# Capturing Evolution of Polymorphic Viruses

Eugene Eberbach*

Computer and Information Science Dept., University of Massachusetts,
North Dartmouth, MA 02747, USA, email: eeberbach@umassd.edu

**Abstract.** In this paper three models of polymorphic viruses are presented. These models had to capture self-reproduction, evolution and damaging payload of polymorphic viruses. They have been derived using a formal model of Evolutionary Computation the Evolutionary Turing Machine, cellular space models, and the $-calculus process algebra for problem solving. Some preliminary results associated with these models are discussed.

## 1 Introduction

A *computer virus* is a program that can reproduce itself by attaching its code to another program (analogous how biological viruses reproduce), and damaging by its payload victim's programs. Worms are like viruses but are self replicating, i.e., they do not need a host program carrier to reproduce and act. Viruses are instances of malicious code sometimes referred to as *malware*.

The term computer virus is derived from and is in some sense analogous to a biological virus. The word *virus* itself is Latin for *poison*. Simplistically, biological viral infections are spread by the virus (a small shell containining genetic material) injecting its contents into a larger organism's cell. The cell then is infected and converted into a biological factory producing replicants of the virus.

Similarly, a computer virus is a segment of machine code (typically 200-4000 bytes) that will copy itself (or a modified version of itself) into one or more larger "host" programs when it is activated. When these infected programs are run, the viral code is executed, and the virus spreads further [21].

There are many types of computer viruses, including companion, excutable program, memory, boot sector, device driver, macro, and source code viruses. Since a computer virus is a program, it can do anything a program can do [22]. Similar to faults in fault-tolerance techniques, viruses can be avoided, detected, diagnosed, contained and recovered from. Antivirus software companies use typically virus scanners to detect computer viruses. In general, the problem of virus detection is undecidable [1]. Virus scanners, by necessity approximate the detecion of viruses from the list of known viruses. Viruses try to hide and users try to find them. Hiding of viruses from detection can be done using compression, encryption, or evolution.

Most of computer viruses are static, i.e., their offsprings are exact copies of the parent. *Polymorphic viruses* on the other hand are viruses that evolve from generation to

*Research partially supported by the grant from Office of Naval Research ONR N00014-06-1-0354

generation. Thus they are very closely related both to evolutionary computation and Artificial Life.

Polymorphic viruses contain a piece of code, called a mutation engine, that can mutate a sequence of machine instructions without changing its functionality, and sophisticated viruses contain mutation engines to mutate the decryptor from copy to copy. The mutation engine itself can be hidden by encrypting it along with the body of the virus. Very little is known about formal models of computer viruses [23, 14]. Even less is known about models of polymorphic viruses. Formal modeling is important because it would allow to understand better the nature of polymorphic viruses and to design a more efficient antivirus software.

In this paper we present three models of polymorphic viruses using Evolutionary Turing Machine, cellular space models, and the $-calculus. The paper is organized as follows: section 2 contains basic definitions how to measure problem-solving performance. In section 3, a formal model of Evolutionary Computation, an Evolutionary Turing Machine (ETM) is presented and polymorphic viruses are expressed as an ETM. In section 4, polymorphic viruses are modeled by automata networks. Section 5, proposes the third model of polymorphic viruses using the $-calculus process algebra for bounded rational agents. Section 6 contains the conclusions and future work.

## 2 Measuring Problem-Solving Performance

An algorithm provides a recipe to solve a given problem. It consists of a finite number of steps/actions, each having well-defined and implementable meaning. An algorithm starts from the initial state and terminates (if successful) in one terminal state from the set of terminal/goal states. If the algorithm reaches its goal state, then we say that the algorithm satisfied its goal and problem has been solved. The goal test (also called a termination condition) determines whether a given state is a goal state. The set of states that is reachable from the initial state forms the search space of the algorithm. Thus the solution of the problem can be interpreted as a search process through the set of states. The state space forms a directed graph (or its special case - a tree) in which nodes are states and the arcs between nodes are actions. Search can be deterministic or nondeterministic/probabilistic. Mulitple solutions can be ranked using an objective function (e.g., a utility or fitness function). In particular, there can be none, one, or several optimal solutions to the problem. Using objective functions allows capturing the process of iterative approximation of solutions and different qualities of solutions in contrast to simply a binary decision: a goal reached or not.

