Algebraic Specification of Computer Viruses and Their Environments

Matt Webster*
M.P.Webster@csc.liv.ac.uk

Department of Computer Science, University of Liverpool, Liverpool, L69 7ZF, UK

Abstract. This paper introduces an approach to formal specification of computer viruses and their environments through the development of algebraic specifications using Gurevich's Abstract State Machines (ASMs) and Goguen's OBJ. Distributed ASMs are used to develop a model of an abstract computer virus, which through a process of refinement is converted into models of different virus types. Further refinement has resulted in an executable specification, written in the Abstract State Machine Language, AsmL. The models strengthen the thesis that any algorithm can be modelled at a natural abstraction level using an Abstract State Machine. The ASM models combined with the AsmL executable specification provide a complementary theoretical and experimental framework for proofs and experiments on computer viruses and their environments, as well as a useful means of classification of different computer viruses types. Next we look at a different approach to computer virus analysis using OBJ, which is used to specify computer viruses written in an ad hoc programming language, SPL. The OBJ specification is shown to be useful for the detection of a virus type that is particularly difficult to detect – the metamorphic computer virus (MCV). Finally, the usefulness of the two specifications for reasoning about computer viruses is discussed and in this regard the two formalisms are compared.

1 Introduction

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

Academic study of computer viruses began in 1987, when Cohen defined an abstract computer virus and gave a proof of the undecidability of computer virus detection, proving that there would be no detection-based panacea for the computer virus problem [1]. Preventing computer viruses through the use of severely

* With thanks to my supervisors Grant Malcolm and Alexei Lisitsa, for showing me the power and beauty of algebraic specification.

restricted operating systems is possible, since the computer virus can only work by modifying stored programs in memory so that they contain a copy of the virus. However, the computer architecture that allows program creation and modification is at the core of the flexibility and efficiency of modern computers; for example, no compiler could run without the creation and modification of stored programs. Thus, finding a cure for computer viruses by restricting their environments is impractical, and reliable detection is impossible. Therefore the fight to stop illicit computer viruses becomes a question of optimization, i.e., "How can we best protect ourselves against computer viruses?" In addition, tractability is a primary concern due to the trade-off between efficiency and thoroughness of anti-virus (AV) scanners. With this and the sustained proliferation of computer viruses in mind, general theories of computer viruses and their environments would be useful. Formal specification of the behaviour of different virus types can provide insight to developers of anti-virus software by highlighting the commonality between different computer viruses, e.g. by encouraging the reuse of detection and disinfection methods. Knowledge of which specific details of the implementation of computer systems afford survival for viruses can be derived from formal models, and can be used to restrict computer virus behaviour through the development of systems that are inherently (and provably) more secure.

In this paper, two separate approaches to computer virus formalisation are presented. In §2 Abstract State Machines [2] are used to build a formal model of an abstract computer virus, which is refined to several more specific virus types. An executable specification in AsmL (based on the ASM models) is given, forming with the ASM models a complementary theoretical and experimental test-bed for formal proofs on computer viruses and their environments, as well as a formal means of computer virus categorisation. OBJ [3] is used in §3 to build a formal framework for computer virology based on order-sorted equational rewriting. An ad hoc simple programming language (SPL) is used to implement computer viruses and executable files, and a formal language representation of an operating system is created. In §3.3 a promising new means of detecting metamorphic computer viruses based on the OBJ specification is demonstrated. In §4 it is shown how these formal algebraic specifications are readily applicable to computer virus detection technology through the application of the developments made in MCV detection and computer virus classification.

1.1 Metamorphic Computer Viruses

The undecidability of computer virus detection is one of the oldest results in the field of computer virology, but anti-virus scanners have traditionally compensated for this by exploiting a weakness common to many naïve computer viruses: constant syntax. The string of binary digits corresponding to a particular computer virus usually remains unchanged from one generation to the next, as does the placement of the computer virus within the infected executable file, e.g. one particular virus might be placed at the start of any executable it infects. AV scanners would search executable files for virus "signatures" – strings of bits

that correspond to a particular virus – at the usual sites of infection within executables files. The presence of a signature would signal that the executable was infected, and that further steps needed to be taken to disinfect the file.

