteq W$. If this is the case, then $V_{out}(i_0)$ is added to W, and the process is repeated for i_1, and so on. The iterative procedure would have to execute n times in the worst case. On each iteration, a subset-checking procedure and a set union procedure would need to be implemented. We could implement these operations using hash tables for the sets, which have an average-case time complexity of $O(1)$ for insertion, search and deletion operations [34]. In order to check that $A \subseteq B$, a hash table search operation would have to execute $|A|$ times, and therefore a subset-checking procedure for a subset A would have average-case time complexity of $O(|A|)$. To perform $A \cup B$ using a hash table insertion operation, each element in the smaller hash table would have to be added to the other hash table, meaning an average-case time complexity of $O(|A|)$ (assuming A to be the smaller set). Therefore we would have an algorithm that iterates n times, and on each iteration performs two other operations that have a linear average-time complexity. Therefore, we might expect the overall time complexity of the equivalence in context algorithm to be no more than $O(n^2)$, although a formal proof would be needed. The algorithmic space complexity would be expected to be low, as the only space requirement for the iterative algorithm is the set of equivalent variables W, which needs to be no larger than the set of variables used by s_1, s_2 and i.

In the case of the equivalence checking using the Maude specification of Intel 64, a formal algorithmic treatment (as described for the equivalence in context algorithm above) would be less desirable for a number of reasons. Firstly, the algorithmic complexity would be dependent on the implementation of the Maude term rewriting engine. Although the implementation details are available, they may change in subsequent versions of Maude which would render inaccurate any formal analysis. Secondly, the al-

gorithmic complexity would also depend upon the term rewriting system implemented within the Maude specification of Intel 64 (represented by the equations and conditional equations). Therefore, any changes to the specification would render any formal results inaccurate. Perhaps the best investigation into the time and space requirements of this form of equivalence checking would be empirical, i.e., a large set of typical inputs of varying size would be prepared and presented to the Maude specification and term rewriting engine for processing. The time/space taken could be measured for each input, and a scatter graph of program size versus time/space taken could be drawn. The result would suggest the magnitude of the time/space complexity (e.g., constant, linear, polynomial, exponential, etc.). For the simple experiments using the Maude specification in Chapter 2, the execution time was on the order of milliseconds; however, an empirical analysis would provide a more convincing account of the computational demands of this approach.

5.2.2 Further Detection of Metamorphic Computer Viruses

So far a subset of the Intel 64 instruction set has been specified using Maude, but there is no reason why the entire instruction set could not be implemented, as several imperative programming languages have been specified using similar approaches [104, 60, 58]. A full specification of Intel 64 would enable application of the detection techniques described in Chapter 2 to computer viruses that use instructions beyond the subset of Intel 64 specified here (see Appendix A for the Maude specification). The technique of (semi-)equivalence proof has been applied to two of the nine computer virus code metamorphosis types given in Section 2.2.1: equivalent sequence replacement and arithmetical/Boolean metamorphism. A practical extension of this work would be to extend and test the techniques shown here for other types of metamorphism. In Section 2.5 a method for proving equivalence in context was given. An extension of this would be to find further means of proving equivalence in context, which would aid the detection of metamorphic computer viruses that employ semi-equivalence based code metamorphosis.

An obvious further application of the methods for computer virus detection described in Chapter 2 is to combine them with other means of metamorphic computer virus detection. For instance, the formally-verified equivalent code library described in Section 2.8.2.1 may not always result in reduction of every generation of a metamorphic computer virus to a normal form. However, the overall syntactic variance of the set of all generations may be significantly reduced, so that another technique may be used to enable detection. For instance, the neural network-based approach of Yoo et al (described in Section 2.9.1.5) relies on the identification of similar code structures,

and therefore may be assisted by an equivalent code library.

An advantage of the Maude specification of Intel 64 described in Chapter 2 is its flexibility. We performed simple reductions to prove equivalence of metamorphic code, one of the most straightforward proofs possible for a specification of a programming language in Maude. However, there are many more techniques that can be used, such as inductive proofs of loop behaviour or proofs of program properties [58]. In addition, there are several automated theorem proving tools that can be used with Maude, including a model checker and inductive theorem prover [105]. These tools can be used in concert with the Intel 64 specification to prove properties of programs written in the language; an obvious useful application would be to prove that an anti-virus program can detect a given set of viruses. This would provide an added level of assurance to the users of anti-virus software: not only do they have anti-virus software installed on their systems, but it is proven formally to work.

An additional use of Maude's built-in model checker is described in the next section on detection of virtualization-based malware.

5.2.3 Detection of Virtualization by Metamorphic Code Generation

Virtual machine-based rootkits can be used to force the user to use an operating system that executes within a virtual machine [126, 84, 127, 52]. The advantages to the potential attacker are obvious; the user would be oblivious to any malicious programs executing outside the virtual machine. Rutkowska describes an approach to detection of virtualized malware from within the virtualized operating system, based on the execution of an Intel 64 assembly language instruction called `sidt` x [126]. When executed, this instruction stores the contents of the interrupt descriptor table register into the destination operand x. The value of x varies depending on whether the `sidt` instruction has been executed inside or outside a virtual machine, and therefore detection is possible. This method is called *Red Pill*.

However, this detection method is not always guaranteed to work, as the user's interaction with the operating system can be controlled and manipulated in order to avoid detection using methods like Red Pill. King et al describe a counter-measure to Red Pill based on emulation [84]. The virtual machine monitor (VMM), which controls execution of the virtual machine, detects when the Red Pill executable is being loaded into memory, and sets a breakpoint to trap the execution of `sidt`. When the breakpoint is reached, the VMM will emulate the instruction, setting the value of the destination operand of `sidt` to a value not indicating detection. The authors note that this detection counter-measure could be defeated by a program R that generates the `sidt` instruction dynamically.

At this point the writers of the malware have two options: they can re-write the virtualization-based malware so that it can detect R, as well as Red Pill, by static analysis. Alternatively, they can trace the execution of programs in order to detect by dynamic analysis any occurrence of Red Pill. King et al note that the latter could be computationally expensive, adding overhead which might result in detection by timing methods (see, e.g., [52]).

Suppose that the former option were chosen. Then, all the malware writers need do in order to avoid detection of their malware is to adjust their program to detect R' as well as R and Red Pill. Therefore, from the perspective of the writers of the Red Pill program, a means of automatic generation of programs that have the same behaviour as Red Pill would be desirable. In other words, we would like to use a metamorphic version of Red Pill, that changes its syntax at run-time in order to evade detection. Clearly, metamorphic engines as seen in metamorphic computer viruses could be used, but they are not reliable, in that the syntactic variants generated are not guaranteed to preserve the semantics of the original program. Therefore, we propose a solution to this problem based on our formal description of Intel 64 assembly language, which could be employed as a means of generating Red Pill variants before or during run-time.

As was discussed in Section 2.4, the Maude specification of Intel 64 denotes a term rewriting system. The usual application of such a system is to apply equations and rewrite rules in order to reduce terms to some terminal form, i.e., to rewrite terms until they can no longer be rewritten. However, it is also possible to perform a search of the rewriting space of a term rewriting system in order to determine whether it is possible to reduce one term to another, and if there are non-deterministic aspects to the term rewriting system, whether there are multiple ways of performing such a reduction. It is also possible to test for some conditional value, and find all rewriting routes that lead to a term satisfying that condition. In this way, Maude can be used for model checking.

Using the Maude specification of Intel 64, it is possible to rewrite a term such as `S[[eax]]`, which denotes the value of `eax` in some store `S`, using a variety of rewrite rules, and check using a breadth-first search of the term rewriting system whether a condition such as `S[[eax]] = "sidt"` ever becomes true, which says that the value of `eax` in some store `S` is equal to `"sidt"`. In other words, it is possible to create a term rewriting system in Maude that constructs programs based on rewrite rules, and search the rewriting space for constructed programs that are satisfy the requirement that `"sidt"` is stored in some variable. Figure 5.1 shows such a term rewriting system that generates four ways of constructing a program that satisfies the condition that `S[[eax]] = "sidt"`. Therefore, it is possible to create a metamorphic code engine based on our formal specification of Intel 64 in Maude.

```
rl [1] : S[[eax]]  =>  S ; mov ebx, "sidt" [[eax]] .
rl [2] : S[[eax]]  =>  S ; mov eax, ebx [[eax]] .
rl [3] : S[[eax]]  =>  S ; mov ecx, ebx [[eax]] .
rl [4] : S[[eax]]  =>  S ; mov eax, ecx [[eax]] .
```

Let the end condition be `s[[eax]] = "sidt"`.
Then, apply any of the following to reach the end condition from `s[[eax]]`:

$$(1,2),(1,2,3),(1,2,3,4),(1,3,4),(1,3,3,4),(1,3,\ldots,3,4).$$

Figure 5.1: A metamorphic engine based on the Maude specification of Intel 64. The four lines beginning with `rl` are rewrite rules that construct programs by appending an instruction to an instruction sequence. The search of the rewriting space then reveals sequences of rewrite rule applications that result in programs that assign `"sidt"` to `eax`.

The previous example shows how we can automatically generate programs that assign the number corresponding to the opcode of `sidt` x to some variable, e.g., register `eax`. Therefore this technique could be used to generate automatically syntactically-mutated forms of a Red Pill program in order to evade detection of the Red Pill program by the VMM. This approach is preferable to other, less formal, means of metamorphism (such as using a metamorphic engine from a computer virus) because we can use the Maude specification of Intel 64 to guarantee that any metamorphic code generated satisfies the requirements of Red Pill.

5.2.4 Modelling Reproduction at Different Abstraction Levels

In Example 8 from Section 3.5.1 we showed how a reproduction model of a copier computer virus which was classified as assisted could be refined to another model in which the classification was unassisted. This was a demonstration of the Unassisted Reproduction Theorem, which says that such a refinement exists for any affordance-based reproduction model. As well as being an interesting feature of affordance-based models, this may have an interesting philosophical implication. For example, in the case of the copier computer virus, we achieved the shift from assisted to unassisted classification by specifying a function h which maps all entities in the assisted model M_{cv} to the reproducer, thus making the classification of $M_{cv}^{\#}$ unassisted. In other words, we have an entity in $M_{cv}^{\#}$ which is, notionally, the conglomeration of all of the entities that afforded actions to the reproducer in M_{cv}. This raises the question of whether we should consider the computer virus to be the reproducer, or the conglomeration of the copier computer virus with the assisting entities in its environment.

In their work on the philosophy of biology, O'Malley & Dupré [109] describe similar

ontological problems when defining a biological individual. They state that it is typical within microbiology to identify a single cell of a prokaryote as an individual; however, many communities of microbes display properties commonly associated with organisms, including coordination of metabolic process, environmental modification and autolysis ("cell suicide") for the benefit of the community. There is growing evidence in the literature of such a holistic notion of biological individuality. For instance, in their paper on the role of microorganisms in corals, Rosenberg et al describe the relationship between organisms and symbiotic microorganisms ("symbionts"), and suggest a new form of evolution in terms of "hologenetics":

> "...the genome of the host can act in consortium with the genomes of the associated symbiotic microorganisms to create a hologenome. This hologenome — given the diversity and fast growth rates of microorganisms — can change more rapidly than the host genome alone, thereby conferring greater adaptive potential to the combined holobiont organism.
>
> [This] leads us to propose a hologenome theory of evolution: the holobiont with its hologenome should be considered as the unit of natural selection in evolution, and microbial symbionts have an important role in adaptation and evolution of higher organisms. Therefore, microorganisms are essential not only in the health and disease of individual higher organisms, but they also are a significant factor in species survival and evolution.
>
> This hologenome theory of evolution is derived primarily from an understanding of the biology of corals. However, a large body of data exists in the literature relating to many eukaryotic organisms and their interaction with symbiotic microorganisms — a literature that could be re-evaluated in light of this theory." [125]

In other words, hologenetics takes the view that the evolving biological individual is not the organism, but the organism in cohort with its symbionts. Rosenberg et al describe a number of different ways in which the symbionts assist the organism, and vice versa. In terms of affordance-based reproduction models, we might model the organism and the symbiont with two reproduction models, with the organism as the reproducer and the symbiont as an assisting entity in one, and the symbiont as the reproducer and the organism as an assisting entity in the other. We are then at a similar point to Section 3.7.2.5, in which we described the male and female as reproducers in two different models, one in which the male reproduces and the female assists, and the other in which the opposite happens.