The performance of search algorithms can be evaluated in four ways (see e.g. [19]) capturing whether a solution has been found, its quality and the amount of resources used to find it.

**Definition 1.** *We say that the search algorithm is*

- Complete *if it guarantees reaching a terminal state/solution if there is one.*
- Optimal *if the solution is found with the optimal value of its objective function.*
- Search Optimal *if the solution is found with the minimal amount of resources used (e.g., the time and space complexity).*
- Totally Optimal *if the solution is found both with the optimal value of its objective function and with the minimal amount of resources used.*

**Definition 2.** *Given an objective function $f : A \times X \rightarrow R$, where $A$ is an algorithm space with its input domain $X$ and codomain in the set of real numbers, $R$, problem-solving can be considered as a multiobjective minimization problem to find $a^* \in A_F$ and $x^* \in X_F$, where $A_F \subseteq A$ are terminal states of the algorithm space $A$, and $X_F \subseteq X$ are terminal states of $X$ such that*

$$f(a^*, x^*) = min\{f_1(f_2(a), f_3(x)); a \in A, x \in X\}$$

*where $f_3$ is a problem-specific objective function, $f_2$ is a search algorithm objective function, and $f_1$ is an aggregating function combining $f_2$ and $f_3$.*

Without losing generality, it is sufficient to consider only minimization problems. An objective function $f_3$ can be expanded to multiple objective funtions if problem considered has several objectives. The aggregating function $f_1$ can be arbitrary (e.g., additive, multiplicative, a linear weighted sum). The only requirement is that it captures properly the dependence between several objectives. In particular, if $f_1$ becomes an identity function, we obtain the Pareto optimality

$$f(a^*, x^*) = min\{(f_2(a), f_3(x)); a \in A, x \in X\}$$

Using Pareto optimality is simpler, however we lose an explicit dependence between several objectives (we keep a vector of objectives ignoring any priorities, on the other hand, we do not have problems combining objectives if they are measured in different "units", for example, an energy used and satisfaction of users). For fixed $f_2$ we consider an optimization problem - looking for minimum of $f_3$, and for fixed $f_3$ we look for minimum of search costs - search optimum of $f_2$.

Objective functions allow capturing covergence and the convergence rate of construction of solutions much better than symbolic goals. Obviously every symbolic goal/ termination condition can be expressed as an objective function. For example, a very simple objective function can be the following: if the goal is satisfied the objective is set to 1, and if not to 0. Typically, much more complex objective functions are used to better express the evolutions of solutions.

Let $(A^*, X^*)$ denotes the set of totally optimal solutions. In particular $X^*$ denotes the set of optimal solutions, and $A^*$ the optimal search algorithms.

Let $Y$ be a metric space, where for every pair of its elements $x, y$ there is assigned the real number $D(x, y) \geq 0$, called *distance*, satisfying three conditions [11]:

1. $D(x, x) = 0$,
2. $D(x, y) = D(y, x)$
3. $D(x, y) + D(y, z) \geq D(x, z)$

The distance function can be defined in different ways, e.g., as the Hamming distance, Euclidean distance, $D(x, y) = 0$ if $x$ satisfies termination condition and $D(x, y) = 1$ otherwise. To keep it independent from representation, and to allow to compare different solving algorithms, we will fix the distance function to the absolute value of difference of the objective functions $D(x, y) = |f(x) - f(y)|$. We extend the definition of the distance from the pairs of points to the distance between a point and the set of points $D(x, Y) = min\{|f(x) - f(y)|; y \in Y\}$

In problem solving, we will be interested in the distance to the set of optimal solutions $Y^*$, i.e., in the distance

$D((a,x),(A^*,X^*))$, and in particular $D(x,X^*)$,$D(a,A^*)$, where $x \in X$ is the solution of the given problem instance, and $a \in A$ is the algorithm producing that solution.

