Formal Affordance-based Models of Computer Virus Reproduction

Matt Webster and Grant Malcolm*

Abstract

We present a novel classification of computer viruses using a formalised notion of reproductive models based on Gibson's theory of affordances. A computer virus reproduction model consists of: a labelled transition system to represent the states and actions involved in that virus's reproduction; a notion of entities that are active in the reproductive process, and are present in certain states; a sequence of actions corresponding to the means of reproduction of the virus; and a formalisation of the actions afforded by entities to other entities. Informally, an affordance is an action that one entity allows another to perform. For example, an operating system might afford a computer virus the ability to read data from the disk. We show how computer virus reproduction models can be classified according to whether or not any of their reproductive actions are afforded by other entities. We give examples of reproduction models for three different computer viruses, and show how reproduction model classification can be automated. To demonstrate this we give three examples of how computer viruses can be classified automatically using static and dynamic analysis, and show how classifications can be tailored for different types of anti-virus behaviour monitoring software. Finally, we compare our approach with related work, and give directions for future research.

Keywords: Computer virus - Malware - Classification - Formal - Reproduction - Model - Affordance - Behaviour monitoring - Detection.

1 Introduction

We present a new approach to the classification of reproducing malware based on Gibson's theory of affordances [14, 15]. This approach arose from work

*Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK. Email: {matt,grant}@csc.liv.ac.uk.

on the related problem of reproduction model classification [38, 35], in which reproduction models can be classified as either unassisted or assisted, or in a multi-dimensional space based on predicates representing abstract actions, which capture whether or not parts of reproductive behaviour are afforded to the reproducer by another entity, or not.

The approach presented here differs from other models and classifications of computer viruses 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 [12], to low-level assembly code semantical descriptions in the vein of our earlier work on metamorphic computer virus detection [36]. We are able to distinguish formally between viruses that require the help of external agency and those that do not.

Computer viruses are detected in a variety of ways based on static and dynamic analysis. Behaviour monitoring is a form of dynamic analysis, which involves observing the behaviour of programs to find suspicious behaviours previously recorded. If a suspicious behaviour is observed, then the behaviour monitor can flag that program or process for further action or investigation. The capabilities of different behaviour monitors will vary, and therefore it is possible that a virus might be detectable using one behaviour monitor, but not another. In fact, a recent study by Filiol et al has shown this to be the case [12]. Suppose that an anti-virus software system has behaviour monitoring detection capabilities, as well as non-behaviour monitoring based detection capabilities, such as signature search. 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. This may be of particular use on systems where resources are limited, such as mobile computing systems.

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, and how this could be automated. 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. It is possible that similar approaches are already used as standard practice to improve the efficiency of anti-virus software, but we demonstrate that affordance-based reproduction models are able to give a theoretical explanation of this methodology.

The paper is organised as follows. First, we finish this section with an overview of existing computer virus classification methods. Then, in Section 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 formal computer virus 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.

We show how this flexibility of classification can be exploited in Section 3, where it is used 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 we compare our affordance-based reproduction models with other approaches to computer virus classification in the literature, and give directions for future research.

1.1 Related Work

The original problem of classification in computer virology lay in distinguishing computer viruses from non-reproducing programs [7], 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. However, further subclassifications of the class of computer viruses have been given in the literature. Adleman [1] divides the computer virus space into four disjoint subsets of computer viruses (benign, Epeian, disseminating and malicious). Spafford [30] gives five different generations of computer viruses which increase in complexity, from "Simple" to "Polymorphic". Weaver et al [34] have given a taxonomy of computer worms based on several criteria including target discovery and worm carrier mechanisms. Goldberg et al [17], Carrera & Erdélyi [5], Karim et al [20, 21] and Wehner [39] present classifications of malware based on phylogenetic trees, in which the lineage of computer viruses can be traced and a "family tree" of viruses constructed based on similar behaviours. Bonfante et al [3, 4] give a classification of computer viruses based on recursion theorems. Gheorghescu [13] gives a method of classification based on reuse of code blocks across related malware strains. A classification given by Bailey et al [2] is based on the clustering of malware that share similar abstract behaviours. In addition, both Filiol [11] and Ször [31] give comprehensive overviews of the state of the art of malware, including classification methods.