In order to avoid detection, the writers of computer viruses began to develop ways of obfuscating the suspect virus code. One attempt at this, the polymorphic computer virus, which changes its syntactic (binary digit) representation using encryption, fails to remain hidden from AV scanners once its means of decryption has been discovered. Once decrypted, all generations of polymorphic computer viruses look alike, and the signature-based approach to detection can be used. A more powerful means of detection avoidance is employed by the metamorphic computer virus. Each successive generation of a metamorphic computer virus modifies the syntax, but leaves the semantics unchanged. In this way the behaviour of each successive generation is the same, but the virus appears to be different. Thus, it becomes much more difficult for an AV scanner to detect a metamorphic computer virus using a signature-based approach, and it is not possible to keep a record of all possible generations because there are an unlimited number[1]. In this way the metamorphic computer can successfully avoid detection [4].

This problem is not confined to a particular programming language. Within any Turing-complete programming language there is some redundancy, that is, the mapping from syntax to semantics is many-to-one[2]. This redundancy allows the metamorphic computer virus to avoid detection (by signature-based methods) through automated syntactic variant generation at run-time.

1.2 Related Work

A good introduction to theoretical computer virology and the formalisation of computer viruses can be found in early work by Cohen [1, 5] and Adleman [6]. Further to Cohen's work on the undecidability of computer virus detection, Spinellis has recently demonstrated [7] that reliable identification of a known virus is NP-complete, and Thimbleby et al [8] have developed a formalism to describe trojans and computer virus infection. Additionally, Lakhotia & Mohammed [9] have studied an approach based on imposing order on high-level language statements in order to reduce the number of syntactic variants and therefore aid MCV detection.

[1] Program size is (theoretically) unlimited, so an infinite number of variants on a single metamorphic computer virus is possible through the introduction of "junk code" that does not alter the state of the machine, but nevertheless has a syntactical representation. A good example would be the `NOP` instruction in 68000 assembly language, which does not alter the state at all upon its execution.

[2] Suppose that there is a Turing-complete programming language for which there is a one-to-one mapping from syntax to semantics. If such a language existed, it would be possible to prove the equivalence of any two programs p_1 and p_2 by translation into this language. Since program equivalence is an undecidable problem, there can be no such language. (This proof assumes that if such a language existed then translation into the language would be possible.)

2 The ASM Approach

Abstract State Machines (formerly known as Evolving Algebras) are a formal approach to algebraic specification of software systems, where each step of an algorithm is treated as a bounded number of changes to an algebra, and the whole algorithm can be described using a finite number of simple transition rules. The ASM thesis states that any algorithm can be modelled at a natural abstraction level using an ASM [2].

Algebraically, an Abstract State Machine is defined as follows. Let A be an Abstract State Machine. The vocabulary (or signature) Υ of A is a finite set of functions, each of a fixed arity. The universes of A are the sets on which the functions of A act, e.g. an ASM that adds two integers will require the set of all integers as one of its universes. The state of A consists of a non-empty set X (the superuniverse of A), together with interpretations of every function name in the signature. The superuniverse of A does not change, but the interpretations of the function names may. The superuniverse contains distinct elements $true$, $false$, $undef$ that let us have Boolean tests, and partial functions (e.g. $f(a) = undef$).

Whilst the structure above is used to define the functions used and the sort sets of an ASM specification, they are glued together to form a complete specification using *rules* [2]. The update rule (variable updates), the conditional rule (conditional statement execution) and the block rule (parallel statement execution) are proven to be sufficient to specify all sequential algorithms [10].

2.1 Distributed ASMs

Distributed ASMs (DASMs) are an extension to the ASM formalism for specification of multi-agent computations [2]. In this paper DASMs are used to model the behaviour of the computer virus' environment – the operating system. A set $Modules$ represents the programs stored in the file store ready to be run. A set $Agents$ represents processes in memory, which run a particular module (program). Agents run in parallel with one another. The user is modelled using a (non-deterministic) *external function* that runs programs randomly by assigning modules to new agents. This could be implemented either by actual user input, or more likely, by a function that non-deterministically selects executable files to be run. When a particular agent has reached fixpoint, that is, when it has ceased to update the store, it is considered to have finished its execution run and terminated. The operating system is simply the distributed ASM itself. The agents run modules, and run concurrently with one another.