**Definition 3.** *For any given problem instance, its solution evolved in the discrete time* $t = 0, 1, 2, ...$ *will be said to be*

- convergent *to the total optimum iff there exists such* $\tau$ *that for every* $t > \tau$ $D((a[t], x[t]), (A^*, X^*)) = 0$,
- asymptotically convergent *to the total optimum iff for every* $\varepsilon$, $\infty > \varepsilon > 0$, *there exists such* $\tau$ *that for every* $t > \tau \quad D((a[t], x[t]), (A^*, X^*)) < \varepsilon$,
- convergent with an error $\varepsilon$ *to the total optimum, where* $\infty > \varepsilon > 0$ *iff there exists such* $\tau$ *that for every* $t > \tau \quad D((a[t], x[t]), (A^*, X^*)) \leq \varepsilon$,
- divergent*, otherwise.*

If solution is convergent and $\tau$ is fixed, then the convergence is algorithmic, otherwise is nonalgorithmic. Asymptotic convergence is always nonalgorithmic (the time is unbounded).

**Definition 4.** *The* convergence rate *to the total optimum is defined as* $D((a[t], x[t]), (A^*, X^*)) - D((a[t+1], x[t+1]), (A^*, X^*))$.

The covergence rate describes the one-step performance of the algorithm, where the positive convergence rate means that the algorithm drifts towards the optimum and the negative rate signifies a drift away from the optimum. With positive convergence rate, the search algorithm will typically converge or asymptotically converge to the optimum. The best search algorithms will have typically a high convergence rate and a small number of steps to reach the optimum.

In the similar way, optimal and search optimal convergence and convergence rate can be defined. If the search algorithm is probabilistic, we use an expected value of the distance function.

## 3 Evolutionary Turing Machines

An evolutionary algorithm is a probabilistic beam hill climbing search algorithm directed by the fitness objective function. The beam (population size) maintains multiple search points, hill climbing means that only a current search point from the search tree is remembered, and a termination condition very often is set to the optimum of the fitness function.

**Definition 5.** *A generic* evolutionary algorithm (EA) *can be described in the form of the functional equation (recurrence relation) working in a simple iterative loop in discrete time* $t$*, called generations,* $t = 0, 1, 2, ...$ *[16, 8]:*

$$x[t+1] = s(v(x[t])), where$$

- $x \subseteq X$ - *is a* population *under a representation consisting of one or more individuals from the set* $X$ *(e.g., fixed binary strings for GAs, Finite State Machines for EP, parse trees for GP, vector of reals for ES),*
- $s$ - *is a* selection *operator (e.g., truncation, proportional, tournament),*
- $v$ - *is a* variation *operator (e.g., variants of mutation and crossover),*

$x[0]$ - *is an* initial *population,*

$F \subseteq X$ *is the set of final populations satisfying the termination condition (goal of evolution). The desirable termination condition is the optimum of the* fitness *function* $f(x[t])$ *- of the best individual in the population* $x[t] \in F$*, where* $f$ *is defined typically in the domain of nonnegative real numbers. In many cases this optimum is not possible to achieve or verify, thus the another stopping criterion is used (e.g., the maximum number of generations, the lack of progress through several generations.).*

The above equation is applicable to all typical EAs, including Genetic Algorithms (GA), Evolutionary Programming (EP), Evolution Strategies (ES), and Genetic Programming (GP). It is possible to use it to describe other emerging subareas like ant colony system [2], or particle swarm optimization [10]. Co-evolutionary algorithms use typically multiple populations, i.e., vectors of vectors are evolved. In fact, there is no restriction on the type of representation used. Sometimes only the order of variation and selection are reversed, i.e., selection is applied first, and variation second. Variation and selection depend on the fitness function. Of course, it is possible to think and implement more complex variants of evolutionary algorithms.

Evolutionary algorithms evolve population of solutions $x$, but they may be the subject of self-adaptation (like in ES) as well. This extension has been used in Evolution Strategies (although typically is limited only to ES parameter optimization, e.g., evolution of standard deviation in Gaussian mutation). Technically, the above means that the domain of the variation operator $v$, selection operator $s$, and the fitness function $f$ are extended to operate both on the population under representaion $x$ as well as on the encoding of the evolutionary algorithm.

Now, we define a formal model of Evolutionary Computation - an *Evolutionary Turing Machine* [5, 7].