Most antivirus software vendors have their own schemes for malware naming, which involve some implicit classification, e.g., names like "W32.Wargbot" or "W97M/TrojanDropper.Lafool.NAA" give some information about the platform (e.g., Microsoft Windows 32–bit) and/or the primary reproductive mode of the virus (e.g., "trojan dropper"). Recently there have been efforts to standardise the many and varied malware naming schemes, e.g., the Common Malware Enumeration (CME) project [24] and the Computer Antivirus Research Organization (CARO) virus naming convention [28]. 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 [18, 13].

2 Computer Virus Classification

2.1 Formal Models of Computer Virus Reproduction

Our formal models of computer virus reproduction are related to our earlier work on the classification of reproduction models [38, 35]. 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 [14, 15]. 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.

For the purposes of our reproduction models, an affordance is a relation between entities in a reproduction system. 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. These reproduction models are shown to be useful to classify viruses according to different criteria, and based on whether they use external entities in their reproductive processes, and to what degree.

Two key elements of the states of a model are the entities that partake in the various states, and the actions that allow one state to evolve into another state. For a computer virus, these states could be abstract, or represent the states of the processor or virtual machine which executes the virus. The entities would be the parts of the computer system that enable the virus to reproduce, e.g., operating system APIs. In general, we assume that a model identifies the key entities or agents that take part in the process being modelled, and has some way of identifying whether a particular entity occurs in a particular state of affairs (e.g., a network service may only be available at certain times). We also assume that a model identifies those actions that are relevant to the computer virus being modelled, and describes which actions may occur to allow one state of affairs to be succeeded by another. Therefore, we will use a labelled transition system to model the dynamic behaviour of a virus.

This basic framework allows us to talk about reproductive processes: we can say that reproduction means that there is some entity v (a computer virus, say), some state s (the initial state of the reproductive process) with v present in state s (denoted "$v \mathrel{\varepsilon} s$" — see Definition 1 below) and some path $p = a_1, \ldots, a_n$ of actions, such that p leads, through a succession of intermediate states, to a state s' with $v \mathrel{\varepsilon} s'$. 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. We assume that the relation $v \mathrel{\varepsilon} s$ can be made abstract enough to accommodate an appropriate laxity in the notion of entity: i.e., we should gloss $v \mathrel{\varepsilon} s$ 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 [7], 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.

So far we have given an informal discussion of affordances, as well as a justification for using labelled transition systems to model the reproductive behaviour of computer viruses. We will now define affordances formally as the set of actions that one entity affords another. We write $\mathit{Aff}(e, e')$ for the actions that entity e affords to entity e'. The idea is that these are actions that are available to e' only in states where e is present. Thus, we require that a model carves up these actions in a coherent way: formally, $a \in \mathit{Aff}(e, e')$ implies that for any state s where e' is present, if the action a is possible (i.e., a leads to at least one state that succeeds s) then the entity e is also present in the state s.

This discussion is summarised in

Definition 1 *An* affordance-based computer virus reproduction model *is a tuple*

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

where

- $(S, A, \longmapsto)$ *is a labelled transition system, in which S is a set of states, A is a set of actions, and $\longmapsto \subseteq S \times A \times S$ is a relation giving labelled state transitions, e.g., $s_1 \overset{a_1}{\longmapsto} s_2$ means that state s_1 proceeds to state s_2 by action a_1;*

- *Ent is a set of 'entities' with $v \in Ent$ the particular computer virus that reproduces in the model;*
- *$\varepsilon \subseteq Ent \times S$ 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 v, i.e., p consists of a sequence $s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} \ldots \xmapsto{a_{n-1}} s_n$ with $s_i \xmapsto{a_i} s_{i+1}$ for $1 \leq i < n$, and with $r \ \varepsilon \ s_1$ and $r \ \varepsilon \ s_n$.*
- *$Aff : Ent \times Ent \to \mathcal{P}(A)$ such that, for any entities e and e', if $a \in Aff(e, e')$, then for all states s with $e' \ \varepsilon \ s$, if a is possible in s (i.e., $s \xmapsto{a} s'$ for some state s'), then $e \ \varepsilon \ s$.*

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?