Clearly, this interaction between two different symbiotic reproducers can be modelled using affordance-based models. It is possible that the refinement from an assisted

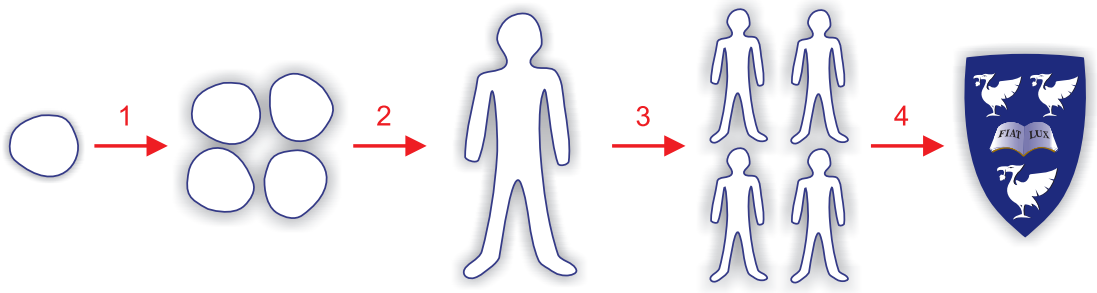


Figure 5.2: Ontological shifts from a cell, to a community of cells, to an organism, to a community of organisms, to a society (in this case, the University of Liverpool). Shifts 2 and 4 can be modelled at present using refinements; shifts 1 and 3 are left for future work.

model to an unassisted model as described above, which depends upon the conglomeration of entities involved in the reproductive process, is similar to the ontological shift from the naïve notion of individual to the community individual. In other words, we could apply the Unassisted Reproduction Theorem to the both the male and female reproduction models, resulting in a refinement to the same reproduction model in which the conglomerate entity is the male and female together. We could do the same thing with the organism and the symbiont reproduction models.

Similar ontological shifts can be seen in the field of economics, where it has been posited that firms (essentially a community of human beings with a common purpose) are reproducers [99]. Therefore, we might consider the refinement process, during which collaborative reproductive entities are conglomerated, to be a model for ontological shifts of this kind. In Figure 5.2 we give a schematic illustration of a series of ontological shifts, in which several entities are conglomerated to a single entity through a refinement between reproduction models. However, there is another relationship between models that would be required to fully describe the progression between the models in Figure 5.2. The model in which a single individual, e.g., a cell, is present is obviously related to the model where a number of individuals of the same species, e.g., a number of cells, are present. In Figure 5.2 these ontological shifts are represented in arrows 1 and 3. At present we can describe arrows 2 and 4 using refinements between models, but there is as yet no way to model the relationship described by arrows 1 and 3. This is obviously a subject for future research[1].

This formal description of ontological shifts may be of interest to those interested

[1]Indeed, our latest research in this area concerns formal component-based models of reproduction [157], in which we can describe how different components of a whole can be related. This could possibly be applied to model the relationship described by arrows 1 and 3, in which an entity becomes a community of similar entities.

in relating ontologies (e.g., [8, 9]), as the formal affordance-based reproduction models in this chapter provide a formal framework for describing ontological shifts in the perception of reproduction and life-like systems.

5.2.5 Metrics for Reliance on External Agency

In Section 4.3.5 we showed how affordance-based computer virus reproduction models can be compared using metrics of reliance on external agency. However there may be further opportunities to create metrics to compare assisted reproduction models in general, based on other factors. For example, if we see the act of reproduction as a computational process of a certain minimal complexity, then if the actions that a reproducer affords itself together are less than the complexity of the whole reproductive process, then there must necessarily be some other (external) entity that compensates for this. Therefore, when comparing two assisted reproduction models that require a similar environment (e.g., two computer viruses), we can compare their reproductive reliance on external agency by comparing the complexity of those reproductive actions that are afforded to the computer viruses, or those that the computer viruses afford themselves. For example, we could assume that the more complex a particular virus's self-afforded actions, the less the reliance on external agency. Of course, this presupposes the existence of some level of abstraction at which we can compare the complexity of actions, but in several cases, such as computer viruses, Tierran organisms, cellular automaton reproducers, etc., such a comparison would seem possible. Different methods of complexity could be used, e.g., space/time complexity, or the Kolmogorov complexity of the reproducer itself.

There is also empirical evidence of differing degrees of reliance on external agency with respect to biological viruses. It is known that, "[viruses] with large genomes depend less on host functions than those with small genomes" [66]. This effectively states that the information content in the self-description (genome) is related to the reliance on external agency (the host cell). Another possible extension of this work would be to use the methods described above to formalise this statement within our ontology.

5.2.6 Strategies for Reproduction

In December 2000, a relatively unprolific virus on the Windows 32-bit platform was able to infect executable files containing prolific network worms [138]. The destructive payload of the virus combined with the infectiousness of the worms created dangerous hybrids that were not predicted by the vendors of anti-malware software. These hybrids

were an emergent property of a complex "ecology" of reproducers, in which reproduction processes could overlap.

A useful extension of this work would be to be able to analyse these ecologies of reproducers, i.e., systems where more than one reproducer is present. Such ecologies could be constructed using affordances common between entities, for example, a bacterium might afford a site of infection for a bacteriophage virus, without necessarily specifying which virus might infect the bacterium. The labelled transition systems of the different reproductive processes could be combined using techniques such as those developed in process algebra [6]. In real-life ecologies, reproducers are capable of interesting behaviours such as crossing a species gap (e.g, biological viruses), or spontaneous virus–worm hybridisation (see above). In being able to build models of ecologies of reproducers by combining their models in a formal way, we could begin to analyse and predict interesting emergent properties of multi-reproducer systems.

5.2.7 Advanced Metrics for Assisted Computer Virus Classification

In Section 4.3.5 we showed how using a simple metric we could compare the reliance on external entities of two viruses written in Visual Basic Script. It should also be possible to develop more advanced metrics for comparing viruses with assisted classification. For example, a certain sequence of actions which require external entities may flag a given viral behaviour with a certain level of certainty. Therefore it would seem logical to incorporate this into a *weighted* metric that reflects the particular characteristics of these viruses. Different metrics could be employed for different languages, if different methods of behaviour monitoring are used for Visual Basic Script and Win32 executables, for example.

Following on from the discussion above, another possible application of our approach is towards the assessment of anti-virus behaviour monitoring software via affordance-based models. There are some similarities between our approach and the recent work by Filiol et al [46] on the evaluation of behavioural detection strategies, particularly in the use of abstract actions in reasoning about viral behaviour. Also, the use of behavioural detection hypotheses bears a resemblance to our proposed anti-virus ontologies. In future we would like to explore this relationship further, perhaps by generating a set of benchmarks based on our ontology and classification, similar to those given by Filiol et al.

Recent work by Bonfante et al [14] discusses classification of computer viruses using recursion theorems, in which a notion of externality is given through formal definitions of different types of viral behaviour, e.g., companion viruses and ecto-symbiotes that require the help of a external entities, such as the files they infect. An obvious extension

of this work would be to work towards a description of affordance-based classification of computer viruses using recursion theorems, and conversely, a description of recursion-based classification in terms of formal affordance theory.

5.2.8 Evaluation of Anti-virus Techniques

In Chapter 4 we described how the capabilities of different anti-virus behaviour monitors correspond to different classifications of the same computer virus as unassisted or assisted. For example, suppose a computer virus uses the operating system to write to files, and one behaviour monitor is able to discern communication between the computer virus and the operating system. In this case it is natural to classify the computer virus as assisted, as the behaviour monitor is able to view the assisted act of reproduction. However, if the behaviour monitor was unable to detect any communication between the computer virus and the operating system, then a classification of the computer virus as unassisted would be more natural, as the act of reproduction is apparently unassisted. Therefore, the most capable behaviour monitor would be that which was able to view every act of every computer virus's reproduction, from handling data in memory to writing to the disk to sending information over the Internet. This omniscient monitor would result in every computer virus being classified as assisted. The least capable behaviour monitor, in constrast, would be unable to discern any of the actions of the computer virus's reproduction, and would result in a classification of every computer virus as unassisted. In between these two extremes there is a spectrum of behaviour monitor capabilities resulting in different structures of the set of computer viruses as a result of their differing capabilities of classification.

However, we can go beyond the classifications of Chapter 4 if we recall the work from Chapter 3 on refinements of affordance-based models. Refinements allow us to relate different models of different abstraction levels. For instance, we may view a reproductive path in a number of different ways, e.g.:

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} s_3 \overset{a_3}{\longmapsto} s_4 \overset{a_4}{\longmapsto} s_5 \overset{a_5}{\longmapsto} s_6 \tag{5.1}$$

$$s \overset{a}{\longmapsto} s, s' \tag{5.2}$$

where (5.1) is a detailed reproductive path, and (5.2) is a less detailed (i.e., more abstract) reproductive path. We could define a refinement between reproduction models using (5.1) and (5.2) as the paths by mapping $s_1, \ldots, s_5$ to s, s_6 to s' and $a_1, \ldots, a_5$ to a. Alternatively, we may wish to the view the entities at different abstraction levels, e.g., $Ent = \{\mathtt{v}, \mathtt{OS}\}$ for the case where there is a computer virus $\mathtt{v}$ and an operating system $\mathtt{OS}$, or $Ent = \{\mathtt{v} + \mathtt{OS}\}$ where we conflate the virus and its environment, resulting in

the conglomerate entity $\mathtt{v} + \mathtt{OS}$.

If we think of a behaviour monitor as viewing the reproductive process of a computer virus at a set level of abstraction, then we can start to relate behaviour monitors in terms of refinements. For example, a reproductive path of a computer virus might be as follows:

$$s_1 \overset{a_1}{\longmapsto} s_2 \overset{a_2}{\longmapsto} s_3 \overset{a_3}{\longmapsto} s_4 \overset{a_4}{\longmapsto} s_5 \overset{a_5}{\longmapsto} s_6$$

However, suppose a given behaviour monitor B can only identify actions a_2, a_4, and therefore the reproductive path appears to be

$$s_1 \overset{a_2}{\longmapsto} t_1 \overset{a_4}{\longmapsto} s_6$$

from the perspective of B. Furthermore, there may be another behaviour monitor B' that is even less capable, and can only detect action a_2. The resulting reproductive path will appear to be

$$s_1 \overset{a_2}{\longmapsto} s_6 \quad .$$

We can therefore see the three transition systems as views of the same reproductive process at different levels of abstraction. There is another interesting connection with the entities, as in Chapter 4 we described how less capable behaviour monitors are able to view a smaller number of assisting entities. Therefore there could be five assisting entities in the original model, one corresponding to each of the five actions. However, B is only able top view actions a_2 and a_4, and so it makes sense only to include the two corresponding entities in its model. Likewise, for the model of B' there would be only one assisting entity. The case where there are no assisting entities would correspond naturally to the case where behaviour monitoring is turned off.

Furthermore, we can relate these refinements to the Maude specification of Intel 64 presented in Chapter 2. The Maude specification can be used an an interpreter for Intel 64. When a computer virus is executed using this interpreter, the resulting state transition system is detailed, and corresponds to the the situation in which the most capable behaviour monitor is able to witness every state transition in the execution of any program. In contrast, less capable behaviour monitors are only able to discern a fraction of all of the actions performed; the result is a "partial" transition system limited to only those actions visible. The least capable behaviour monitors are those which can discern no actions. If the behaviour monitor knows that the computer virus has reproduced, then the resulting transition system would be minimal, e.g., s $\overset{a}{\longmapsto}$ s'. In the case where reproduction cannot be established, the transition system could even be empty (i.e., $\longmapsto = \emptyset$).

5.2.9 Affordance-based Models and Multi-Reproducers

As we described in Chapters 3 and 4, it is possible to model and classify a variety of reproductive systems using affordance-based models. These models are ecological in nature, and provide a way of describing the reproductive process of a reproducer, as well as the entities in its environment which assist in the various stages of reproduction. An interesting extension to this work might be a formalisation of what might be termed *multi-reproducers*. Multi-reproducers are sets of entities that assist each other in the reproduction of that set. This kind of reproduction can be seen in several places, including k-ary malware [44], autocatalytic sets [82] and reproducing robots such as those developed recently at Cornell University [170].