2.2 An Abstract Computer Virus

An ASM for the abstract computer virus, which is designed to encompass the entire class of computer viruses, is as follows.

Superuniverse, $\boldsymbol{S_{AV}} = Agents \cup Modules \cup Rules \cup Boolean \cup \{undef\}$, where $Agents$ is the set of agents that are run by the distributed ASM; $Modules$ is the set of modules; and $Rules$ is the set of well-formed rules for ASMs.

Vocabulary, Υ_{AV}

The functions in the vocabulary are defined as follows.

- Infected: $Modules \rightarrow Boolean$. Returns true iff a module has been infected with the viral rule. Returns false otherwise.
- Infect: $Modules \times Rules \rightarrow \mathcal{P}(Modules)$. Infection by a computer virus is a modification to the set $Modules$ in the specification, so that the set of ASM rules corresponding to the virus m_v is executed by (at least) one more module than was previously the case. m_v is returned by ThisProgram, and added to some $m \in Modules$ by Infect.
- ThisProgram: $\rightarrow Rules$. Returns the viral rule. This models a "real-life" function that can analytically (or otherwise) generate the viral code in the module, and return it. A possible implementation of this would be to delimit the viral code within an infected program using some sequence of bits (e.g. from an assembly language perspective, a number of `NOP`s strung together), which the virus can later use to derive its own code when it comes to copying it for infection.
- State: $\rightarrow$ {Done} $\cup$ $\{undef\}$. Used to enforce sequentiality; initial value is $undef$.

Rules The set of rules corresponding to the abstract virus is as follows. The ASM block rule specifies that any given block of statements will execute in parallel, therefore any list of rules will execute in parallel unless sequentiality is enforced using guards. Since the following rule should execute only once per execution of the module in which it resides, we ensure this through the use of the nullary function State, which is set to Done once a single infection has taken place.

```
AbstractVirus = if not( State = Done ) then
                  choose m in Modules satisfying not( Infected(m) )
                    Modules := Infect( m, ThisProgram )
                    Infected(m) := true
                    State := Done
                  endchoose
                endif
```

The viral rule above copies itself to other modules in the distributed ASM. When the modified (infected) modules are run, the viral module finds even more modules to infect.

Specification of computer viruses is a new application of ASMs, and as such has strengthened the ASM thesis that any algorithm can be modelled at a natural level of abstraction by an Abstract State Machine [2].

Through a process of refinement it is straightforward to specify more specific virus types based on, and conforming to, the description of the abstract virus.

This method has been used to specify a further four virus types: the parasitic virus, the companion virus, the overwriting virus and the backup virus (a.k.a. "the file worm"). We shall look in detail at the refinement to the parasitic virus.

2.3 The Parasitic Virus

The first refinement of the abstract virus produced the parasitic virus, perhaps the most common virus type. The parasitic virus replicates by copying itself from one executable file from another, e.g. by prepending itself to the executable code corresponding to a particular program, so that when that program is next run by the user, the parasitic virus code is run momentarily before the rest of the executable. This allows the parasitic virus another chance to replicate.

The process of refinement was a modification of only one of the functions in the vocabulary, and one of the rules. All else remains unchanged. The function Infect was replaced by Infect′, a slightly modified function performing the following update:

- Infect′(m, r) =
  Modules **:=** $(Modules - m) \cup$ addRuleToModule(m, r)
  where addRuleToModule(m, r) = RulesOf$(m) \cup \{r\}$

In other words, Infect′(m,r) removes module m from the set of Modules, and adds a new module which is the result of adding rule r (the parasitic virus rule) to a new module consisting of the rules that were in m. Infect′ is defined using the ASM sub-machine above. (RulesOf returns the set of rules corresponding to the module m. The behaviour of addRuleToModule is defined by the equation above.)

The rule "$Modules$ **:=** Infect(m, thisProgram)" is replaced by "m **:=** Infect(m, thisProgram)" indicating that only one module is updated by the infection process, which is consistent with the typical parasitic virus [1].

The refinements from the abstract virus to the other virus types follow a similar process.