The first question is actually a specific case of the general problem of reproducer classification. In evolutionary systems, there is variation both in the genome and the phenome, so this 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.

2.2 Classifying Computer Viruses

The key distinction in our classification 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 we shall see, it is not the computer viruses themselves that are classified as unassisted or assisted, but their reproduction models. In fact, it is possible to create affordance-based reproduction models of the same computer virus that are classified differently. In Section 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 2.3 and 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 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 behaviour 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 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, which we present as follows.

Definition 2 *A computer virus reproduction model can be classified as* unassisted *iff for all actions a in the reproducer's path p, there is no entity e such that $e \neq r$ and $a \in Aff(e, r)$. Conversely, a computer virus reproduction*

```
     ...
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 1: Statements from the Unix shell script virus showing use of `echo` and `chmod`.

model can be classified as assisted *iff there is some action a in the path p such that $a \in \mathit{Aff}(e, r)$ for $e \neq r$.*

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.

2.3 Modelling a Unix Shell Script Virus

The virus given in Figure 1 is a Unix shell script virus which runs when interpreted using the Bourne-again shell (Bash). The first three statements of the virus define three variables that contain an encrypted version of 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 1) corresponds to the transition $s_i \xmapsto{a_i} s_{i+1}$. 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

$$s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} \ldots \xmapsto{a_8} s_9$$

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 know 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 $a_4, a_5, a_6, a_7 \in \mathit{Aff}(\texttt{echo}, v_B)$ and $a_8 \in \mathit{Aff}(\texttt{chmod}, v_B)$. Since we know that these actions are afforded by other entities to the reproducer, logically, these entities must be present in the states preceding these actions, in line with condition 4 of Definition 1. Therefore, $\texttt{echo}\ \varepsilon\ s_3$, $\texttt{echo}\ \varepsilon\ s_4$, $\texttt{echo}\ \varepsilon\ s_5$, $\texttt{echo}\ \varepsilon\ s_6$ and $\texttt{chmod}\ \varepsilon\ s_7$. In addition, we know that the Bash virus reproduces, and therefore 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 2 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. We will denote the component of the respective models by using the same subscript, so that S_{M_B} refers to the set of states of reproduction model M_B, and $S_{M'_B}$ refers to the set of states of 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 $Aff_{M'_B}(e, e') = \emptyset$ for all $e, e' \in Ent_{M'_B}$. 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 2, M'_B is classifed as an unassisted reproduction model.

2.4 Modelling Virus.VBS.Archangel

Archangel (see Figure 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 `ScriptName`, 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 an Windows system 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 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:

$$p_{M_A} = s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} \ldots \xmapsto{a_{32}} s_{33}$$

There are two different objects which enable the Archangel virus at diffferent 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