**Definition 6.** *An* Evolutionary Turing Machine (ETM) *is a series of (possibly infinite) Turing Machines* $TM[t]$ *working on population* $x[t]$ *in generations* $t = 0, 1, 2, ...,$ *where*

- *each* $\delta[t]$ *transition function of Turing Machine* $TM[t]$ *represents (encodes) an evolutionary algorithm with population* $x[t]$*, and evolved in generations* $0, 1, 2, ..., t$*,*
- *only generation* $0$ *is given in advance, and any other generation depends on its predecessor only, i.e., the outcome of the generation* $t = 0, 1, 2, ...$ *is the pair* $(TM[t+1], x[t+1])$ *by applying the recursive variation* $v$ *and selection* $s$ *operators working on population* $x[t]$ *and on the evolutionary algorithm/transitions* $\delta[t]$ *as well,*
- $(TM[0], x[0])$ *is the initial Turing Machine operating on its input - an initial population* $x[0]$*,*
- *the goal (or halting) state of ETM is represented by any pair* $(TM[t], x[t])$ *satisfying the termination condition. The desirable termination condition is the optimum of the fitness performance measure*
  $f(TM[t], x[t], t) = f_1(f_2(M[t], t), f_3(x[t], t), t)$ *of the best individual from the population of solutions and evolutionary algorithms, where* $f_1$ *is an aggregating function,* $f_2$ *is an evolutionary algorithm fitness function, and* $f_3$ *is a problem-specific fitness function. The fitness function, without confusion, is denoted as*
  $f(TM[t], x[t]) = f_1(f_2(M[t]), f_3(x[t]))$ *in short. If the termination condition is satisified, then the ETM halts (t stops to be incremented), otherwise a new pair* $TM[t+1]$ *and its input/population* $x[t+1]$ *is generated.*

Note that because the fitness function can be the subject of evolution as well (it is a part of TM control $\delta$), evolution is an *infinite* process. Note also, that by definition, evolution is a *self-adaptive* process, i.e., an evolutionary algorithm/TM "hardware" $\delta[t]$ is changeable as well. This means that variation operators are extended both to evolve population $x$ and evolutionary algorithm itself. Currently in most cases an evolutionary algorithm is static, or at least changing much slower compared to the rate of change of its population $x$. Changing the transiton function $\delta[t]$ of the TM can be thought as some kind of evolvable hardware, or assuming fixed hardware we can think about reprogrammable evolutionary algorithms. This leads us immediately, following Turing's ideas, to the notion of the Universal Turing Machine and its extension - a *Universal Evolutionary Turing Machine*.

In general, every evolutionary algorithm ($EA$) can be encoded as an instance of TM $M$ operating on population $x$, i.e., pairs $(M, x)$ consisting of a Turing machine $M$ encoding of $EA$ (of course, for every algorithm, this is possible) and its input $x$ being the specific input of the Universal Turing Machine UTM. This applies to EAs operating on real numbers as well (we can encode/approximate real numbers in a digital form with an arbitrary precision). We can use for that, for instance, the IEEE 754 floating point representation standard. For recursive (algorithmic) solutions this will be sufficient (it may not suffice for nonrecursive solutions).

Then a *Universal Evolutionary Algorithm* represents instances of Turing Machines from a Universal Turing Machine representing all possible evolutionary algorithms. In other words, the *Universal Evolutionary Algorithm* consists of all pairs $(M, x)$, where $M$ is TM encoding of evolutionary algorithm, and $x$ is its input population.

We can define a *Universal Evolutionary Turing Machine* as an abstraction of all possible ETMs, in the similar way, as a Universal Turing Machine has beed defined, as an abstraction of all possible Turing Machines.

**Definition 7.** *A* Universal Evolutionary Turing Machine UETM *is a series of (possibly infinite) instances of Universal Turing Machines* $(M[t], x[t])$*:*

$$(M[t], x[t])_{t=0,1,2,...} = [(M[0], x[0]), (M[1], x[1]), ...],$$

*working on populations* $(M[t], x[t])$ *in generations* $t = 0, 1, 2, ...,$ *where*

- *each* $M[t]$ *represents (encodes) an evolutionary algorithm with population* $x[t]$*, and evolved in generations* $0, 1, 2, ..., t$*,*
- *only generation* 0 *is given in advance, and any other generation depends on its predecessor only, i.e., the outcome of generation* $t = 0, 1, 2, ...$ *is the pair* $(M[t + 1], x[t+1])$ *by applying the recursive variation* $v$ *and selection* $s$ *operators operating on population* $x$ *and (possibly) on evolutionary algorithm* $M$ *as well,*
- $(M[0], x[0])$ *is the initial evolutionary algorithm* $M[0]$ *operating on its input - an initial population* $x[0]$*,*
- *the goal (or halting) state of* $UETM$ *is represented by any pair* $(M[t], x[t])$ *satisfying the termination condition. The desirable termination condition is the optimum of the fitness performance measure* $f(M[t], x[t]) = f_1(f_2(M[t]), f_3(x[t]))$ *of the best individual from the population of solutions and evolutionary algorithms, where* $f_1$ *is an aggregating function,* $f_2$ *is an evolutionary algorithm fitness function, and* $f_3$ *is a problem-specific fitness function. If the termination condition is satisfied, then*