2.4 The AsmL Executable Implementation

In order to test experimentally the accuracy and applicability of the abstract descriptions of the various virus types, an executable implementation of the viruses was created using AsmL, the Abstract State Machine Language developed by Microsoft Research [11]. AsmL is an object-oriented language for developing executable ASM specifications, and so the most natural way of implementing the executable computer virus test-bed was to create an object-oriented model of the computer virus environment, consisting of a store class, an operating system class encapsulating a store instance and OS routines (methods), and a user simulation class, encapsulating an OS instance and methods for running executables in the store using the OS routines. The refined computer viruses and executable files were also modelled using classes, all of which implemented a class interface

to allow them all to be executed in the same manner, regardless of their differences in behaviour, i.e. all objects `x` implementing the `File` interface could be executed using `x.Program()`. In this way, objects corresponding to files in the store could be granted executable status. Parasitic computer viruses could be implemented by letting the file allocation table (a mapping from file handles to programs) correspond to a sequence of programs. (This is analogous to the case with a modern OS where an executable file can lie over multiple segmented blocks on a hard drive.) Thus, parasitic virus infection takes place when the viral code copies itself to a new executable file, whose file handle is then added to the list of files to be run when a given file name is run by the simulated user [12].

The AsmL specifications of the particular virus types were created using a refinement process from their ASM versions in order to ensure the accuracy of their behaviour. Execution of the AsmL virus specifications proved empirically that the specifications were accurate, as the behaviour exhibited was that expected of the respective virus types.

As well as strengthening the ASM thesis, the specification of computer viruses using ASMs and AsmL doubles as a computer viruses test-bed and framework for proofs on computer viruses, creating a formal specification that can be analysed theoretically through proofs on the ASMs, or empirically by execution of the AsmL implementation.

3 The OBJ Approach

After modelling computer viruses using Abstract State Machines, another means of algebraic program specification was used in order to gain a different perspective on the problem, as well as to explore and compare the different proof methods afforded by the various approaches to algebraic program semantics.

3.1 An OBJ Specification for Computer Viruses

OBJ is a formal notation for the algebraic specification of software systems [3]. Like ASMs, it is algebraic, but the approach to semantics is quite different. In OBJ, data types are defined as sorts in an order-sorted algebra. Ground terms can be transformed into more complex terms using equational rewriting, which in turn can be used to prove theorems expressed using the terms through *reduction* – a series of one or more equational rewrites.

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

```
obj PEANO is
      sort Nat .

      op  0 : -> Nat .
      op  s_ : Nat -> Nat .
      op  _+_ : Nat Nat -> Nat .
endo
```

This construct is called a *module*. `sort Nat` specifies that we are dealing with only one sort of data in the superuniverse of the algebra used in this specification. There are three operations defined. The first, `0` is a nullary operation (constant) and can be used by other non-nullary operations (of the correct ranks) to generate more complex terms. `s_` is a unary operation that takes a `Nat` and returns a `Nat`, and `_+_` is an infix binary operation that takes two `Nat`s and returns a `Nat`. `s_` will be used for the successor function, and `_+_` (quite naturally) will be addition.

In OBJ the semantics of a programming language is given using a series of "equations", which are actually string rewriting rules. So, if we wanted to give the semantics for the `_+_` operator above, we could do this as follows:

```
obj PEANO-EQUATIONS is
   pr PEANO . *** Import the PEANO module

   vars M N : Nat .
   eq  M + 0  =  M .
   eq  M + s(N)  =  s(M + N) .
endo
```

Now we have specified the semantics of addition, or rather, we have made the `_+_` operator behave as a function which returns the value of the sum of two natural number operands in Peano notation. As mentioned before, the `=` is actually a rewriting operator, showing that the term on the left is rewritten to the term on the right. Using OBJ, we can perform a reduction in order to reduce a term to the most reduced form possible, i.e. the OBJ parser keeps applying rewrite rules as long as there is a rewrite rule that will apply. A typical reduction would look like:

```
OBJ> reduce s(s(s(0))) + s(s(0)) .
  ...
rewrites: 3
result Nat: s (s (s (s (s 0))))
```

An important notion in OBJ is that of reduction as proof. Since each of the equational rewrites holds for the natural numbers, the above reduction can be seen as a proof that "$3 + 2 = 5$". Whilst trivial, this example demonstrates the expressive power of the OBJ formalism.