Filiol describes k-ary malware, which are systems of k computer programs which are able to mutually reproduce [44]. k-ary malware are particularly difficult to detect, as each program k_i is not individually reproducing, but rather acts in concert with the other $k-1$ programs in mutual reproduction. Autocatalytic sets are somewhat similar. Kauffman [82] describes these as sets of molecules in which each molecule's production is catalysed by some other molecule(s) in the set. Therefore, these sets constitute reproductive wholes, in which the set is the reproducer. Another example of multi-reproduction is the Cornell University reproducing robots [170], in which a set of cubic robots is able to reproduce itself using other cubic robots lying around in its environment (see Figure 5.3).

Each of these systems could be represented by an affordance-based model in which the set itself is the reproducer. However, we could also think of each set of size n in terms of n reproduction models corresponding to the members of the set, in which each member is assisted in its reproduction by other members of the set, which appear in the model as entities. It seems likely that this would be possible using the existing formalisation of affordance-based models. However, an interesting question arises: how do we relate the model in which the set is the reproducer with the set of models in which the components are the reproducers? It may be possible to relate each component model with the set model using a construction along the lines of Definition 12 (p. 83), in which the component and its assisting entities are mapped to the reproducer in the set model. However, is there any way to relate the component models together?

It may be that this problem is a special case of the concerns described in Section 5.2.4, but further investigation is required to determine this.

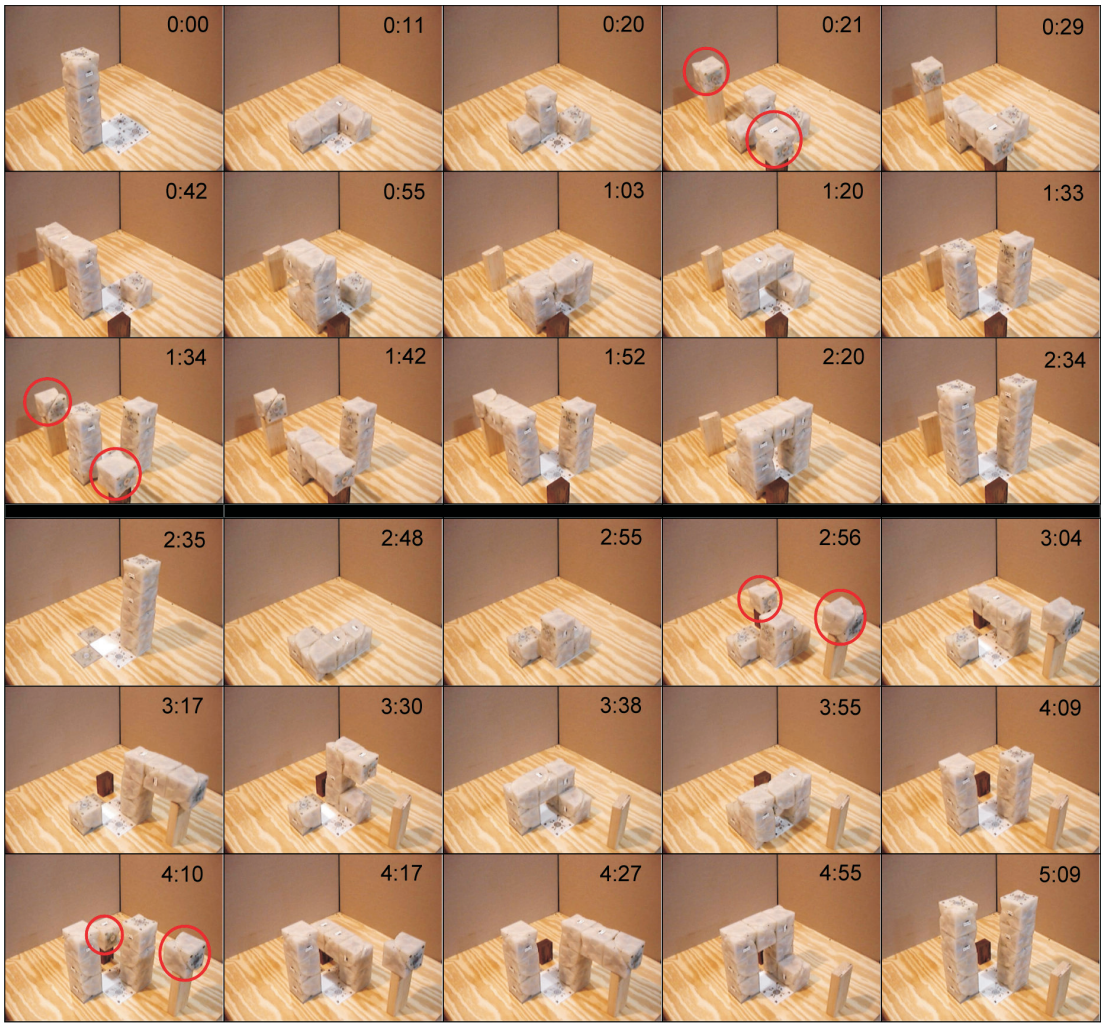


Figure 5.3: Reproducing robots developed at Cornell University by Zykov, Mytilinaios, Adams, and Lipson (reproduced with permission) [170]. Each cube is a robot. The 4-module conglomerate robot in the top-left frame moves through a variety of configurations, collecting other cubic robots from its environment (encircled in red), in order to create a copy of itself, visible in the bottom-right frame.

Appendix A: Intel 64 Specification

This appendix contains the Maude specification of a 10–instruction subset of the Intel 64 assembly programming language instruction set. These specifications were derived from the informal descriptions in the official Intel literature [75].

Below is a listing of the Maude functional modules `I-64-SYNTAX`, and `I-64-SEMANTICS`, which specify the syntax and semantics of the subset of the Intel 64 assembly programming language. These modules are the basis for all of the Intel 64 Maude reduction proofs in Chapter 2.

Maude Specification of Intel 64

```
*** i64.maude
*** Specification of a subset of the Intel 64 instruction set:
*** {MOV, ADD, SUB, TEST, XOR, AND, OR, PUSH, POP, NOP}.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** This module defines the syntax of a subset of I-64.
fmod I-64-SYNTAX is
  protecting INT .
  sorts Variable Expression Stack EInt .
  sorts Instruction InstructionSequence .
  subsort Instruction < InstructionSequence .
  subsorts Variable EInt < Expression .
  subsort Int < EInt .

  op dadd_,_ : Variable Expression -> Instruction [prec 20] .
  op dsub_,_ : Variable Expression -> Instruction [prec 20] .

  *** I-64 instructions
  op mov_,_ : Variable Expression -> Instruction [prec 20] .
```

```
  op add_,_ : Variable Expression -> Instruction [prec 20] .
  op sub_,_ : Variable Expression -> Instruction [prec 20] .
  op nop : -> Instruction .
  op push_ : Expression -> Instruction [prec 20] .
  op pop_ : Variable -> Instruction [prec 20] .
  op and_,_ : Variable Expression -> Instruction [prec 20] .
  op or_,_ : Variable Expression -> Instruction [prec 20] .
  op xor_,_ : Variable Expression -> Instruction [prec 20] .
  op test_,_ : Variable Expression -> Instruction [prec 20] .
  *** helper operations
  op stackPush : Expression Stack -> Stack .
  op stackPop : Stack -> Stack .
  op stackTop : Stack -> EInt .
  op _next_ : EInt Stack -> Stack [prec 15] .
  op stackBase : -> Stack .
  op msb : EInt -> EInt .
  op isZero : Expression -> EInt .
  op isZero : EInt -> EInt .
  op parity : EInt -> EInt .
  *** error messages
  op emptyStackError1 : -> Stack .
  op emptyStackError2 : -> EInt .
  *** I-64 registers
  ops eax ebx ecx edx ebp esp esi edi ip : -> Variable .
  *** I-64 EFLAGS register
  ops cf of sf af zf pf : -> Variable .
  *** equality operation
  op _is_ : EInt EInt -> Bool .
  op _is_ : Stack Stack -> Bool .
  *** extending the Int sort to include "undef"
  op undef : -> EInt .


  *** overloaded Boolean operations - these should be in proof scripts
  *** but OBJ3 v.2.0 won't allow it
  op _band_ : EInt EInt -> EInt [prec 35] .
  op _bor_ : EInt EInt -> EInt [prec 35] .
```

```
endfm

*** This module defines the semantics of the I-64 instructions
*** whose syntax is defined in I-64-SYNTAX.
fmod I-64-SEMANTICS is
  protecting I-64-SYNTAX .
  sort Store .

  *** stores
  ops s : -> Store .
  op initial : -> Store .

  *** operators for defining the semantics of I-64
  op _[[_]] : Store Expression -> EInt [prec 30] .
  op _[[stack]] : Store -> Stack [prec 30] .
  op _;_ : Store Instruction -> Store [prec 25] .
  op _;_ : InstructionSequence InstructionSequence ->
  InstructionSequence [gather (E e) prec 26] .

  *** variables for rewriting rules
  vars S S1 S2 S3 : Store .
  vars I I1 I2 I3 : EInt .
  vars INT INT1 INT2 : Int .
  vars V V1 V2 V3 : Variable .
  vars E E1 E2 E3 : Expression .
  vars ST ST1 ST2 : Stack .
  vars P1 P2 : InstructionSequence .

  *** evaluation of instruction sequences
  eq S ; (P1 ; P2 ) = (S ; P1) ; P2 .

  *** _is_ semantics
  eq I1 is I2 = (I1 == I2) .
  eq ST1 is ST2 = (ST1 == ST2) .
  *** the value of any integer in a store is the integer itself
  eq S[[I]] = I .
  *** initial values of variables and the stack
```

```
eq initial[[stack]] = stackBase .
ceq initial[[V]] = undef
  if V =/= ip .
eq initial[[ip]] = 0 .

*** Axioms to deal with static analysis of primitive
*** operators such as +, -, |, &, xor .
eq I | I = I .
eq I & I = I .
eq (I1 + I2) is (I3 + I2) = I1 is I3 .
eq (I1 + I2) is (I1 + I2) = true .
eq (I1 - I2) is (I1 - I2) = true .
eq (I1 | I2) is (I1 | I2) = true .
eq (I & S1[[V1]]) is (I & S2[[V2]]) = S1[[V1]] is S2[[V2]] .
eq isZero(I1 & I2) is isZero(I1 & I2) = true .
eq parity(I1 & I2) is parity(I1 & I2) = true .
eq msb(I1 & I2) is msb(I1 & I2) = true .
eq isZero(I1 | I2) is isZero(I1 | I2) = true .
eq parity(I1 | I2) is parity(I1 | I2) = true .
eq msb(I1 xor I2) is msb(I1 xor I2) = true .
eq isZero(I1 xor I2) is isZero(I1 xor I2) = true .
eq parity(I1 xor I2) is parity(I1 xor I2) = true .
eq msb(I1 | I2) is msb(I1 | I2) = true .
eq (I1 xor I2) is (I1 xor I2) = true .


*** I-64 instruction semantics
eq S ; and V,E [[V]] = S[[V]] & S[[E]] .
ceq S ; and V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of .
eq S ; and V,E [[stack]] = S[[stack]] .
eq S ; and V,E [[ip]] = S[[ip]] + 1 .
eq S ; and V,E [[sf]] = msb( S[[V]] & S[[E]] ) .
eq S ; and V,E [[zf]] = isZero( S[[V]] & S[[E]] ) .
eq S ; and V,E [[pf]] = parity( S[[V]] & S[[E]] ) .
eq S ; and V,E [[cf]] = 0 .
```

```
eq S ; and V,E [[of]] = 0 .

eq S ; or V,E [[V]] = S[[V]] | S[[E]] .
ceq S ; or V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of .
eq S ; or V,E [[stack]] = S[[stack]] .
eq S ; or V,E [[ip]] = S[[ip]] + 1 .
eq S ; or V,E [[sf]] = msb( S[[V]] | S[[E]] ) .
eq S ; or V,E [[zf]] = isZero( S[[V]] | S[[E]] ) .
eq S ; or V,E [[pf]] = parity( S[[V]] | S[[E]] ) .
eq S ; or V,E [[cf]] = 0 .
eq S ; or V,E [[of]] = 0 .

eq S ; xor V,E [[V]] = S[[V]] xor S[[E]] .
ceq S ; xor V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of .
eq S ; xor V,E [[stack]] = S[[stack]] .
eq S ; xor V,E [[ip]] = S[[ip]] + 1 .
eq S ; xor V,E [[sf]] = msb( S[[V]] xor S[[E]] ) .
eq S ; xor V,E [[zf]] = isZero( S[[V]] xor S[[E]] ) .
eq S ; xor V,E [[pf]] = parity( S[[V]] xor S[[E]] ) .
eq S ; xor V,E [[cf]] = 0 .
eq S ; xor V,E [[of]] = 0 .

eq S ; test V,E [[V]] = S[[V]] .
ceq S ; test V1,E [[V2]] = S[[V2]]
  if V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of .
eq S ; test V,E [[stack]] = S[[stack]] .
eq S ; test V,E [[ip]] = S[[ip]] + 1 .
eq S ; test V,E [[sf]] = msb( S[[V]] & S[[E]] ) .
eq S ; test V,E [[zf]] = isZero( S[[V]] & S[[E]] ) .
eq S ; test V,E [[pf]] = parity( S[[V]] & S[[E]] ) .
eq S ; test V,E [[cf]] = 0 .
eq S ; test V,E [[of]] = 0 .
```

```
eq S ; mov V,E [[V]] = S[[E]] .
ceq S ; mov V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip .
eq S ; mov V,E [[stack]] = S[[stack]] .
eq S ; mov V,E [[ip]] = S[[ip]] + 1 .

eq S ; add V,E [[V]] = (S[[V]] + S[[E]]) .
ceq S ; add V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip .
eq S ; add V,E [[stack]] = S[[stack]] .
eq S ; add V,E [[ip]] = S[[ip]] + 1 .

*** special version of add ("dynamic add") that keeps
*** results of additions within I-64 limits (2^32-1).
eq S ; dadd V,E [[V]] = (S[[V]] + S[[E]]) & 4294967295 .
ceq S ; dadd V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip .
eq S ; dadd V,E [[stack]] = S[[stack]] .
eq S ; dadd V,E [[ip]] = S[[ip]] + 1 .

eq S ; sub V,E [[V]] = (S[[V]] - S[[E]]) .
ceq S ; sub V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip .
eq S ; sub V,E [[stack]] = S[[stack]] .
eq S ; sub V,E [[ip]] = S[[ip]] + 1 .

*** special version of add ("dynamic sub") that keeps
*** results of additions within I-64 limits (2^32-1).
eq S ; dsub V,E [[V]] = (S[[V]] - S[[E]]) & 4294967295 .
ceq S ; dsub V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip .
eq S ; dsub V,E [[stack]] = S[[stack]] .
eq S ; dsub V,E [[ip]] = S[[ip]] + 1 .

eq S ; push E [[stack]] = stackPush(S[[E]],S[[stack]]) .
ceq S ; push E [[V]] = S[[V]]
```

```
    if V =/= ip .
  eq S ; push E [[ip]] = S[[ip]] + 1 .

  eq S ; pop V [[stack]] = stackPop(S[[stack]]) .
  eq S ; pop V [[V]] = stackTop(S[[stack]]) .
  ceq S ; pop V1 [[V2]] = S[[V2]]
    if V1 =/= V2 and V2 =/= ip .
  eq S ; pop V [[ip]] = S[[ip]] + 1 .

  ceq S ; nop [[V]] = S[[V]]
    if V =/= ip .
  eq S ; nop [[stack]] = S[[stack]] .
  eq S ; nop [[ip]] = S[[ip]] + 1 .

  *** Stack helper operations semantics
  eq stackPush(I,ST) = I next ST .
  eq stackPop(I next ST) = ST .
  eq stackPop(stackBase) = emptyStackError1 .
  eq stackTop(I next ST) = I .
  eq stackTop(stackBase) = emptyStackError2 .
endfm
```