*the UETM halts (t stops to be incremented), otherwise a new pair $M[t+1]$ and its input/population $x[t+1]$ is generated.*

In other words, by a *Universal Evolutionary Turing Machine (UETM)* we mean such ETM which takes as the input a pair $(M[t], x[t])$ and behaves like ETM $M[t]$ with input $x[t]$ for $t = 0, 1, 2, ....$ UETM stops when ETM stops.

Note that ETM evolves TM transition functions (i.e., it can be said that it is a kind of evolvable hardware), whereas UETM shifts evolution to be done in software primarily.

### 3.1 Polymorphic Viruses as ETMs

We will describe polymorphic viruses and its environament (infected programs) as special instances of the Universal Evolutionary Turing Machine.

**Definition 8.** *A* polymorphic virus together with its infected environment *is a series of (possibly infinite) instances of Universal Turing Machines* $(V[t], x[t])$*:*

$$(V[t], x[t])_{t=0,1,2,...} = [(V[0], x[0]), (V[1], x[1]), ...],$$

*working on populations* $(V[t], x[t])$ *in generations* $t = 0, 1, 2, ...,$ *where*

- *each* $V[t]$ *represents (encodes) a polymorphic virus code in the environment* $x[t]$*, and evolved in generations* $0, 1, 2, ..., t$*,*
- *only generation* $0$ *is given in advance, and any other generation depends on its predecessor only, i.e., the outcome of generation* $t = 0, 1, 2, ...$ *is the pair* $(V[t+1], x[t+1])$ *by applying the recursive variation* $v$ *and selection* $s$ *operators operating on population* $x$ *and evolving virus* $V$ *by its mutation engine as well,*
- $(V[0], x[0])$ *is the initial polymorphic virus* $V[0]$ *operating on its input - an initial programming environment* $x[0]$*,*
- *the goal (or halting) state of* $UETM$ *is represented by any pair* $(V[t], x[t])$ *satisfying the termination condition. The desirable termination condition is the optimum of the fitness performance measure* $f(V[t], x[t]) = f_1(f_2(V[t]), f_3(x[t]))$ *of the best individual from the population of infected programs and polymorphic viruses, where* $f_1$ *is an aggregating function,* $f_2$ *is a polymorphic virus fitness function capturing its ability to reproduce and damage its environment* $x[t]$*, and* $f_3$ *is a problem-specific fitness function (describing performance of non-infected environment). If the termination condition is satisfied, then the UETM halts (t stops to be incremented), otherwise a new pair* $V[t+1]$ *and its input/population* $x[t+1]$ *is generated. From the point of view of anivirus techniques (infected environment),* $f$ *should be equal to* $f_3$*, i.e., virus has to be neutralized fully. From the point of view of polymorphic virus* $f$ *should mimic* $f_2$*.*

## 4 Cellular Space Models

Cellular space (or automata) based models include cellular automata, neural nets and automata networks. We will concentrate here on cellular automata and random automata networks.

Cellular automata are closely associated with the notion of artificial life. Artificial Life studies biological phenomena by attempting to reproduce them in an alternate media:

software (virtual life), wetware (alternative life) or hardware (synthetic life) [13]. Typical representatives of each category are cellular automata, biomolecular engineering, and cellular computers - each operating in its own environment. Research on self-replicating systems is important because it considers the possibility of self-reproduction, self-repair, and cooperation. Some systems [15] allow for self-replication (creation of a copy of the original structure), and some for universal computation (in the sense of universal Turing machine), and some for universal construction (a machine can construct whatever machine's description is given as input, including itself). Categories of cellular space models include cellular automata, non-uniform cellular automata [20] (not identical transitions rules for cells), and models with complex automata networks (not identical transitions rules and graph is not regular) [9].

Cellular automata [25], originally introduced by von Neumann and Ulam are a classical example of ALife [13].