3.2 Computer Viruses in OBJ

Algebraic specification of imperative programs is one of the principal uses of OBJ [13]. Semantics can be easily given to functions and statements using equational rewriting, e.g.:

```
eq S ; X := VAL [[X]] = VAL .
cq S ; X := VAL [[Y]] = S[[Y]] if X =/= Y .
```

The first rewrite rule says that after the statement `X := VAL` has been evaluated, the value of `X` in store `S` is equal to the value `VAL`. The second rule says that the value of `Y` (such that `X` $\neq$ `Y`) in store `S` is unchanged by the evaluation of the same statement, which we would expect to be the case. This is the traditional approach to the semantics of stores for programming languages defined in OBJ.

Soon after considering the possibility of expressing the semantics of a computer virus in OBJ, it became apparent that the method of giving semantics to the store as outlined above would be less intuitive, and therefore a hindrance to the overall specification and computer virus test-bed. The reason for this is that computer viruses are meta-programs (i.e. programs whose input or output are programs), and therefore need to be able to read the code of other programs in the file system, including themselves. This meant that the file system (which can be seen as an extended store that can contain programs) would have to be encoded explicitly in the string being rewritten during a reduction. For this reason it was decided to develop a different method of giving semantics to programs in order to deal with meta-programs in the most intuitive way – by modelling an operating system capable of editing and maintaining a file store and executing executable files therein.

When specifying the behaviour of an OS, there are a number of factors to consider:

- There needs to be a representation of a file system, which can be modified and from which the contents of files can be retrieved.
- There needs to be a representation of a user interface, into which a list of executable files to be run (for example) can be inputted. This need not be interactive or elaborate by any means; for example, the list of files to be run could simply be a static value hard-coded into the specification.
- The effects of the execution of a program on the file system need to be preserved after the program has finished executing.
- Common operating system functionality must be supported, e.g. file handling routines to delete a file; a method of loading and executing instructions stored in the file store as executable programs.
- Many operating systems employ a notion of environmental variables – special variables whose values are kept by the OS and are accessible to programs executed by the OS. For example, a useful environmental variable might be the file handle of the current file running, so that any program can obtain its own filename.

With the above considerations the most logical way of tackling the problem was to encode the state of the whole operating system, including file system, memory state and environmental variables, as a single string. This string is necessarily quite long ($\sim$ 80 tokens for an OS with a file system containing two short executable files). The disadvantage of such an explicit approach to operating system semantics is the large number of rewrites needed to calculate the effects of running programs when compared with the traditional OBJ approach discussed earlier. For example, a single execution of a simple computer virus requires over

1500 rewrites to evaluate. (Whilst this may sound excessive, the reduction takes place in a fraction of a second on a modern personal computer.) This extra overhead is compensated by the intuitiveness of the implementation, which reduces development time, decreases the likelihood of conceptual programming errors, and increases the ease of interpretation of the specification and results of reductions.

(As is the case in standard OBJ semantics, the meaning of the statements stored within the executable files must be defined. For this purpose a Simple Programming Language (SPL) was created and its semantics defined using OBJ. SPL is a simple Turing-complete programming language with basic features like variables, loops and conditionals.)

Here is a typical example of a string representing the state of the OS:

```
lfha ~ s(0) /\ cfr ~ nofile /\ ! /\/\ 0 :::: varc := 0 ; eof
||| s(0) :::: counter := 0 ; counter := s(counter) ; eof ||| !
```

This unwieldy-looking string is not as complex as it might seem at first – it is simply a concatenation of two lists. Firstly there is the environmental variable list: "`lfha ~ s(0) /\ cfr ~ nofile /\ !`". Here there are two environmental variables, `lfha` (Last File Handle Assigned) and `cfr` (Current File Running), whose values are `s(0)` and `nofile` respectively. The Peano notation for natural numbers is used for file names, so the last file handle assigned has the value 1. The `cfr` variable is set to `nofile` which (somewhat paradoxically) means that no file is currently running. The environmental variable list elements are delimited by the `/\` operator, and the end of the list is denoted by `!`, which represents the empty environmental variable list.