Proof from Example 1, p. 29

```
*** vout.maude
*** Vout proofs for mov v1, v2. Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

fmod VOUT is
  pr I-64-SEMANTICS .
  ops s s' : -> Store .
  ops v1 v2 v3 z : -> Variable .
  ops value1 value2 : -> Int .

  eq s[[v1]] = value1 .
  eq s[[v2]] = value2 .
```

```
  eq s[[ip]] = value1 .
  eq s'[[ip]] = value2 .

  eq value1 is value2 = false .

endfm

*** Prove that v1 is in Vout( mov v1, v2) (should be FALSE)
red s[[v1]] is s ; mov v1, v2[[v1]] .

*** Prove that ip is in Vout( mov v1, v2) (should be FALSE)
red s[[ip]] is s ; mov v1, v2[[ip]] .

*** Prove that there is no other variable in Vout(mov v1,v2)
*** (should be TRUE)
*** NB: it is obvious that z =/= ip and z =/= v1
red s[[z]] is s ; mov v1, v2[[z]] .
```

Proof from Example 2, p. 30

```
*** vin.maude
*** Vin proofs for mov v1, v2. Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

fmod VIN1 is
  pr I-64-SEMANTICS .
  ops s s' : -> Store .
  ops v1 v2 v3 z : -> Variable .
  ops value1 value2 : -> Int .

  var V : Variable .

  eq s[[v2]] = value1 .
  eq s'[[v2]] = value2 .
  eq value1 is value2 = false .
  *** assume that s and s' are equivalent apart from v2
  ceq s[[V]] = s'[[V]]
```

```
      if V =/= v2 .
endfm

*** prove that v2 is in Vin(mov v1, v2) (should be FALSE)
red s ; mov v1, v2 [[v1]] is s' ; mov v1, v2 [[v1]] .

fmod VIN1A is
  pr I-64-SEMANTICS .
  ops s s' : -> Store .
  ops v1 v2 v3 z : -> Variable .
  ops value1 value2 : -> Int .

  var V : Variable .

  eq s[[ip]] = value1 .
  eq s'[[ip]] = value2 .
  eq value1 is value2 = false .
  *** assume that s and s' are equivalent apart from ip
  ceq s[[V]] = s'[[V]]
      if V =/= ip .
endfm

*** prove that ip is in Vin(mov v1, v2) (should be FALSE)
red s ; mov v1, v2 [[ip]] is s' ; mov v1, v2 [[ip]] .

fmod VIN2 is
  pr I-64-SEMANTICS .
  ops s s' : -> Store .
  ops v1 v2 v3 z : -> Variable .
  ops value1 value2 : -> Int .

  var V : Variable .

  *** assume that s and s' are equivalent apart from z
  ceq s[[V]] = s'[[V]]
      if V =/= z .
  eq s[[z]] =/= s'[[z]] = true .
```

```
  *** it is obvious that z =/= ip and z =/= v2

endfm

*** prove that no other variable is in Vin(mov v1, v2)
*** by showing that after executing mov v1,v2
*** the resulting stores are equivalent at Vout(mov v1, v2)
*** -- both reductions should be TRUE.

red s ; mov v1, v2 [[v1]] is s' ; mov v1, v2 [[v1]] .
red s ; mov v1, v2 [[ip]] is s' ; mov v1, v2 [[ip]] .
```

Proofs from Section 2.6.1

```
*** bistro.maude
*** Proof script for virus Win95/Bistro code fragments.
*** Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

fmod I-64-PROOF-1 is
  pr I-64-SEMANTICS .

  ops a b : -> InstructionSequence .

  *** Allomorph 1 fragment 0 -vs- Allomorph 2 fragment 0
  eq a =  push ebp ; mov ebp, esp .
  eq b =  push ebp ; push esp ; pop ebp .

endfm


*** Proof: equivalence w.r.t. every variable except ip

*** This should be FALSE.
red s ; a [[ip]] is s ; b [[ip]] .

*** These should be TRUE.
```

```
red s ; a [[stack]] is s ; b [[stack]] .
red s ; a [[ebp]] is s ; b [[ebp]] .

fmod I-64-PROOF-1 is
  pr I-64-SEMANTICS .

  op dword1 : -> EInt .
  ops c d : -> InstructionSequence .

  *** Allomorph 1 fragment 1 -vs- Allomorph 2 fragment 1
  eq c =  mov esi, dword1 ; test esi, esi .
  eq d =  mov esi, dword1 ; or esi, esi .

endfm


*** Proof: equivalence.

*** These should be TRUE.
red s ; c [[esi]] is s ; d [[esi]] .
red s ; c [[ip]] is s ; d [[ip]] .
red s ; c [[zf]] is s ; d [[zf]] .
red s ; c [[pf]] is s ; d [[pf]] .
red s ; c [[sf]] is s ; d [[sf]] .
red s ; c [[cf]] is s ; d [[cf]] .
red s ; c [[of]] is s ; d [[of]] .

*** QED

fmod I-64-PROOF-1 is
  pr I-64-SEMANTICS .
  op dword2 : -> EInt .
  ops e f : -> InstructionSequence .

  *** Allomorph 1 fragment 2 -vs- Allomorph 2 fragment 2
  eq e =  mov edi, dword2 ; or edi, edi .
  eq f =  mov edi, dword2 ; test edi, edi .
```

```
endfm

*** Proof: equivalence
*** These should be TRUE.
red s ; e [[edi]] is s ; f [[edi]] .
red s ; e [[ip]] is s ; f [[ip]] .
red s ; e [[zf]] is s ; f [[zf]] .
red s ; e [[pf]] is s ; f [[pf]] .
red s ; e [[sf]] is s ; f [[sf]] .
red s ; e [[cf]] is s ; f [[cf]] .
red s ; e [[of]] is s ; f [[of]] .

*** QED
```

Proofs from Section 2.6.2

```
*** zmorph.maude
*** Proof script for virus Win9x.Zmorph.A code fragments.
*** Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** Note that dadd and dsub are used in place of add and sub -
*** this is simply to use the versions of add and sub that are
*** designed for dynamic analysis.

fmod I-64-PROOF is
  pr I-64-SEMANTICS .
  ops g h : Store -> Store .
  var S : Store .

  *** Allomorph 1 -vs- Allomorph 2
  eq g(S) =   S ; mov edi, 2580774443 ;
                  mov ebx, 467750807 ;
                  dsub ebx, 1745609157 ;
                  dsub edi, 150468176 ;
                  xor ebx, 875205167 ;
```

```
                    push edi ;
                    xor edi, 3761393434 ;
                    push ebx ;
                    push edi .
  eq h(S) = S ; mov ebx, 535699961 ;
                    mov edx, 1490897411 ;
                    xor ebx, 2402657826 ;
                    mov ecx, 3802877865 ;
                    xor edx, 3743593982 ;
                    dadd ecx, 2386458904 ;
                    push ebx ;
                    push edx ;
                    push ecx .

endfm

*** Proof: equivalence modulo the stack
*** This should be true
red g(s)[[stack]] is h(s)[[stack]] .

*** QED
*** Check the state of the stack.
red g(s)[[stack]] .
```

Proofs from Example 3, p. 38

```
*** eic1.maude
*** Proof of equivalence in context.
*** Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** Note that dadd and dsub are used in place of add and sub -
*** this is simply to use the versions of add and sub that are
*** designed for dynamic analysis.

fmod I-64-PROOF is
  pr I-64-SEMANTICS .
```

```
  ops g  h psi : -> InstructionSequence .
  var S : Store .

  *** Allomorph 1 -vs- Allomorph 2
  eq g =  mov edi, 2580774443 ;
                   mov ebx, 467750807 ;
                   dsub ebx, 1745609157 ;
                   dsub edi, 150468176 ;
                   xor ebx, 875205167 ;
                   push edi ;
                   xor edi, 3761393434 ;
                   push ebx ;
                   push edi .
  eq h  = mov ebx, 535699961 ;
                   mov edx, 1490897411 ;
                   xor ebx, 2402657826 ;
                   mov ecx, 3802877865 ;
                   xor edx, 3743593982 ;
                   dadd ecx, 2386458904 ;
                   push ebx ;
                   push edx ;
                   push ecx .

  eq psi  = mov edi, 0 ;
                   mov ebx, 0 ;
                   mov ecx, 0 ;
                   mov edx, 0 .
endfm


*** Proof: semi-equivalence before executing psi

*** These should be TRUE
red s ; g [[stack]] is s ; h [[stack]] .
red s ; g [[ip]] is s ; h [[ip]] .
*** These should be FALSE
red s ; g [[edi]] is s ; h [[edi]] .
```

```
red s ; g [[ebx]] is s ; h [[ebx]] .
red s ; g [[ecx]] is s ; h [[ecx]] .
red s ; g [[edx]] is s ; h [[edx]] .

*** QED

*** Proof: equivalence after executing psi
*** These should be TRUE
red s ; g ; psi [[stack]] is s ; h ; psi [[stack]] .
red s ; g ; psi [[ip]] is s ; h ; psi [[ip]] .
red s ; g ; psi [[edi]] is s ; h ; psi [[edi]] .
red s ; g ; psi [[ebx]] is s ; h ; psi [[ebx]] .
red s ; g ; psi [[ecx]] is s ; h ; psi [[ecx]] .
red s ; g ; psi [[edx]] is s ; h ; psi [[edx]] .

*** QED
```