**Definition 9.** A *cellular automaton* [9] is a pair $(D, \{M\})$ consisting of a cellular space $(D, \{Q\})$ (countably infinite, locally-finite regular (each node has the same degree) directed graph $D$ with states $Q$ assigned to each node) and common finite-state machine $M$ with input alphabet $\Sigma = Q^d$, $Q^d = Q \times ... \times Q$ ($d$ times) and local transition functions

$$\delta : Q \times \Sigma \to Q$$

The global evolution (dynamics) of a cellular automaton is a discrete dynamical system (self-map) $T : C \times C$, where $C = \Pi_i \, Q$ is a set of configurations (total states)

$$T(x)_i = \delta(x_i, x_{i_1}, ..., x_{i_d})$$

A cellular automaton operates locally as follows. A copy of a common finite state machine $M$ occupies each vertex of a regular graph (cellular space) which is a cell. Synchronously, each copy of $M$ looks up its input in the states $x_{i_1}, ..., x_{i_d}$ of its neighboring cells and its own state $x_i$, and then changes its state according to its local dynamics $\delta$. The cellular automaton performs its calculation by repeating these atomic local rules a possibly very large number of times for all sites resulting in an emergent behavior (global dynamics) of the whole system.

Von Neumann was interested whether minimal CA can be built such that UTM can be embedded (will be computation universal), can construct any other automaton (will be construction universal), can reproduce itself (subset of construction universality)

Von Neumann answered all these affirmatively. The bad news was that unfortunately the level of description of his universal constructor was too fine to be simulated, and additionally, the design was left by von Neumann incomplete (it was completed by his student A.W. Burks after von Neumann death). Von Neumann's 2D Euclidean (i.e., 4 NEWS neighbors, but truly 5 neighbors if to count itself as a neighbor) CA (with 29 states for each cell, i.e., $|Q| = 29$) , weakly rotation symmetric CA, consisting of many millions (!) of cells, was initially considered with the property of construction universality, and von Neumann showed that his CA was capable of simulating an arbitrary Turing machine (i.e., it was construction and computation universal).

Chris Langton in 1984 at Santa Fe Institute [12] dropped the criterion of universal construction (only reproduction, no universal constructibility or computability) and simulated pure self-reproduction. Langton was able to show the complete reproduction

process of its constructor (i.e., Langton's SR loop) by creating a snail-shaped pattern growing on a cellular space. He used an 8-state, 86-cell loop that required 108 replication/FSM transition rules. With respect to computer science, this result was a remarkable milestone, but with respect to universal reproduction, Langton's loop did nothing but propagated: the description was restricted to the development of the copy (no universal construction or computability). Other self-reproducing models can be found in [15].

### 4.1 Polymorphic Viruses in Cellular Space Models

Von Neumann's universal constructor should be able to model self-reproduction and payload of polymorphic viruses, however it is too complex and to general to be useful to model computer viruses. On the other hand, Langton's loop can model only pure self-reproduction, thus at most it can model computer worms without payload and environment, i.e., it is too simplistic for our purposes.

Additionally, the requirement that all cells have to be the same and with the same number of neighbors is too restrictive. For sure, infected programs and viruses contain different piece of code. Automata networks do not have such restrictions: cells can be modeled by different finite automata, and the number of neighbors can vary. Of course, cellular automata are a special case of automata networks, thus formally automata networks can model universal construction and self-reproduction too.

**Definition 10.** Formally, an automata network [9] is a pair $(D, \{M_i\})$ consisting of a cellular space $(D, \{Q_i\})$ (countably infinite, locally-finite directed graph $D$ with states $Q_i$ assigned to i-th node) and an associated family of finite-state machines $M_i$ (only finitely many of which are distinct) with input alphabet $\Sigma_i = Q_{i_1} \times ... \times Q_{id_i}$ and local transition functions

$$\delta_i : Q_i \times \Sigma_i \to Q_i$$

The global evolution (dynamics) of an automata network is best viewed as a discrete dynamical system (self-map) $T : C \times C$, where $C = \Pi_i Q_i$ is a set of configurations (total states)

$$T(x)_i = \delta_i(x_i, x_{i_1}, ..., x_{i_{d_i}})$$

An automata network operates locally as follows. A copy of a finite-state machine $M_i$ occupies each vertex (cell) $i$ of $D$. Synchronously, each copy $M_i$ looks up its input in the states $x_{i_1}, ..., x_{id_i}$ of its neighbor cells and its own state $x_i$, and then changes its state according to a local dynamics $\delta_i$. Next the atomic move is repeated any (possibly very large) number of times.