The next operator is `/\/\` which separates the environmental variable list from the second of the two lists: the file list. The file list delimiter is the `|||` operator, so we can see that we have two files in this particular file system. The file format in the OBJ specification is as follows:

```
op _::::_ : FileHandle StatementList -> File .
```

So, by looking at the file list we can tell that the file with handle `0` has the statement list "`varc := 0 ; eof`" within.

This statement list is the program corresponding to the file with file handle `0`. In this case, the program consists of a single executable statement which assigns the value `0` to variable `varc`. The statement delimiter is `;`, and the `eof` operator that follows simply denotes the end of file (i.e. the statement list has come to an end). After the `eof` comes the file delimiter `|||` and then the listing for the file whose handle is `s(0)`. Finally, the file list ends with the empty file list operator `!`.

With this infrastructure it is possible to define the semantics of SPL, and by extension, meta-programs such as computer viruses. To compare, here is the equivalent assignment operation to the one shown in §3.2.

```
eq execS V := VAL in EVL /\/\ FL = EVL[V isnow VAL] /\/\ FL .
```

`execS` is a operation that executes a statement (in this case `V gets VAL`) relative to an environmental variable list (`EVL`) and a file list (`FL`). The equation above states that after `V := VAL` has executed, the value of `V` is updated to that of `VAL` in the environmental variable list by using the `_[_isnow_]` operator to request a environmental variable list update. The result of the rewrite will then match to the left hand side of another rewrite rule that will effect the update. (Note: during development of the specification the environmental variable list came to contain the values of program variables also.)

Every SPL statement is defined in a similar way in the OBJ specification, and a set of application programming interface (API) functions (e.g. OS routines to delete a file or update an environmental variable) are defined also. Thus, in SPL a simple computer virus might look like:

```
1      s(0) ::::  fh1 := getFileHandle ;
2                 myName := getSelfName ;
3                 nfh := newFileHandle ;
4                 counter := 0 ;
5                 lastLine := label end ;
6                 do { line := getLine(myName,counter) ;
7                      writeToFile(line,nfh) ;
8                      counter := s(counter) ;
9                      eof }
10                while ( not(line == lastLine) ) ;
11                prepend(nfh,fh1) ;
12                deleteFile(nfh) ;
13                label end ;
14                eof
```

In line 1 the variable `fh1` is assigned the returning value of the function `getFileHandle`, which returns an arbitrary file handle from the file list. In line 2 the function `getSelfName` is called to in order to return the file handle of the current file running (i.e. the file handle of "self") and it is assigned to variable `myName`. In line 3 the variable `nfh` is set to the value of a new file handle, that is, a file handle that is not currently being used by any other file. This is so that a temporary file can be created and written to during the infection process. In lines 4-5 two variables that are needed for the forthcoming `do {_} while (_)` loop are initialised.

In lines 6-10 we encounter the loop. In the first iteration of the loop, the `0`th statement (i.e. the first line) of the file named `myName` (which is the file currently running and therefore the file containing the virus) is read in by the function `getLine(_,_)` and assigned to the variable `line`. Next, this statement is written to the temporary file whose handle is `nfh` using the `writeToFile(_,_)` function. In line 8 the variable `counter` is incremented, so that on the next iteration of the loop the following line will be read in and written to the file `nfh`, and so on.

The net effect of this loop is that a copy of the virus is placed in a temporary file (`nfh`). The loop stops when the statement that has just been copied (`line`)

is equal to the value of variable `lastLine`, which is set to `label end`. `label end` is the last executable line of the virus program (line 13) and separates the virus from the rest of the infected executable. Clearly, in this case the virus exists in a file alone, but the purpose of the guard is to make sure that in future generations only the virus is copied and not the rest of the host executable.

Next comes a call to the function `prepend(nfh,fh1)`, which causes the statements corresponding to `nfh` to be added to the start of the file `fh1`, in the order they appeared in `nfh`. This is the most crucial stage of viral infection, where the virus attaches itself to the host. In line 12 the temporary file `nfh` is deleted from the file system. Line 13 is the label mentioned in previous paragraph, and the final line (14) denotes the end of the file.