Proofs from Example 4, p. 40

```
*** eic2.maude
*** Proof of equivalence in context.
*** Use with i64.maude.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** Note that dadd and dsub are used in place of add and sub -
*** this is simply to use the versions of add and sub that are
*** designed for dynamic analysis.

fmod I-64-PROOF is
  pr I-64-SEMANTICS .
  ops g h psi' : -> InstructionSequence .
  var S : Store .

  *** Allomorph 1 -vs- Allomorph 2
  eq g =  mov edi, 2580774443 ;
                   mov ebx, 467750807 ;
                   dsub ebx, 1745609157 ;
```

```
                    dsub edi, 150468176 ;
                    xor ebx, 875205167 ;
                    push edi ;
                    xor edi, 3761393434 ;
                    push ebx ;
                    push edi .
  eq h =  mov ebx, 535699961 ;
                    mov edx, 1490897411 ;
                    xor ebx, 2402657826 ;
                    mov ecx, 3802877865 ;
                    xor edx, 3743593982 ;
                    dadd ecx, 2386458904 ;
                    push ebx ;
                    push edx ;
                    push ecx .

  eq psi' = pop edi ;
                    pop ebx ;
                    pop ecx ;
                    mov edx, ecx .
endfm


*** Proof: semi-equivalence before executing psi'
*** These should be TRUE
red s ; g [[stack]] is s ; h [[stack]] .
red s ; g [[ip]] is s ; h [[ip]] .
*** These should be FALSE
red s ; g [[edi]] is s ; h [[edi]] .
red s ; g [[ebx]] is s ; h [[ebx]] .
red s ; g [[ecx]] is s ; h [[ecx]] .
red s ; g [[edx]] is s ; h [[edx]] .

*** QED

*** Proof: equivalence after executing psi'
*** These should be TRUE
```

```
red s ; g ; psi' [[stack]] is s ; h ; psi' [[stack]] .
red s ; g ; psi' [[ip]] is s ; h ; psi' [[ip]] .
red s ; g ; psi' [[edi]] is s ; h ; psi' [[edi]] .
red s ; g ; psi' [[ebx]] is s ; h ; psi' [[ebx]] .
red s ; g ; psi' [[ecx]] is s ; h ; psi' [[ecx]] .
red s ; g ; psi' [[edx]] is s ; h ; psi' [[edx]] .

*** QED
```

Appendix B: Unix Computer Virus Specification

This appendix contains the Maude specification of the execution of a Unix Bash script computer virus. This specification was used in Chapter 3.

Maude Specification

```
*** bash0.maude
*** Execution of a Unix shell script computer virus
*** in a simplified version of the Bash interpreter.
*** Virus: cp $0 $0.copy
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

fmod SCRIPT-LANGUAGE is
  sorts    Var FileHandle Expression Statement .
  subsort  Var FileHandle < Expression < Statement .

  op $0 : -> Var .
  op  null : -> Expression .
  op  _.copy : Expression -> Expression .
  op  _.copy : FileHandle -> FileHandle .
  op  cp _ _ : Expression Expression -> Statement .
  op  subst : Expression Expression -> Expression .

  vars  E E' : Expression .
  var   FH : FileHandle .

  eq  subst(E, $0)  =  E .
  eq  subst(E, FH)  =  FH .
```

```
  eq  subst(E, E' .copy)  =  subst(E, E').copy .
endfm

view Statement from TRIV to SCRIPT-LANGUAGE is
  sort Elt to Statement .
endv

fmod FILE is
  pr  (LIST *(sort List{X} to Statements, op __ to _;_)){Statement} .
  sort  File .
  op  [_:_] : FileHandle Statements -> File .
endfm

view File from TRIV to FILE is
  sort Elt to File .
endv

fmod BASH is
  pr SCRIPT-LANGUAGE .
  pr (LIST *(sort List{X} to Filestore)){File} .
  sort  State .

  op  _ [$0:_] : Filestore Expression -> State .
  op  _|_ : State Statements -> State .
  op  fetch : FileHandle Filestore -> Statements .
  op  copy : Statements Expression Filestore -> Filestore .

  var  O : State .
  var  FS : Filestore .
  vars E E1 E2 : Expression .
  vars S S' : Statements .
  vars FH FH' : FileHandle .

  eq  O | nil  =  O .
  eq  FS [$0: E] | FH ; S  =  FS [$0: FH] | fetch(FH,FS) ; S .
  eq  fetch(FH, nil)  =  nil .
  eq  fetch(FH, [FH : S] FS)  =  S .
```

```
  cq  fetch(FH, [FH' : S] FS)  =  fetch(FH, FS)  if  FH =/= FH' .
  eq  FS [$0: E] | (cp E1 E2) ; S  =
      copy(fetch(subst(E,E1),FS), subst(E,E2), FS) [$0: E] | S .
  eq  copy(S, FH, nil)  =  [FH : S] .
  eq  copy(S, FH, [FH : S'] FS)  =  [FH : S] FS .
  cq  copy(S, FH, [FH' : S'] FS)  =  [FH' : S'] copy(S,FH,FS)
        if  FH =/= FH' .
endfm

fmod VIRUS-EXAMPLE is
  pr  BASH .
  op  virus : -> FileHandle .
endfm

red  [virus : cp $0 $0 .copy] [$0: null] | virus .
```

Appendix C: Bacteriophage Specification

This appendix contains two Maude specifications of the reproduction of the T4 bacteriophage virus. The first is an abstract, schematic model, and the second is a less abstract, more detailed model. Both of these specifications were used in Chapter 3.

T4 Bacteriophage Reproduction — Abstract Model

```
*** t4schematic.maude
*** A schematic model of the T4 bacteriophage life-cycle.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** The labelled transition system (the labels are literally
*** the labels of the rewrite rules).
mod SCHEMATIC-T4LTS is
  *** states of the transition system
  sort  State .

  *** the individual states
  ops  s1 s2 s3 s4 s5 s6 s7 : -> State .

  *** the transitions
  rl     [attach] :  s1  =>  s2 .
  rl  [penetrate] :  s2  =>  s3 .
  rl     [inject] :  s3  =>  s4 .
  rl [synthesise] :  s4  =>  s5 .
  rl     [mature] :  s5  =>  s6 .
  rl    [release] :  s6  =>  s7 .
endm

*** Search for a finalised term that cannot
```

```
*** be rewritten any further. (Result should
*** be a path from s1 to s7.)
search s1 =>! S:State .
show path 6 .

*** The entities.
mod SCHEMATIC-T4MODEL0 is
  pr SCHEMATIC-T4LTS .

  *** the entities in the model
  sort  Entity .
  *** just one entity: a T4 bacteriophage
  op  t4phage : -> Entity .
  *** entities in states relation
  op  _ in _  : Entity State -> Bool .

  var S : State .

  *** t4phage is present in all states
  eq  t4phage in S  =  true .
endm

*** An alternative model.
mod SCHEMATIC-T4MODEL1 is
  pr SCHEMATIC-T4LTS .

  *** the entities in the model
  sort  Entity .
  *** a T4 bacteriophage
  op  t4phage : -> Entity .
  *** and a cell
  op  theCell : -> Entity .
  *** entities in states relation
  op  _ in _  : Entity State -> Bool .

  var S : State .
```

```
  *** t4phage is present in all states
  eq  t4phage in S  =  true .
  *** theCell present in s1..s6
  eq  theCell in S  =  (S =/= s7) .
endm

*** Search for a state in which the cell
*** is not present. Should return a path
*** leading to s7.
search in SCHEMATIC-T4MODEL1 :
  s1 =>* S:State
  such that not(theCell in S:State) .
show path 6 .

*** Another alternative model.

mod SCHEMATIC-T4MODEL2 is
  pr SCHEMATIC-T4LTS .

  *** the entities in the model
  sort  Entity .
  *** a T4 bacteriophage
  op  t4phage : -> Entity .
  *** and a cell
  op  theCell : -> Entity .
  *** and its nucleus
  op  theNucleus : -> Entity .
  *** entities in states relation
  op  _ in _  : Entity State -> Bool .

  var S : State .

  *** t4phage is present in s1, s2, s6 and s7;
  eq  t4phage in S  =  (S =/= s3) and (S =/= s4) and (S =/= s5) .
  *** theCell and theNucleus are present in s1..s6
  eq  theCell in S  =  (S =/= s7) .
  eq  theNucleus in S  =  true .
```

```
endm

*** Search for states in which the t4phage appears.
*** Should return s1, s2, s6 and s7.
search in SCHEMATIC-T4MODEL2 :
  s1 =>* S:State
  such that t4phage in S:State .
```

T4 Bacteriophage Reproduction — Detailed Model

```
*** t4cell.maude
*** A model of the T4 bacteriophage life-cycle.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** The soup represents a solution which may appear
*** inside or outside a cell.
mod  SOUP{X :: TRIV}  is
  sort  Soup .
  subsort  X$Elt < Soup .

  op  empty  : -> Soup .
  op  _ _  : Soup Soup -> Soup [assoc comm id: empty] .
  op  _in_ : X$Elt Soup -> Bool .

  vars E E' : X$Elt .
  var  S : Soup .

  eq  E in empty  =  false .
  eq  E in E' S  =  (E == E') or (E in S) .
endm

*** The inner stuff represents what can be contained
*** in the soup, e.g., RNA and bacteriophages.
mod  INNERSTUFF  is
  sorts  RNA Nucleus T4BP InnerStuff .
  subsorts  RNA T4BP  <  InnerStuff .
  op  t4[_]  : RNA -> T4BP .
```

```
endm

*** View the sort Elt from the TRIV module as
*** the sort InnerStuff in the INNERSTUFF module.
view  InnerStuff  from TRIV to INNERSTUFF is
  sort Elt to InnerStuff .
endv

mod  ENTITY  is
  *** Set up lists of inner stuff.
  protecting  (SOUP *(sort Soup to InnerSoup,
  op empty to emptyIS,
  op _ _ to _:_,
  op _in_ to _inIS_))
  {InnerStuff} .

  sorts  Cell Entity .
  subsorts  InnerStuff Cell Nucleus <  Entity .

  op  cell  :  Nucleus InnerSoup -> Cell .
  op  [_]-[_] : Cell T4BP -> Cell .
  op  [_]<[_] : Cell T4BP -> Cell .
endm

view Entity from TRIV to ENTITY is
  sort Elt to Entity .
endv

mod  T4CELL-MODEL  is
  protecting  (SOUP *(sort Soup to State)){Entity} .
  protecting ENTITY .

  var   N : Nucleus .
  var   IS : InnerSoup .
  vars  T T1 T2 : T4BP .
  vars  C C1 C2 : Cell .
  vars  R R' : RNA .
```

```
rl [attach] :  C T  =>  [C]-[T] .
rl [penetrate] :  [C]-[T]  =>  [C]<[T] .
rl [inject] :  [cell(N, IS)]<[t4[R]]  =>  cell(N, (IS : R)) .
rl [synthesise] :
   cell(N,  (R : IS))  =>  cell(N, (R : IS : R)) .
rl [mature] :
   cell(N,  (R : IS))  =>  cell(N, (t4[R] : IS)) .
rl [release] :  cell(N, T : IS)  =>  T : IS .

op  _ in1 _  : Entity State -> Bool .
op  strip  : Cell -> Cell .
op  stripAll  : State -> State .
op  getRNA  : T4BP -> RNA .
op  _matches_  : RNA State -> Bool .
op  _matchesIS_  : RNA InnerSoup -> Bool .
op  _inCell_  : RNA Cell -> Bool .

var  S : State .
var  E : Entity .

eq  [[C]<[T1]]<[T2]  =  [[C]<[T2]]<[T1] .
eq  C in1 S  =  strip(C) in stripAll(S) .
eq  T in1 S  =  getRNA(T) matches S .
eq  E in1 S  =  E in S  [owise] .
eq  strip([C]-[T])  =  strip(C) .
eq  strip([C]<[T])  =  strip(C) .
eq  strip(cell(N,IS))  =  cell(N,emptyIS) .
eq  stripAll(empty)  =  empty .
eq  stripAll(C S)  =  strip(C) stripAll(S) .
eq  stripAll(E S)  =  E stripAll(S)  [owise] .
eq  getRNA(t4[R])  =  R .
eq  R matches empty  =  false .
eq  R matches T S  =  (R == getRNA(T)) or (R matches S) .
eq  R matches C S  =  (R inCell C) or  (R matches S) .
eq  R matches S  =  false  [owise] .
eq  R inCell [C]<[T]  =  R inCell T .
```

```
  eq  R inCell [C]-[T]  =  (R == getRNA(T)) or (R inCell C) .
  eq  R inCell cell(N,  IS)  =  R matchesIS IS .
  eq  R matchesIS emptyIS  =  false .
  eq  R matches R' IS  =  (R == R') or (R matchesIS IS) .
  eq  R matches T IS  =  (R == getRNA(T)) or (R matchesIS IS) .
endm

mod EXAMPLE is
  protecting T4CELL-MODEL .

  ops rna1 rna2 rna3 rna4 : -> RNA .
  ops n1 n2 : -> Nucleus .
  ops init init2 init3 : -> State .
  op initSmall : -> State .
  op chapter3example : -> State .
  op o : -> InnerSoup .
  ops cell1 cell2 : -> Cell .

  eq cell1 = cell(n1, emptyIS) .
  eq cell2 = cell(n2, emptyIS) .
  eq init  = cell1 cell(n2, emptyIS) t4[rna1] t4[rna2]
             t4[rna3] t4[rna4] .
  eq init2 = cell1 t4[rna1] .
  eq init3 = cell1 cell2 t4[rna1] .
  eq initSmall = cell1 t4[rna1] .
  eq chapter3example  = t4[rna1] cell1 t4[rna1] .
endm

*** search for states reachable from initSmall in which
*** cell1 is not present.
search [1, 10] in EXAMPLE :
  initSmall =>+ S:State  such that not(cell1 in1 S:State) .
show path 8 .

*** search for states reachable from initSmall in which
*** cell1 is not present, and t4[rna1] is present.
search [1, 12] in EXAMPLE :
```

```
  initSmall =>+ S:State  such that not(cell1 in1 S:State)
  and t4[rna1] in1 S:State .
show path 8 .
```

