ASM-based Modelling of Self-Replicating Programs

Computer Science Technical Report
ULCS-05-005
Presented at the 11th International Workshop on Abstract State Machines (ASM 2004)

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

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

Abstract. Self-replicating programs are a class of algorithms it seems has not yet been modelled using the Abstract State Machine (ASM) formalism. In this paper an attempt at modelling the sub-class of computer viruses is shown. Implicit in modelling a computer virus is modelling the environment it needs to survive, and this also is detailed. Finally, an account of the experience of implementing these models in AsmL, the Abstract State Machine Language from Microsoft Research, is given.

1 Introduction

The class of self-replicating programs includes viruses and worms, and was described in detail in [1]. Viruses are the focus for this project, and were defined formally by Cohen [2]:

> We define a computer 'virus' as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows.

2 Modelling an Operating System

An operating system has been modelled using a distributed ASM. Distributed ASMs are described in the Lipari guide [3]. In the model, modules represent the programs stored in the filestore ready to be run. Agents represent 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

* I thank my Honours project supervisor, Alexei Lisitsa, for suggesting I submit this paper to ASM 2004.

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. We shall see that ASM rules are versatile enough to simulate system calls, e.g. reading and writing files, which are vital for the viral infection process.

3 A Viral Module

We use the following universes:

- $Agents$ - the set of agents that are run by the distributed ASM.
- $Modules$ - the set of modules. Each agent runs a module. Each module is a set of rules (a single-agent program), and is similar to a non-distributed ASM.
- $Rules$ - the set of well-formed rules for ASMs.

We use the following vocabulary:

- $Infected : Modules \rightarrow Boolean$. Returns true if a module has been infected with the viral rule.
- $addRuleToModule : Modules \times Rules \rightarrow Rules$, where $Rules$ is the set of well-formed rules. Since a module is just an ASM program, and an ASM program is just a rule composed of other rules, we can assign the rule returned by this function to a module name to model the addition of a rule to a module. We could define this function set theoretically as: $addRuleToModule(m, r) = RulesOf(m) \cup \{r\}$, where $RulesOf(m)$ refers to the set of rules corresponding to the module m.
- $thisProgram :\rightarrow Rules$. Returns the viral rule. This models a "real-life" function that can analytically (or otherwise) identify 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`[1]s strung together), which the virus can later use to derive its own code when it comes to copying it for infection.

The viral module looks like this:

Program_{Viral} = choose m in $Modules$ satisfying not(Infected(m))
    m := addRuleToModule(m, thisProgram)
    Infected(m) := true
endchoose

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

[1] `NOP` is a 68000 assembly language mnemonic, and stands for "No operation".

4 Implementation using AsmL

4.1 Modelling the Virus's Environment

The object-oriented features of AsmL [4] were most useful during implementation of the model of the computer virus and its environment. The environment was based on the von Neumann computer architecture, and consisted of three classes, `Storage`, `OS` and `User`. It was the intention that the user must only interact with the store via the operating system (OS), although an executable program (e.g. a virus) could interact with the store directly.
`Storage` encapsulates the file store and some low-level operating system methods, e.g. `Add(...)` and `Remove(...)` for adding and removing files from the filespace. `OS` encapsulates user-level operating system methods. The `User` class models the user, and contains methods for populating the file space with executables, installing the first instance of the virus, and running random programs. The latter method models the external function mentioned earlier, and chooses filenames (which are modelled as integers and stored in a set instance variable within the `Storage` class) non-deterministically, then uses a method from the `OS` class to run them:

```
choose f in filenames
 os1.Run( f )
```

4.2 Modelling Executable Files

One of the biggest challenges in designing a simulated file system was to be able to have an AsmL set of the various types of executable file (e.g. "Hello world" executable, virus etc.). Ideally, the virtual store consists of such a set, in this case an instance variable within the `Storage` class. It would be possible to implement such a set by enumerating the desired types during set declaration, e.g.

```
var s as Set of Executable or Virus
```

However this leads to later incompatibility if a new virus class needs to be added to the simulation. (All sets and methods declarations using the disjunctive type must be updated.) The problem was eventually solved by creating an interface to give classes executable status:

```
interface File
 public abstract Program()
```

Any class that implements `File` must provide a definition for the abstract method, `Program()`. In this way, a `Set of File` can be constructed, and the `Program()` method can be called on any object in that set, regardless of the object class. Taken within the context of the simulation, the File interface is analogous to the flag an operating system sets for a given file in order to give it executable status. Indeed, within the simulation any file wanting to become executable must implement the File interface.

4.3 Modelling Viral Behaviour

The Storage class has an instance variable, `FAT`, that models the file allocation table of the simulated store. `FAT` is a set of `Integer` × `Seq of File` tuples, and associates filenames (integers) with sequences of objects that implement the File interface (`Seq of File`). When a user selects a filename to be run by the operating system, the operating system retrieves the sequence of File objects corresponding to the filename given using the `FAT` instance variable. Then, the operating system (through a low-level method in the Storage class) executes each File object in the sequence in turn, by invoking its `Program()` method. The code looks something like this:

```
foreach i in seqToRun
 i.Program()
```

When the viral `Program()` method is executed, it searches for an uninfected file. If one is found, a new instance of the virus class is created, and the constructor modifies the file allocation table (FAT instance variable in the Storage class) so that the sequence of File objects corresponding to the filename being infected now includes a virus object (the virus object refers to itself using the AsmL keyword `me`). From now on, each time that particular file is executed by the user, the virus object will execute also. The executable has been infected.

References

1. Cohen, F.B.: It's Alive! The New Breed of Living Computer Programs. John Wiley & Sons (1994) ISBN 0471008605.
2. Cohen, F.: Computer viruses – theory and experiments. Computers and Security **6** (1987) 22–35
3. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
4. Microsoft Research: The Abstract State Machine Language. (`http://research.microsoft.com/foundations/AsmL/`) Accessed 18th April 2005.