```
     ...
    If Not fso.FolderExists(newfolderpath) Then
6     Set newfolder = fso.CreateFolder(newfolderpath)
    End If
7   fso.Copy Wscript.ScriptName, "C;\WINDOWS\SYSTEM\fun.vbs", True
8   fso.Move "C:\WINDOWS\SYSTEM\*.*","C;\WINDOWS\MyFolder\"
     ...
11  set fso=CreateObject("Scripting.FileSystemObject")
    If Not fso.FolderExists(newfolderpath) Then
12    Set newfolder = fso.CreateFolder(newfolderpath)
    End If
13  fso.Copy Wscript.ScriptName, "C;\MyFolder", True
14  fso.Copy Wscript.ScriptName, "C;\WINDOWS\SYSTEM\fun.vbs", True
15  fso.Move "C;\WINDOWS\SYSTEM32","C:\WINDOWS\SYSTEM"
16  fso.Copy Wscript.ScriptName, "C;\WINDOWS\SYSTEM\SYSTEM32\
                                         fun.vbs", True
17  fso.Copy Wscript.ScriptName, "C;\WINDOWS\StartMenu\Programs\
                                         StartUp\fun.vbs", True
18  fso.Delete "C:\WINDOWS\COMMAND\EBD\AUTOEXEC",True
19  fso.Delete "C:\WINDOWS\Desktop\*.*"
20  fso.Copy Wscript.ScriptName, "C:\\fun.vbs", True
21  set shell=wscript.createobject("wscript.shell")
     ...
```

Figure 2: Statements from Virus.VBS.Archangel showing the use of `fso` and `WScript`.

is also an entity which must appear in the model, and therefore we define $\mathit{Ent}_{M_A} = \{v_A, \mathtt{wsh}\}$, where v_A is the Archangel virus and `wsh` is the Windows Script Host, of which v_A is the reproducer in this model. Since the Windows Script Host allows the virus to create an instance of the `FileSystemObject` class, as well as access the `WScript.ScriptName` variable, we know that the statements in which these object references appear are actions that must be afforded by the Windows Script Host to the Archangel virus. The object `fso` of class `FileSystemObject` appears in statements 6, 7, 8 and 11–20; `WScript` appears in statements 7, 13, 14, 16, 17, 20 and 21. The actions that correspond to these statements are those actions with those numbers as subscripts and therefore $a_i \in \mathit{Aff}_{M_A}(\mathtt{wsh}, v_A)$ for $6 \leq i \leq 8$ and $11 \leq i \leq 21$. By Definition 1 we know that the Windows Script Host must be present in every state preceding these actions, and so $\mathtt{wsh}\ \varepsilon_{M_A}\ s$ for each state s in which one of these actions is possible (i.e., s_6, s_7, s_8 and so on — for the complete list please consult Figure 3). 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 2, 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 $\mathit{Aff}_{M'_A}(e, e') = \emptyset$ for all $e, e' \in \mathit{Ent}_{M'_A}$. 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 2, the computer virus reproduction model M'_A is classified as unassisted.

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. Space limitations do not allow us to include the full output of the decompiler (which is over

$$\begin{tikzcd}
 & & & & & & & & s_{13} \arrow[r, "a_{13}"] & s_{14} \arrow[r, "a_{14}"] & \cdots \arrow[r, "a_{32}"] & s_{33} \\
 & & & & s_7 \arrow[r, "a_7"] & s_8 \arrow[r, "a_8"] & \cdots \arrow[r, "a_{11}"] & s_{12} \arrow[ur, "a_{12}"] \arrow[dr, "a_{13}"'] & & & & \\
 & & & & & & & & s_{34} \arrow[rr, "a_{14}"'] & & \cdots \arrow[r, "a_{32}"'] & s_{53} \\
s_1 \arrow[r, "a_1"] & s_2 \arrow[r, "a_2"] & \cdots \arrow[r, "a_5"] & s_6 \arrow[ruu, "a_6"] \arrow[rdd, "a_7"'] & & & & & & & & \\
 & & & & & & & & s_{59} \arrow[r, "a_{13}"] & s_{60} \arrow[r, "a_{14}"] & \cdots \arrow[r, "a_{32}"] & s_{79} \\
 & & & & s_{54} \arrow[rr, "a_8"] & & \cdots \arrow[r, "a_{11}"] & s_{58} \arrow[ur, "a_{12}"] \arrow[dr, "a_{13}"'] & & & & \\
 & & & & & & & & s_{80} \arrow[rr, "a_{14}"'] & & \cdots \arrow[r, "a_{32}"'] & s_{99}
\end{tikzcd}$$

Figure 3: Labelled transition system for Virus.VBS.Archangel. The reproductive path is indicated by a dashed line.

500 lines); however, we present an overview of Strangebrew's reproductive behaviour for the purposes of 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 [12]. The use of abstract actions does not compromise the accuracy of classification as unassisted or assisted; 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.

Using this assumption we will define a 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 of the virus presented above, represent the behaviour of the Strangebrew virus:

a_1 = Search for host file containing the virus.
a_2 = Open host file.
a_3 = Find viral code in host file.
a_4 = Read in viral code.
a_5 = Close host file.
a_6 = Search for a file to infect.
a_7 = Open file to infect.
a_8 = Find insertion point.
a_9 = Write viral code to file.
a_{10} = Close infected file.

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}:

$$s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} \ldots \xmapsto{a_{10}} s_{11}$$

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 $\mathit{Ent}_{M_S} = \{v_S, \mathtt{raf}\}$, 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 $a_2, a_7 \in \mathit{Aff}_{M_S}(\mathtt{raf}, v_S)$. By Definition 1, we know that any entity which affords an action must be present in all states that precede that action, and therefore $\mathtt{raf}\ \varepsilon_{M_S}\ s_2$ and $\mathtt{raf}\ \varepsilon_{M_S}\ s_7$. We also know that the virus, as the reproducer in the model, must be present in the initial and final states, and so $v_S\ \varepsilon_{M_S}\ s_1$ and $v_S\ \varepsilon_{M_S}\ s_{10}$.

By Definition 2, 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 fulfill 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 $\mathit{Aff}_{M'_S}(e, e') = \emptyset$ for all $e, e' \in \mathit{Ent}_{M'_S}$. Since the only entity is the reproducer, we need only state the minimal assumption from Definition 1 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.

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 3.2 and 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 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 [7], and even for a particular known virus, can be a difficult problem in itself.

The limitations of 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 elsewhere in the literature (e.g., ch. 5, [11]). Overall, classification by automated means is possible but limited, as are most other forms of classification for virus detection.

3.1 Behaviour Monitoring and Classification

In Section 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., [27, 33, 40, 26]). 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 those that do 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