The overall effect of running the above virus program is that the virus searches for another executable file in the file system, which it infects by prepending its own code to that of the executable.

3.3 An OBJ-based Approach to Detecting MCVs

As mentioned in §1.1, metamorphic computer viruses (MCVs) are able to conceal themselves from detection by anti-virus scanners by changing their syntactic form whilst their semantic form (i.e. behaviour) remains unchanged. Using the OBJ specification above it becomes possible to start to reason about MCVs and means of detecting them. It was found whilst specifying SPL using OBJ that equivalent statement lists would be quite straightforward to prove using OBJ. For instance, a metamorphic computer virus in SPL could use the redundancy inherent in `do {_} while (_)` and `while (_) do {_}` to generate a syntactic variant of itself and therefore aid concealment. This equivalence can be specified in OBJ:

```
eq execS do {SL1} while (T) in EVL /\/\ FL =
 execSL SL1 ;; while (T) do {SL1} ; eof in EVL /\/\ FL .
```

This rewrite rule would reduce all occurrences of `do {_} while (_)` to that of `while (_) do {_}` , so any MCV using that method of syntactic variant generation alone could be easily detected.

Another example of metamorphism in computer viruses uses the `NOP` assembly language instruction to generate syntactic variants. For example, a computer virus during the process of replication could place `NOP` instructions (which do not affect the state of the machine) at pseudo-random locations within its code, and in so doing generate syntactic variants. The following OBJ rewrite rule would remove the `NOP`s inserted by such a metamorphism function:

```
eq execS NOP in EVL /\/\ FL = EVL /\/\ FL .
```

This rewrite rule effectively states that any program containing a `NOP` instruction is semantically equivalent to the same program without that instruction.

It is possible to specify many other such syntactic equivalents using OBJ rewriting logic, and also prove that they hold in the given programming language.

In doing so, a reduced form of the programming language SPL (perhaps we could call it SPL′) would be created. Any MCV instance translated from SPL to SPL′ using OBJ rewriting rules would then be much more easily detected than before, as the number of syntactic variants in SPL′ is greatly reduced.

4 Conclusion

Both Abstract State Machines and OBJ offer a precise semantics for algebraic program specification, validation and verification, but the particulars of the individual notations foster different approaches to the specification of computer viruses and their environments.

It was found that Abstract State Machine specifications can be first developed at a high abstraction level in order to lay the foundations for the more complete specification, e.g. the example of the abstract virus. It then becomes possible using refinement steps to move towards a more applicable specification in keeping with the original abstract model. This approach emphasises the relationships of computer viruses to one another and with their environments, and could easily be used as a means of categorising computer viruses. The ASM models, combined with the AsmL implementation forms a duet of theoretical and experimental frameworks for proofs related to computer viruses. For example it is possible to prove that a particular operating system feature would affect computer virus behaviour in a certain way. This manner of formalisation would prove useful for computer virologists who might want to develop a less virus-prone operating system, or to prove that a particular update to an anti-virus scanner would indeed detect, for example, all instances of a certain virus. This is even more straightforward due to AsmL's built-in conformance testing tools.

It is beneficial to categorise computer viruses in an algebraic manner using ASMs, as it places all viruses on a "family tree" of computer viruses, which itself could perhaps be a part of an even larger family tree of self-replicating programs. (Other types of self-replicating programs include cellular automata, network worms, artificial life forms, etc..) It is natural to think of computer viruses in terms of their differences in size, commonness, replication mechanism, side-effects, etc., but the benefit of ASM specification has been to show formally how computer viruses relate to one another. Anti-virus scanners are forced to take computer viruses as dissimilar entities as they scan for executable file infection, but perhaps an abstract view of the computer virus problem could assist the development of more secure systems that are intrinsically less vulnerable to the computer virus threat. Study of the mechanisms by which all computer viruses are able to survive would enable us to see how we might develop such systems, and a formal, algebraic approach is consistent with this aim.

In addition to the benefits to computer virology, the ASM specification of computer viruses and their environments has strengthened the ASM thesis that any algorithm can be modelled at a natural abstraction level by an Abstract State Machine, as computer viruses and the broader class of self-replicating programs have not been modelled before using the ASM formalism.