Appendix D: Anti-Virus Specification

This appendix contains the Maude specifications of different anti-virus behaviour monitors used in Chapter 4.

Maude Specification

```
*** av.maude
*** A Maude specification of different anti-virus ontologies,
*** and how they handle statement lists.
*** Used with Maude 2.3 built: Feb 14 2007 17:53:50

*** set up basic sorts for actions, events, entities

fmod ACTION is
  sorts Action .
endfm

fmod EVENT is
  sort Event .
endfm

*** set up views from trivial module so that we can have
*** lists and sets of these things

view Action from TRIV to ACTION  is
  sort Elt to Action .
endv

view Event from TRIV to EVENT is
  sort Elt to Event .
```

```
endv

*** Defines the library operations available to anti-virus
*** software.
fmod AV-LIBRARY is
  *** parameterise the LIST module with Action and Event -
  *** this gives us lists of Actions and Events
  pr LIST{Action} .
  pr LIST{Event} .
  sort Class .

  ops a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 : -> Action .

  *** Anti-virus software can observe a virus executing,
  *** identify the entities at work and then classify
  *** the resulting reproduction model.
  op observe : List{Action} -> List{Event} .
  op classify : List{Event} -> Class .
  op Assisted : -> Class .
  op Unassisted : -> Class .

  vars S : Action .
  vars SL : List{Action} .
  var C : Event .
  var CL : List{Event} .

  eq observe( S SL ) = observe(S) observe(SL) .
  eq classify( nil ) = Unassisted .
  ceq classify( CL ) = Assisted
    if CL =/= nil .
endfm


*** AV1 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** Calls to Windows Scripting Host are observable,
*** therefore WSH is a separate entity.
```

```
fmod AV1 is
  pr AV-LIBRARY .
  ops CreateObject Randomize GetFolder GetFile ScriptFullName
      Copy : -> Event .

  eq observe( a1 ) = nil .
  eq observe( a2 ) = CreateObject .
  eq observe( a3 ) = GetFolder .
  eq observe( a4 ) = GetFile ScriptFullName .
  eq observe( a5 ) = nil .
  eq observe( a6 ) = Copy .
endfm

*** baby virus: a1 a2 a3 a4 a5 a6
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .

*** AV1 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** Calls to Windows Scripting Host are observable,
*** therefore WSH is a separate entity.
fmod AV2 is
  pr AV-LIBRARY .

  ops s1 s2 s3 s4 s5 s6 : -> Event .

  eq observe( a1 ) = s1 .
  eq observe( a2 ) = s2 .
  eq observe( a3 ) = s3 .
  eq observe( a4 ) = s4 .
  eq observe( a5 ) = s5 .
  eq observe( a6 ) = s6 .
endfm

*** baby virus: a1 a2 a3 a4 a5 a6
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .
```

```
*** AV3 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** No events are observable for any action, i.e., behaviour
*** monitoring is turned off.
fmod AV3 is
  pr AV-LIBRARY .

  var A : Action .
  var LA : List{Action} .

  *** All actions result in no events being detected
  eq observe( LA ) = nil .
endfm

*** baby virus: a1 a2 a3 a4 a5 a6
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .

*** AV4 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** Calls to GetFolder (afforded by WSH) are observable,
*** therefore WSH is a separate entity.
fmod AV4 is
  pr AV-LIBRARY .
  ops CreateObject Randomize GetFolder GetFile ScriptFullName
      Copy : -> Event .

  eq observe( a1 ) = nil .
  eq observe( a2 ) = nil .
  eq observe( a3 ) = GetFolder .
  eq observe( a4 ) = nil .
  eq observe( a5 ) = nil .
  eq observe( a6 ) = nil .
endfm

*** baby virus: a1 a2 a3 a4 a5 a6
```

```
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .

*** AV5 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** Calls to GetFolder (afforded by WSH) are observable,
*** therefore WSH is a separate entity.
*** However, this virus does not use GetFolder, the line 3
*** of this new virus is
*** Set HOME = "C:\" .
*** Therefore this statement is unobservable.
fmod AV5 is
  pr AV-LIBRARY .

  ops CreateObject Randomize GetFolder GetFile ScriptFullName
      Copy : -> Event .

  eq observe( a1 ) = nil .
  eq observe( a2 ) = nil .
  eq observe( a3 ) = nil .
  eq observe( a4 ) = nil .
  eq observe( a5 ) = nil .
  eq observe( a6 ) = nil .
endfm

*** baby virus: a1 a2 a3 a4 a5 a6
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .

*** AV4 defines the behaviour monitoring capabilities of an
*** antivirus program.
*** Calls to GetFolder (afforded by WSH) are observable,
*** therefore WSH is a separate entity.
fmod AV6 is
  pr AV-LIBRARY .

  op a3' : -> Action .
```

```
  ops CreateObject Randomize GetFolder GetFile ScriptFullName
      Copy : -> Event .

  eq observe( a1 ) = nil .
  eq observe( a2 ) = nil .
  eq observe( a3 ) = GetFolder .
  eq observe( a3' ) = nil .
  eq observe( a4 ) = nil .
  eq observe( a5 ) = nil .
  eq observe( a6 ) = nil .
endfm

*** Baby virus version 1
red observe( a1 a2 a3 a4 a5 a6 ) .
red classify( observe( a1 a2 a3 a4 a5 a6 ) ) .

*** Baby virus version 2
red observe( a1 a2 a3' a4 a5 a6 ) .
red classify( observe( a1 a2 a3' a4 a5 a6 ) ) .
```

List of Figures

1.1 Cohen's formal definition of computer viruses. 4
1.2 Von Neumann's reproducing automaton. 9

2.1 Allomorphic fragments of Win95/Bistro. 32
2.2 Allomorphic fragments of Win9x.Zmorph.A. 35
2.3 Application of equivalence-in-context. 44

3.1 Reproduction in cellular automata. 65
3.2 The T4 bacteriophage virus. 66
3.3 Informal reproducer classification based on affordances. 68
3.4 Allowed refinements between classes of affordance-based models. 83
3.5 Refinement arrows between M, $M^{\#}$ and N. 85
3.6 A possible labelled transition system for a model of Langton's loop. . . 90
3.7 Taylor's classification of reproducers. 103

4.1 Unix shell script virus. 127
4.2 Virus.VBS.Archangel. 130
4.3 Labelled transition system for Virus.VBS.Archangel. 131
4.4 MINI-44 virus. 137
4.5 Virus.VBS.Baby. 141
4.6 A variant of Virus.VBS.Baby. 148

5.1 A metamorphic engine based on the Maude specification of Intel 64. . . 176
5.2 Ontological shifts for a biological system. 178
5.3 Reproducing robots developed at Cornell University. 184

Bibliography

[1] VX Heavens. `http://vx.netlux.org/`. Accessed 22nd March 2008.

[2] Hi-tech crime: The impact on UK business 2005. National Hi-Tech Crime Unit Report, 2006.

[3] Bryant Adams and Hod Lipson. A universal framework for self-replication. In *European Conference on Artificial Life (ECAL'03)*, pages 1–9, 2003.

[4] Leonard M. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology — CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 354–374, 1990.

[5] Eli Bachmutsky. Self-replicating loops & Ant, February 1999. Java program. `http://necsi.org/postdocs/sayama/sdsr/java/loops.java`. Accessed 7th April 2008.

[6] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335:131–146, 2005.

[7] Mark A. Bedau, John S. McCaskill, Norman H. Packard, Steen Rasmussen, Chris Adami, David G. Green, Takashi Ikegami, Kunihiko Kaneko, and Thomas S. Ray. Open problems in artificial life. *Artificial Life*, 6:363–376, 2000.

[8] Trevor Bench-Capon and Grant Malcolm. Formalising ontologies and their relations. In Trevor Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *Proc. 10th International Conf. on Database and Expert Systems Applications (DEXA'99)*, pages 250–259. Springer Lecture Notes in Computer Science volume 1677, 1999.

[9] Trevor Bench-Capon, Grant Malcolm, and Michael Shave. Semantics for interoperability: relating ontologies and schemata. In V. Marik, W. Retschitzegger, and O. Stepanovka, editors, *Proceedings of DEXA 2003*, pages 703–712. Springer Lecture Notes in Computer Science volume 2736, 2003.

[10] Paul Bernays. *Axiomatic Set Theory*. North-Holland, Amsterdam, 1958.

[11] Daniel Bilar. On callgraphs and generative mechanisms. *Journal in Computer Virology*, 3:299–310, 2007.

[12] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Abstract detection of computer viruses. Technical report, INRIA, 2005. Third Workshop on Applied Semantics (APPSEM'05). `http://hal.inria.fr/inria-00115368/en/`. Accessed 7th April 2008.

[13] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Toward an abstract computer virology. In *Proceedings of the Second International Colloquium on Theoretical Aspects of Computing (ICTAC 2005)*, volume 3722 of *Lecture Notes in Computer Science*. Springer, 2005.

[14] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. On abstract computer virology: from a recursion-theoretic perspective. *Journal in computer virology*, 1(3–4):45–54, 2006.

[15] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. A classification of viruses through recursion theorems. In S.B. Cooper, B. Löwe, and A. Sorbi, editors, *CiE 2007*, volume 4497 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2007.

[16] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. An implementation of morphological malware detection. In *17th European Institute for Computer Antivirus Research Annual Conference Proceedings (EICAR 2008)*, pages 47–62, 2008.

[17] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[18] Jean-Marie Borello, Ludovic Mé, and Éric Filiol. Limits of metamorphic viruses detection tools [*sic*]. *Annals of Telecommunications*. To appear.

[19] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.

[20] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[21] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[22] John Byl. Self-reproduction in small cellular automata. *Physica D*, 34:295–299, 1989.

[23] Philip L. Campbell. The denial-of-service dance. *IEEE Security & Privacy*, 3(6):34–40, 2005.