```
     ...
2   Set FSO = CreateObject("Scripting.FileSystemObject")
3   Set HOME = FSO.FindFolder(".")
4   Set Me_ = FSO.FindFile(WScript.ScriptName)
     ...
6   Me_.CopyFile(Baby)
```

Figure 4: Statements from Virus.VBS.Baby showing the use of external methods and attributes.

scanning, all viruses are classified as unassisted. Therefore, all viruses with a unassisted classification are impossible to detect at run-time by behaviour monitoring, whereas those classified as assisted have detectable behaviours can therefore 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.

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) 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.FindFolder(...)` method to access the directory in which the virus is executing. In line 5 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 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 [11], which are are often very implementation-specific (see, e.g., [31]). 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 `FindFolder()` method, statement 4 references the `FindFile()` method and `ScriptName` attribute, and statement 6 refers to the `CopyFile()` 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, [11]). 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.

3.3 Static Analysis of Virus.VBS.Archangel

In Section 2.4 we described Archangel (see Figure 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 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`, `Copy`, `ScriptName`, `Move`, `CreateObject`, `Delete`, `CreateShort-Cut`, `ExpandEnvironment`, `WindowStyle`, `Save`, `CreateTextFile`, `Write-Line`, `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.

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 3.2.

A specification of an anti-virus behaviour monitoring program was written in Maude [37] — a formal high-level language based on rewriting logic and algebraic specification [6]. Maude is strongly related to its predecessor, OBJ [16], a formal notation and theorem prover based on equational logic and algebraic specification. Like OBJ, Maude can be used for software specification [25], as abstract data types can be defined using theories in membership equational logic, called functional 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 we can 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 .
```

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 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 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 FindFolder FindFile ScriptName
    CopyFile : -> Event .
eq observe( a1 ) = nil .
eq observe( a2 ) = CreateObject .
eq observe( a3 ) = FindFolder .
eq observe( a4 ) = FindFile ScriptName .
eq observe( a5 ) = nil .
eq observe( a6 ) = CopyFile .
```

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 `FindFolder()` method:

```
eq observe( a1 ) = nil .
eq observe( a2 ) = nil .
eq observe( a3 ) = FindFolder .
eq observe( a4 ) = nil .
eq observe( a5 ) = nil .
eq observe( a6 ) = nil .
```

Action `a3` corresponds to the following statement in the Baby virus:

```
     ...
2   Set FSO = CreateObject("Scripting.FileSystemObject")
3   Set HOME = "\"
4   Set Me_ = FSO.FindFile(WScript.ScriptName)
     ...
6   Me_.CopyFile(Baby)
```

Figure 5: Statements from a variant of Virus.VBS.Baby that does not use the `FindFolder()` method.

```
Set HOME = FSO.FindFolder(".")
```

To show how different viruses are classified, we will define a variant of the Baby virus that does not use the `FindFolder()` method (see Figure 5). 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 `FindFolder()` are observable, and therefore action `a3'`, which does not use `FindFolder()`, 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.

3.5 Comparing Behaviour Monitor Configurations

In the static analysis examples presented in Sections 3.2 and 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 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 how based on 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, [31]) 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 3.6.

3.6 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 certain virus are observable by the behaviour monitoring software that an accurate (or unique) behaviour signature is 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.

3.6.1 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 which we might class as being the responsibility of 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 that behaviour monitoring.

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 VBS viruses in Sections 3.2 and 3.3), or by dynamic analysis (in the case of the Baby virus classified using Maude in Section 3.4).

Algorithms for static analysis-based classification of computer viruses into either assisted or unassisted classes can be implemented using string matching algorithms. A computer virus programmed in any executable language can be represented as a string of binary digits. We might wish to define a set of "components", which are instruction substrings, 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) [23, 10], then we can classify any virus as (un)assisted 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 Section 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 a simple extension of existing anti-virus behaviour monitoring software.

4 Conclusion

We have shown how it possible to classify reproducing malware, such as computer viruses, using formal affordance-based models of reproduction. We are able to formalise a reproductive process using a labelled transition system, and divide up the environment of a computer virus into separate entities, of which the computer virus (as the reproducer in the reproductive system) is one. Then, we can attribute different actions in the reproductive process to different entities, and based on these dependencies classify the computer virus as unassisted (if the virus is not afforded any of the actions in its reproductive path by other entities) or assisted (if the virus is afforded actions in its reproductive path by other entities). In addition, computer viruses that are classified as assisted can be compared using metrics based on arbitrary criteria, e.g., whether the number of observable behaviours is enough to generate an accurate behavioural signature.

We have shown how reproduction models and their classifications can be described formally, and how this process can be automated using algorithms. We have shown how metrics for classificaiton of assisted computer viruses might be implemented by giving a simple example of such a metric. We have applied our approach to a variety of real-life computer viruses written in the Visual Basic Script, Bash script and Java programming languages.

We have shown that classification of a computer virus as unassisted or assisted depends notionally on whether it is "dependent" on external entities for its reproductive behaviour, or not. By constructing our notion of externality with respect to a particular anti-virus behaviour monitor, the resulting classification divides computer viruses into those behaviour can be observed (assisted), or not (unassisted). By modifying the definition of the anti-virus software being modelled, the viruses can be easily re-classified to suit other types of anti-virus software.

We discussed in Section 3 how this classification might be applied to computer virus detection by enabling prioritisation of detection. For example, a set of viruses classified as unassisted will not be detectable at run-time by behaviour monitoring, and therefore we can concentrate our efforts on those virus in this class for detection by non-behavioural means. It may be the case that this process has already been implemented by computer anti-virus software; however, we present this as a logical consequence of our theoretical framework, and as evidence of the explanatory power of the formalism described in this paper.