OBJ has great applicability in the domain of programming language specification, and naturally this led to the specification of a formal language which was used to describe the state of the operating system and programs running thereon. This is perhaps the most natural approach to computer virus specification given the structure of OBJ modules in allowing powerful customisation of syntax and semantics. This particular approach may also be of assistance to future research involving the specification of operating systems.

Examples of OBJ program semantics in the literature are concerned with the specification of a single program relative to a store. This work has shown that this approach is extendable to the operating system. A file store was modelled using a single string in order to ensure that effects on the file system of the execution of a program persisted beyond the end of the execution of that program. At first this approach seemed somewhat inefficient compared with traditional approaches, but compensated for its apparent lack of grace through a natural approach to operating system semantics, thus lending itself well to proofs. Also, the problem of modelling meta-programs was solved by simulating OS API routines using the Simple Programming Language used in the virus specifications.

A new way of exploring the detection of metamorphic computer viruses has been developed using the OBJ specification and further research based on this is pending. The ongoing aim of this study is to identify and isolate a set of rewrite rules that would reduce the number of syntactic variants per semantic description to the greatest extent, and in doing so provide a reliable means of MCV detection. The advantage of exploration into such rewriting rules using OBJ is that the formality of the notation ensures the equivalence of programs translated into "reduced" forms and the accuracy of MCV detection once translation has taken place.

The use of SPL in this work has been ad hoc; the system produced is a proof of concept for the technique of formal syntactic variance reduction by translation. An additional use of the OBJ specification would be in the equational rewriting-based removal of junk code, which is non-effective code added to successive generations by MCVs in order to create syntactic variants and avoid detection. It is expected that these approaches would be best applied alongside other anti-virus strategies, including heuristic-based scanning, disassembly and procedure abstraction, as well as the new developments of control flow graph generation and neural network-based approaches to metamorphic computer virus detection [14, 9, 15]. The next step will be to take the technique and apply it to a form of assembly language, e.g. the Intel® IA-32 instruction set. The semantics of IA-32 will be specified using OBJ, and once done examples of computer viruses using that architecture will be experimented upon, with the intention of the discovery of rewriting rule sets that are facilitative of the detection of MCVs outside the laboratory.

References

1. Cohen, F.: Computer viruses – theory and experiments. Computers and Security **6**(1) (1987) 22–35

2. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
3. Goguen, J.A., Walker, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In Goguen, J.A., Malcolm, G., eds.: Software Engineering with OBJ: Algebraic Specification in Action. Kluwer Academic Publishers (2000) ISBN 0792377575.
4. Ször, P., Ferrie, P.: Hunting for metamorphic. In: Virus Bulletin Conference Proceedings. (2001)
5. Cohen, F.: Computational aspects of computer viruses. Computers and Security **8** (1989) 325–344
6. Adleman, L.M.: An abstract theory of computer viruses. In: Advances in Cryptology - CRYPTO '88. Volume 403 of Lecture Notes in Computer Science. (1990) 354–374
7. Spinellis, D.: Reliable identification of bounded-length viruses is NP-complete. IEEE Transactions on Information Theory **49**(1) (2003) 280–284
8. Thimbleby, H., Anderson, S., Cairns, P.: A framework for modelling trojans and computer virus infection. The Computer Journal **41**(7) (1998)
9. Lakhotia, A., Mohammed, M.: Imposing order on program statements to assist anti-virus scanners. In: Proceedings of Eleventh Working Conference on Reverse Engineering, IEEE Computer Society Press (2004)
10. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic **1**(1) (2000) 77–111
11. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Technical report, Microsoft Research (2004) MSR-TR-2004-27.
12. Webster, M.: ASM-based modelling of self-replicating programs. Technical Report ULCS-05-005, Department of Computer Science, University of Liverpool, UK (2005) Presented at the 11th International Workshop on Abstract State Machines (ASM 2004).
13. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. Massachusetts Institute of Technology (1996) ISBN 026207172X.
14. Lakhotia, A., Singh, P.K.: Challenges in getting 'formal' with viruses. Virus Bulletin, September 2003 (2003) 15–19
15. Yoo, I.: Visualizing windows executable viruses using self-organizing maps. In: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security. (2004)