[24] Ero Carrera and Gergely Erdélyi. Digital genome mapping — advanced binary malware analysis. In *Virus Bulletin Conference*, September 2004.

[25] Fabricio Chalub and Christiano Braga. A modular rewriting semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, 2004.

[26] David M. Chess and Steve R. White. An undetectable computer virus. In *Virus Bulletin Conference*, September 2000.

[27] Mohamed R. Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In *Proceedings of the Fourth ACM Workshop on Recurring Malcode (WORM)*, pages 73–78, 2006.

[28] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46. ACM Press, 2005.

[29] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[30] E.F. Codd. *Cellular Automata*. Academic Press, New York, 1968.

[31] Fred Cohen. Computer viruses — theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[32] Fred Cohen. Computational aspects of computer viruses. *Computers and Security*, 8:325–344, 1989.

[33] Frederick B. Cohen. *It's Alive! The New Breed of Living Computer Programs*. John Wiley & Sons, 1994.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[35] Marcelo d'Amorim and Grigore Roşu. An equational specification for the Scheme language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.

[36] Richard Dawkins. *The Selfish Gene*. Oxford University Press, USA, 1990. First published 1976. ISBN: 0192860925.

[37] A. K. Dewdney. Computer recreations: In the game called core war hostile programs engage in a battle of bits. *Scientific American*, 250(5):14–22, May 1984.

[38] A. K. Dewdney. Computer recreations: A core war bestiary of viruses, worms and other threats to computer memories. *Scientific American*, March 1989.

[39] K. Eric Drexler. *Engines of Creation: The Coming Era of Nanotechnology*. Anchor Books, 1986.

[40] Éric Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):70–75, 2007.

[41] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[42] Azadeh Farzan, José Meseguer, and Grigore Roşu. Formal JVM code analysis in JavaFAN. In *Proceedings of Algebraic Methodology and Software Technology (AMAST) 2004*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147, 2004.

[43] Eric Filiol. *Computer Viruses: from Theory to Applications*. Springer, 2005. ISBN 2287239391.

[44] Eric Filiol. Formalisation and implementation aspects of k-ary (malicious) codes. *Journal in Computer Virology*, 3:75–86, 2007.

[45] Eric Filiol, Marko Helenius, and Stefano Zanero. Open problems in computer virology. *Journal in Computer Virology*, 1:55–66, 2006.

[46] Eric Filiol, Grégoire Jacob, and Mickaël Le Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3:23–37, 2007.

[47] Eric Filiol and Sébastien Josse. A statistical model for undecidable viral detection. *Journal in Computer Virology*, 3:65–74, 2007.

[48] Richard Ford and Eugene H. Spafford. Happy birthday, dear viruses. *Science*, 317:210–211, July 2007.

[49] Robert A. Freitas, Jr. and William P. Gilbreath, editors. *Advanced Automation for Space Missions*, NASA Conference Publication 2255. NASA Scientific and Technical Information Branch, 1982. Proceedings of the 1980 NASA/ASEE Summer Study, Santa Clara, California, USA. June 23–August 29, 1980.

[50] Robert A. Freitas Jr. and Ralph C. Merkle. *Kinematic Self-Replicating Machines*. Landes Bioscience, 2004. ISBN 1570596905. `http://www.molecularassembler.com/KSRM.htm`. Accessed 26th March 2008.

[51] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, 223:120–123, 1970.

[52] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *11th Workshop on Hot Topics in Operating Systems (HOTOS-X)*, 2007.

[53] Alejandra Garrido, José Meseguer, and Ralph Johnson. Algebraic semantics of the C preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, Department of Computer Science, University of Illinois at Urbana–Champaign, February 2006.

[54] Marius Gheorghescu. An automated virus classification system. In *Virus Bulletin Conference*, October 2005.

[55] James J. Gibson. The theory of affordances. *Perceiving, Acting and Knowing: Toward an Ecological Psychology*, pages 67–82, 1977.

[56] James J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, 1979. ISBN 0395270499.

[57] Kurt Gödel. *On formally undecidable propositions of Principia mathematica and related systems*. Oliver & Boyd, Edinburgh, 1962. Translation of Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I", 1931.

[58] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Massachusetts Institute of Technology, 1996. ISBN 026207172X.

[59] Joseph A. Goguen and Grant Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, Boston, 2000. ISBN 0792377575.

[60] Joseph A. Goguen, Timothy Walker, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000. ISBN 0792377575.

[61] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin. Constructing computer virus phylogenies. *Journal of Algorithms*, 26(1):188–208, 1998.

[62] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn, and Robert Richardson. 2006 computer crime and security survey. Computer Security Institute (CSI) and Federal Bureau of Investigation (FBI) report. `http://GoCSI.com/`. Accessed 7th April 2008.

[63] Sarah Gordon. Virus writers: The end of the innocence? IBM Antivirus Research Scientific Papers, September 2000. Presented at Virus Bulletin Conference.

[64] Sarah Gordon. What's in a name? Symantec Security Response White Paper, October 2002. `http://www.symantec.com/avcenter/reference/whatsinaname.pdf` Accessed 7th April 2008.

[65] Sarah Gordon. Virus and vulnerability classification schemes: Standards and integration. Symantec Security Response White Paper, February 2003. `http://www.symantec.com/avcenter/reference/virus.and.vulnerability.pdf` Accessed 7th April 2008.

[66] Allan Granoff and Robert G. Webster, editors. *Encyclopedia of Virology*, volume 3, pages 1414–15. Academic Press, 1999. Entry on"Virus Multiplication Cycle".

[67] Adam Greenfield. *Everyware: the dawning age of ubiquitous computing*. New Riders, Berkeley, CA, USA, 2006.

[68] Lutz H. Hamel and Joseph A. Goguen. Towards a provably correct compiler for OBJ3. In *Proceedings of the 6th International Symposium on Programming*

Language Implementation and Logic Programming, volume 844 of *Lecture Notes In Computer Science*. Springer, 1994.

[69] Martin Henz and Janardan Misra. Towards a framework for observing artificial life forms. In *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*, pages 23–30. IEEE Press, 2007.

[70] Michael Hilker and Christoph Schommer. SANA — security analysis in internet traffic through artificial immune systems. In Serge Autexier, Stephan Merz, Leon van der Torre, Reinhard Wilhelm, and Pierre Wolper, editors, *Workshop "Trustworthy Software" 2006*. IBFI, Schloss Dagstuhl, Germany, 2006.

[71] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*, chapter 16, page 499. Penguin, 2000.

[72] Zhi hong Zuo, Qing xin Zhu, and Ming tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8):2962–2966, 2005.

[73] Tim Hutton. John von Neumann's universal constructor. `http://www.sq3.org.uk/Evolution/JvN/`. Accessed 7th April 2008.

[74] Intel Corporation. *IA-32 Intel®Architecture Software Developer's Manual*, March 2006. `http://www.intel.com/design/pentium4/manuals/index_new.htm` Accessed 7th April 2008.

[75] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manual*, November 2007. `http://www.intel.com/products/processor/manuals/index.htm` Accessed 19th March 2008.

[76] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[77] Michael David Jones. Tevenphage.png. `http://en.wikipedia.org/wiki/Image:Tevenphage.p` Accessed 7th April 2008.

[78] Md. Enamul Karim, Andrew Walenstein, and Arun Lakhotia. Malware phylogeny using maximal pi-patterns. In *EICAR 2005 Conference: Best Paper Proceedings*, pages 156–174, 2005.

[79] Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1:13–23, 2005.

[80] Kaspersky Lab. Win95.Zmorph. `http://www.avp.ch/avpve/newexe/win95/zmorhp.stm`. Accessed 7th April 2008.

[81] Michael Katelman and José Meseguer. A rewriting semantics for ABEL with applications to hardware/software co-design and analysis. *Electronic Notes in Theoretical Computer Science*, 176:47–60, 2007.

[82] Stuart Kauffman. *At Home in the Universe: The Search for Laws of Complexity.* Penguin Books, 1996.

[83] Jeffrey O. Kephart. A biologically inspired immune system for computers. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV, Proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, Cambridge, Massachusetts, 1994.

[84] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[85] George J. Klir, editor. *Trends in General Systems Theory.* John Wiley & Sons, 1972.

[86] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[87] Jimmy Kuo and Desiree Beck. The common malware enumeration initiative. *Virus Bulletin*, pages 14–15, September 2005.

[88] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of Eleventh Working Conference on Reverse Engineering.* IEEE Computer Society Press, 2004.

[89] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10:135–144, 1984.

[90] Christopher G. Langton. Studying artificial life with cellular automata. *Physica D*, 22:120–149, 1986.

[91] Christopher G. Langton, editor. *Artificial Life: An Overview.* MIT Press, 1995. ISBN: 0262121891.

[92] Lars Löfgren. An axiomatic explanation of complete self-reproduction. *Bulletin of Mathematical Biology*, 30(3):415–425, 1968.

[93] Lars Löfgren. Relative explanations of systems. In G. Klir, editor, *Trends in General Systems Theory*. John Wiley & Sons, 1972.

[94] Jason D. Lohn and James A. Reggia. Automatic discovery of self-replicating structures in cellular automata. *IEEE Transactions on Evolutionary Computation*, 1(3):165–178, September 1997.

[95] Mark Ludwig. *The Little Black Book of Computer Viruses*. American Eagle Publications, Inc., Arizona, USA, 1990.

[96] Mark Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, Inc., Arizona, USA, 1995.

[97] Mark A. Ludwig. *Computer Viruses, Artificial Life and Evolution: The Little Black Book of Computer Viruses Volume II*. American Eagle Publications, Inc., Arizona, USA, 1993.

[98] Pavel O. Luksha. Formal definition of self-reproductive systems. In Russell K. Standish, Mark A. Bedau, and Hussein A. Abbass, editors, *Artificial Life VIII*. MIT Press, 2002.

[99] Pavel O. Luksha. The firm as a self-reproducing system. In *Proceedings of 47th International System Science Society Conference*, 2003.

[100] Charles C. Mann. Homeland insecurity. Atlantic Monthly, September 2002. `http://www.theatlantic.com/doc/200209/mann`. Accessed 7th April 2008.

[101] Edwin Martin. John Conway's game of life. Java program. `http://www.bitstorm.org/gameoflife/`. Accessed 7th April 2008.

[102] Barry McMullin. John von Neumann and the evolutionary growth of complexity: Looking backwards, looking forwards.... *Artificial Life*, 6:347–361, 2000.

[103] Barry McMullin. Thirty years of computational autopoiesis: A review. *Artificial Life*, 10:277–295, 2004.

[104] José Meseguer and Grigore Roşu. The rewriting logic semantics project. In *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005)*, volume 156 of *Electronic Notes in Theoretical Computer Science*, pages 27–56. Elsevier, 2005.

[105] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[106] Moinuddin Mohammed. Zeroing in on metamorphic computer viruses. Master's thesis, University of Louisiana at Lafayette, 2003.

[107] Jose Andre Morales, Peter J. Clarke, Yi Deng, and B. M. Golam Kibria. Testing and evaluating virus detectors for handheld devices. *Journal in Computer Virology*, 2(2), 2006.

[108] Chrystopher Nehaniv and Kerstin Dautenhahn. Self-replication and reproduction: Considerations and obstacles for rigorous definitions. In C. Wilke, S. Altmeyer, and T. Martinetz, editors, *Third German Workshop on Artificial Life: Abstracting and Synthesizing the Principles of Life*, pages 283–290. Verlag Harri Deutsch, 1998. ISBN: 3-8171-1591-1.

[109] Maureen A. O'Malley and John Dupré. Size doesnt matter: towards a more inclusive philosophy of biology. *Biology and Philosophy*, 22:155–191, 2007.

[110] Naoaki Ono and Takashi Ikegami. Self-maintenance and self-reproduction in an abstract cell model. *Journal of Theoretical Biology*, 206:243–253, 2000.

[111] Nicolas Oros and Chrystopher L. Nehaniv. Sexyloop: Self-reproduction, evolution and sex in cellular automata. In *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*, 2007.

[112] Umberto Pesavento. An implementation of von Neumann's self-reproducing machine. *Artificial Life*, 2:337–354, 1995.

[113] Jonathan Pincus and Brandon Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[114] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. DOI:10.1016/j.jlap.2004.03.002.

[115] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, 2007.

[116] Bill Purves, David Sadava, Gordon Orians, and Craig Heller. *Life, the Science of Biology*. Sinauer Associates, Inc., MA 01375, USA, 7th edition, 2003. ISBM: 0716798565.