We have shown via multiple reproduction models and classifications of the same virus (see Sections 2.3, 2.4, 2.5, 3.2 and 3.3) that our reproduction models and classifications are sufficiently unconstrained so as to allow flexibility of classification, and therefore it might seem that our classification is arbitrary. We consider this flexibility rather than arbitrariness, however, as it allows for the classification of computer viruses towards more efficient detection methods for anti-virus software, as given in Section 3. Therefore, once we have settled upon a fixed notion of externality, our classification provides the means to classify viruses in a formal and useful way to help improve the possibility of detection. Furthermore, through this classification we have introduced a means to compare formally the abilities of different anti-virus software that employ behaviour monitoring. As given in Section 3.5, the anti-virus software most able to detect viruses by behaviour monitoring will be those whose ontologies minimise the classification of viruses as unassisted, and maximise the numbers of viruses classified as assisted.

4.1 Comparison with Other Approaches

Computer virus classification schemes are numerous and diverse. While the means of a particular classification might be objective, the decision of preference of one classification over another can often be subjective; in this sense classification is in the eye of the beholder. Consequently it is difficult to assess rationally how well our classification works in comparison to those that have come before. Most classifications arise from some insight into the universe of objects being classified, and therefore the only requirement upon a classification being considered “worthy” is that it should have some explanatory power. Therefore, instead of attempting a futile rationalization of our classification versus the many interesting and insightful classifications of others, such as those presented Section 1, we will delineate the explanatory power of our approach, pointing out any relevant similarities to other approaches.

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 [32], who makes the distinction between unassisted and assisted reproduction with respect to artificial life.

Many other formal descriptions of computer viruses are based on descriptions of functionality and behaviour. For example, Cohen describes viral behaviour using Turing machines [8], Adleman uses first-order logic [1] and Bonfante et al [3, 4] base their approach on recursion theorems. Our approach differs in that the focus is not the virus's behaviour, but rather on the ecology of the virus, i.e., the environment in which reproduction takes place. For example, we might consider an operating system or a network to be essential parts of the virus's environment which facilitate reproduction, and by casting them as external entities we can then classify as (un)assisted, or by using metrics as given in Section 3.6.

Our approach bears some similarities to the work of Filiol et al [12] on their formal theoretical model of behaviour-based detection, which uses abstract actions (similar to those used in Section 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 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.

Our classification of computer viruses is a special case of the construction and classification of reproduction models from our earlier work [38, 35], 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 Spafford [30] in his description of computer viruses as artificial life, and by Cohen's treatise [9] 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 [22, 29, 19], and in that sense we hope that the formal relationship between computer viruses and other life forms has been further demonstrated by this paper, and could assist in the application of concepts from the study of natural and artificial life to problems in the field of computer virology. 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 Spafford and Cohen: that computer viruses are not merely a dangerous annoyance or a computational curiosity, but a life form in their own right.

4.2 Future Work

In Section 3.6 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 with a certain level of certainty a given viral behaviour. 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.

In Section 3 we described some methods for automatic classification by static and dynamic analysis. A natural extension of this work would be to describe these methods formally, perhaps by using the formal definition of reproduction models as a starting point. A useful application would be formal proofs of the assertions made informally in Section 3.1, e.g., that all computer virus reproduction models are classified as unassisted when that model describes a computer virus executed within a sandbox.

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. As mentioned before, there are some similarities between our approach and the recent work by Filiol et al [12] 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 antivirus ontologies. In future we would like to explore this relationship further, perhaps by generating a set of benchmarks based on our formal reproduction models and classifications, similar to those given in [12].

Recent work by Bonfante et al [3] 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.