[117] T. S Ray. An approach to the synthesis of life. In *Artificial Life II*, pages 371–408. Addison-Wesley, California, 1991. ISBN 0201525712.

[118] T.S. Ray. Evolution, complexity, entropy, and artificial reality. *Physica D*, 75:239–263, 1994.

[119] Chris Reed and Timothy J. Norman. A formal characterisation of Hamblin's action–state semantics. *Journal of Philosophical Logic*, 36:415–448, 2007. DOI: 10.1007/s10992-006-9041-z.

[120] Daniel Reynaud-Plantey. The Java mobile risk. *Journal in Computer Virology*, 2(2), 2006.

[121] Melanie R. Rieback, Bruno Crispo, and Andrew S. Tanenbaum. Is your cat infected with a computer virus? In *IEEE Pervasive Computing and Communications (PERCOM 2006)*, 2006.

[122] Robert Rosen. On a logical paradox implicit in the notion of a self-reproducing automaton. *Bulletin of Mathematical Biophysics*, 21:387–394, 1959.

[123] Robert Rosen. *Life Itself*. Columbia University Press, 1991.

[124] Robert Rosen. *Essays on Life Itself*. Columbia University Press, 1999. ISBN: 978-0231105118.

[125] Eugene Rosenberg, Omry Koren, Leah Reshef, Rotem Efrony, and Ilana Zilber-Rosenberg. The role of microorganisms in coral health, disease and evolution. *Nature Reviews Microbiology*, 5:355–362, 2007.

[126] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://www.invisiblethings.org/papers/redpill.html`, November 2004. Accessed 19th March 2008.

[127] Joanna Rutkowska. Subverting Vista^{TM} kernel for fun and profit. Black Hat Briefings 2006, Las Vegas, USA, August 2006. `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf` Accessed 19th March 2008.

[128] Ralf Sasse and José Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. *Electronic Notes in Theoretical Computer Science*, 176:29–46, 2007.

[129] Hiroki Sayama. A new structurally dissolvable self-reproducing loop evolving in a simple cellular automata space. *Artificial Life*, 5:343–365, 1999.

[130] Erwin Schrödinger. *What is Life?* Cambridge University Press, 1944.

[131] John F. Shoch and Jon A. Hupp. The "worm" programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.

[132] Moshe Sipper. An introduction to artificial life. *Explorations in Artificial Life (special issue of AI Expert)*, pages 4–8, September 1995.

[133] Moshe Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4:237–257, 1998.

[134] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. Principles of a computer immune system. In *1997 New Security Paradigms Workshop*. ACM Press, 1997.

[135] Eugene H. Spafford. Computer viruses as artificial life. *Journal of Artificial Life*, 1(3):249–265, 1994.

[136] Diomidis Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[137] Symantec Corporation. Virus naming conventions. `http://www.symantec.com/avcenter/vnameinfo.html`. Accessed 7th April 2008.

[138] Symantec Press Centre. Symantec warns computer users of destructive Christmas Day virus/worm mutation. `http://www.symantec.com/region/reg_ap/press/my_001219b.html`, 2000. Accessed 26th March 2008.

[139] Peter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005. ISBN 0321304543.

[140] Peter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference Proceedings*, 2001.

[141] Masahiro Tanaka. An application of information theory to biological evolution. *Journal of Theoretical Biology*, 85:789–806, 1980.

[142] Timothy John Taylor. *From Artificial Evolution to Artificial Life*. PhD thesis, University of Edinburgh, 1999. `http://www.tim-taylor.com/papers/thesis/`. Accessed 26th March 2008.

[143] Gary P. Thompson II. The quine page. `http://www.nyx.net/~gthompso/quine.htm`. Accessed 7th April 2008.

[144] Gerard Torenvliet. We can't afford it!: the devaluation of a usability term. *Interactions*, 10(4):12–17, July-August 2003.

[145] Sampo Töyssy and Marko Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2):109–119, 2006.

[146] Wiebe van der Hoek and Michael Wooldridge. On the logic of cooperation and propositional control. *Artificial Intelligence*, 164:81–119, 2005. DOI:10.1016/j.artint.2005.01.003.

[147] F.G. Varela, H.R. Maturana, and R. Uribe. Autopoiesis: the organization of living systems, its characterization and a model. *Biosystems*, 5:187–196, 1974.

[148] Luis P. Villarreal. Are viruses alive? *Scientific American*, 291(6):100–105, December 2004.

[149] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966. Edited by Arthur W. Burks.

[150] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, 2006.

[151] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *WORM '03: Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pages 11–18. ACM Press, 2003.

[152] Bruce Weber. Life. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, CA 94305-4115, USA, Spring 2006. `http://plato.stanford.edu/archives/spr2006/entries/life/`. Accessed 26th March 2008.

[153] Matt Webster. Algebraic specification of computer viruses and their environments. In Peter Mosses, John Power, and Monika Seisenberger, editors, *Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005). University of Wales Swansea Computer Science Report Series CSR 18-2005*, pages 99–113, 2005.

[154] Matt Webster. ASM-based modelling of self-replicating programs. Technical Report ULCS-05-005, Department of Computer Science, University of Liverpool,

UK, 2005. Presented at the 11th International Workshop on Abstract State Machines (ASM 2004).

[155] Matt Webster and Grant Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification. *Journal in Computer Virology*. DOI: 10.1007/s11416-008-0094-0. To appear.

[156] Matt Webster and Grant Malcolm. Formal affordance-based models of computer virus reproduction. *Journal in Computer Virology*. DOI: 10.1007/s11416-007-0079-4. To appear.

[157] Matt Webster and Grant Malcolm. Hierarchical components and entity-based modelling in artificial life. In *Proceedings of the Eleventh International Conference on Artificial Life (ALIFE XI)*. MIT Press, Cambridge, MA. To appear.

[158] Matt Webster and Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, December 2006. DOI: 10.1007/s11416-006-0023-z.

[159] Matt Webster and Grant Malcolm. Reproducer classification using the theory of affordances. In *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*, pages 115–122. IEEE Press, 2007.

[160] Matt Webster and Grant Malcolm. Reproducer classification using the theory of affordances: Models and examples. *International Journal of Information Technology and Intelligent Computing*, 2(2), 2007.

[161] Matt Webster and Grant Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification. In Vlasti Broucek and Eric Filiol, editors, *17th European Institute for Computer Antivirus Research Annual Conference Proceedings (EICAR 2008)*, pages 99–119, 2008.

[162] Stephanie Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320, 2007. arXiv:cs/0504045v1 [cs.CR].

[163] Norbert Wiener. *Cybernetics, or Control and Communication in the Animal and the Machine*. John Wiley & Sons, 1948.

[164] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge, 1994. Reprint of 1921 edition. Translated by D.F. Pears and B.F. McGuinness.

[165] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002. ISBN 047149691X.

[166] Christos Xenakis. Malicious actions against the GPRS technology. *Journal in Computer Virology*, 2(2), 2006.

[167] In Seon Yoo and Ulrich Ultes-Nitsche. Non-signature based virus detection: Towards establishing a unknown virus detection technique using SOM. *Journal in Computer Virology*, 2(3), 2006.

[168] InSeon Yoo. Visualizing Windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, 2004.

[169] Zhihong Zuo and Mingtian Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 47(6):627–633, 2004.

[170] Victor Zykov, Efstathios Mytilinaios, Bryant Adams, and Hod Lipson. Self-reproducing machines: A set of modular robot cubes accomplish a feat fundamental to biological systems. *Nature*, 435(7038):163–164, 2005.

Index

Abstract State Machines, 162
affordance, 121
affordances, 60, 62, 74, 123
algebra, 16, 163
algebraic specification, 16
  Maude, *see* Maude
arbitrariness of assistance, 108
artificial life, 9–10, 90, 93, 95, 96, 100, 102, 110, 118, 157
  systems, 9
astrobiology, 8
auto-catalytic sets, 183
automated theorem proving, 174
autopoiesis, 99, 111

bacteriophage, 66, 70, 72–75, 78–81, 117
behaviour monitor, 139, 181
behaviour monitoring, 121–122, 140–141
  configurations, 150
biological individual, 177

compiler, 67
computer virus
  $x86$ assembly, 136–137
  as artificial life, 14, 63, 95, 157, 166
  classification, 126
  classifications, 153–161
  definition
    formal, 4
    informal, 1
  detection
    complexity, 5, 18, 172
    types, 6–8
    undecidability, 5
  economic cost, 3
  Elk Cloner, 3
  entities, 124
  entity–components, 138
  history, 3
  hybrids, 179
  metamorphic, 4, 17, 124, 173
    detection methods, 45–53
    dynamic analysis, 31
    static analysis, 36
    types, 18
  source code, 66
  Unix Bash script, 76, 84, 127–129
  virtualization, *see* virtualization-based malware
  Virus.Java.Strangebrew, 126, 132
  Virus.VBS.Archangel, 129–131, 136, 143–144, 149
  Virus.VBS.Baby, 141, 144
  Win95/Bistro, 31
  Win9x.Zmorph.A, 35, 38
  writers, 2, 4
control-flow analysis, 46
Cornell self-reproducing robots, 183
cybernetics, 10
cybernetics and systems theory, 107

damaged cell, 67

data-flow analysis, 46

ecology, 75
equivalence, 28
    instruction sequences, 28, 30
    semi-, 28, 42
    stores, 28
equivalence in context, 29, 36, 38, 43, 172

false negative, definition, 6
false positive, definition, 6
formal approach, 10

Gödel numbering, 154, 163

Haskell, 145
hologenetics, 177

Intel 64, 22, 173
    instruction syntax, 23
    stack semantics, 26
Interrupt Service Routine, 136

Journal in Computer Virology, 5

k-ary malware, 183
Kerckhoff's principle, 4
Kleene's recursion theorem, 155
Kolmogorov complexity, 157, 179

Langton's loops, 9, 62, 64, 90–92, 116
life
    artificial, *see* artificial life
    classification, 60
    Game of, 63, 67, 93–94, 100
    informal definition, 8

malware, 1
Maude, 20, 53, 77, 80, 119, 123, 144
    as an interpreter, 27, 182
    Intel 64, 23, 24, 55, 172
    languages specified in, 57
    program semantics, 22, 27
    rewriting and reduction, 21
    store semantics, 22, 24
memes, xi
metrics for classification, 179, 180
multiagent systems, 112

natural selection, 88
neural network, 52

ontology, 178
operational semantics, 76

Peano arithmetic, 21
photocopy, 68
process algebra, 180

Red Pill, 174
reductionism, 114
rely/guarantee, 113
reproducing programs, 1
reproduction, 8, 60
    assisted, 88
      conjecture, 68
      theorem, 86
    cellular automata, 100
    classification, 96–104
    examples, 60
    exotic forms, 110
    exotic forms of, 95
    informal classification, 63
    informal definition, 60
    multi-, 183
    sexual, 110
    similar terms, 60
    strategies, 179
    time and information, 116
    triviality, 76, 110

unassisted, 88
conjecture, 69
logical paradox, *see* Rosen's paradox
theorem, 83
reproduction model
affordance-based, 74
allowed refinements, 82
Assisted Reproduction Theorem, 87
classification, 75–76, 126, 166
by aspects, 88
classification by metrics, 149–150
computer virus, 124
formal definition, 73
non-trivial vs. trivial, 76
permissiveness, 78
refinement, 81
Unassisted Reproduction Theorem, 84, 178
unassisted vs. assisted, 75
reproduction modelling
abstraction levels, 176
challenges, 59
reproduction models
motivation, 72
Rewriting Logic Semantics Project, 57, 119
Rosen's paradox, 97, 115

sandbox, 143, 147, 150
science vs. humanities, 10
self set, 73, 116
space exploration, 113
string matching, 151
systems theory, 10

T4 bacteriophage, *see* bacteriophage
theory of affordances, *see* affordances
Tierra, 9, 63, 102

variable
double meaning, 23
V_{in}, definition, 29
viral set, 73, 116, 124
virtualization-based malware, 54, 174–176
virus, 1
biological, 179, *see* bacteriophage
computer, *see* computer virus
Von Neumann, 59
model of computation, 3
reproducing automaton, 9, 62, 64, 65, 74, 101
V_{out}, definition, 29

worker bee, 8
worm, 1, 141, 156, 157, 159–161