Following on from earlier work [35, 38], it might also be possible to further sub-classify the space of computer viruses using notions of abstract actions such as the sets of actions corresponding to the self-description or reproductive mechanism of the computer virus. We might formalise this by defining predicates on the actions in a reproduction model; e.g., one predicate might hold for all actions which are part of the payload, i.e., that part of the virus that does not cause the virus to reproduce, but instead produces some side-effect of virus infection, for instance, deleting all files of a certain type. We could then classify a computer virus reproduction model based on whether the actions, for which the predicate holds, are afforded by entities other than the computer virus. Defining a number n of predicates would therefore result in up to 2^n unique classifications (the exact number depends on the independence of the predicates).

Acknowledgements

The authors would like to thank the referees of the Journal in Computer Virology, and the participants of the 2nd International Workshop on the Theory of Computer Viruses (TCV 2007), for their useful questions and comments. The Unix shell script virus in Figure 1 was based on a similar work by Bruce Ediger, published on Gary P. Thompson II's, "The Quine Page" (`http://www.nyx.net/~gthompso/quine.htm`).

A Note on the Inclusion of Virus Code

It was important for the demonstration of our computer virus reproduction models to include excerpts from the source code of some reproducing malware for illustrative purposes, in the vein of Cohen [9] and Filiol [11], 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 construction and classification of affordance-based computer virus reproduction models.

References

[1] 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.

[2] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. Technical Report CSE-TR-530-07, Department of Electrical Engineering and Computer Science, University of Michigan, April 2007.

[3] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. On abstract computer virology: from a recursion-theoretic perspective. *Journal in computer virology*, 1(3–4), 2006.

[4] 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.

[5] Ero Carrera and Gergely Erdélyi. Digital genome mapping — advanced binary malware analysis. In *Virus Bulletin Conference*, September 2004.

[6] 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.

[7] Fred Cohen. Computer viruses — theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[8] Fred Cohen. Computational aspects of computer viruses. *Computers and Security*, 8:325–344, 1989.

[9] Frederick B. Cohen. *It's Alive! The New Breed of Living Computer Programs.* John Wiley & Sons, 1994.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[11] Eric Filiol. *Computer Viruses: from Theory to Applications.* Springer, 2005. ISBN 2287239391.

[12] 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.

[13] Marius Gheorghescu. An automated virus classification system. In *Virus Bulletin Conference*, October 2005.

[14] James J. Gibson. The theory of affordances. *Perceiving, Acting and Knowing: Toward an Ecological Psychology*, pages 67–82, 1977.

[15] James J. Gibson. *The Ecological Approach to Visual Perception.* Houghton Mifflin, Boston, 1979. ISBN 0395270499.

[16] 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.

[17] 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.

[18] 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 2007-10-28).

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[24] Jimmy Kuo and Desiree Beck. The common malware enumeration initiative. *Virus Bulletin*, pages 14–15, September 2005.

[25] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 2007. To appear.

[26] 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.

[27] Daniel Reynaud-Plantey. The Java mobile risk. *Journal in Computer Virology*, 2(2), 2006.

[28] Fridrik Skulason and Vesselin Bontchev. A new virus naming convention. CARO meeting, 1991.

[29] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. Principles of a computer immune system. In *1997 New Security Paradigms Workshop*. ACM Press, 1997.

[30] Eugene H. Spafford. Computer viruses as artificial life. *Journal of Artificial Life*, 1(3):249–265, 1994.

[31] Peter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005. ISBN 0321304543.

[32] Timothy John Taylor. *From Artificial Evolution to Artificial Life*. PhD thesis, University of Edinburgh, 1999. `http://www.tim-taylor.com/papers/thesis/` (accessed 2007-10-28).

[33] Sampo Töyssy and Marko Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2), 2006.

[34] 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.

[35] Matt Webster and Grant Malcolm. Reproducer classification using the theory of affordances: Models and examples. *International Journal of Information Technology and Intelligent Computing*. To appear.

[36] 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.

[37] Matt Webster and Grant Malcolm. Formal affordance-based models of computer virus reproduction — Maude specification, October 2007. `http://www.csc.liv.ac.uk/~matt/pubs/maude/1/`.

[38] 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.

[39] Stephanie Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320, 2007. arXiv:cs/0504045v1 [cs.CR].

[40] Christos Xenakis. Malicious actions against the GPRS technology. *Journal in Computer Virology*, 2(2), 2006.