There are two alternative ways to model polymorphic viruses as automata networks: with or without explicit fitness function. In first approach we partition the set of all cells $M_i$ into cells infected by viruses $V_i$ and healthy cells $H_i$. Then the transition functions are responsible for replication and payload of viruses. The final states of automata represent the goal of normal or infected cells. In this approach is difficult to capture effects of viruses and anti-virus software. To capture that instead of terminal states, we can use explicit fitness functions. Each cell will have its associated fitness function $f(M_i) = f_1(f_2(V_i), f_3(H_i))$, where $f_1$ is an aggregating function, $f_2$ is a fitness function capturing infection of cell $M_i$ converted to infected cell $V_i$, and $f_3$ is a fitness function of the healthy cell. For healthy cell $f_2$ is equal to 0. From the point of view of anivirus

techniques (infected environment), $f$ should be equal to $f_3$, i.e., virus has to be neutralized fully. From the point of view of polymorphic virus $f$ should mimic $f_2$.

## 5 The $-Calculus Process Algebra of Bounded Rational Agents

The $-calculus is a mathematical model of processes capturing both the final outcome of problem solving as well as the interactive incremental way how the problems are solved. The $-calculus is a process algebra of Bounded Rational Agents for interactive problem solving targeting intractable and undecidable problems. It has been introduced in the late of 1990s [3, 4, 24, 6]. The $-calculus (pronounced *cost* calculus) is a formalization of resource-bounded computation (also called anytime algorithms), proposed by Dean, Horvitz, Zilberstein and Russell in the late 1980s and early 1990s [19]. Anytime algorithms are guaranteed to produce better results if more resources (e.g., time, memory) become available. The standard representative of process algebras, the $\pi$-calculus [17] is believed to be the most mature approach for concurrent systems.

The $-calculus rests upon the primitive notion of *cost* in a similar way as the $\pi$-calculus was built around a central concept of *interaction*. Cost and interaction concepts are interrelated in the sense that cost captures the quality of an agent interaction with its environment. The unique feature of the $-calculus is that it provides a support for problem solving by incrementally searching for solutions and using cost to direct its search. The basic $-calculus search method used for problem solving is called $k\Omega$-optimization. The $k\Omega$-optimization represents this "impossible" to construct, but "possible to approximate indefinitely" universal algorithm. It is a very general search method, allowing simulation of many other search algorithms, including A*, minimax, dynamic programming, tabu search, or evolutionary algorithms. Each agent has its own $\Omega$ search space and its own limited horizon of deliberation with depth $k$ and width $b$. Agents can cooperate by selecting actions with minimal costs, can compete if some of them minimize and some maximize costs, and be impartial (irrational or probabilistic) if they do not attempt optimize (evolve, learn) from the point of view of the observer. It can be understood as another step in the never ending dream of universal problem solving methods recurring throughout all computer science history. The $-calculus is applicable to robotics, software agents, neural nets, and evolutionary computation. Potentially it could be used for design of cost languages, cellular evolvable cost-driven hardware, DNA-based computing and molecular biology, electronic commerce, and quantum computing. The $-calculus leads to a new programming paradigm *cost languages* and a new class of computer architectures *cost-driven computers*.

In $-calculus everything is a cost expression: agents, environment, communication, interaction links, inference engines, modified structures, data, code, and meta-code. $-expressions can be simple or composite. Simple $-expressions $\alpha$ are considered to be executed in one atomic indivisible step. Composite $-expressions $P$ consist of distinguished components (simple or composite ones) and can be interrupted. The $-Calculus syntax is presented below.

**Definition 11.** *The $-calculus* The set of $-calculus process expressions consists of simple $-expressions $\alpha$ and composite $-expressions $P$, and is defined by the following syntax:

| | | | |
|---|---|---|---|
| $\alpha$ | ::= | $(\$_{i\in I}\ P_i)$ | cost |
| | \| | $(\rightarrow_{i\in I}\ c\ P_i)$ | send $P_i$ with evaluation through channel $c$ |
| | \| | $(\leftarrow_{i\in I}\ c\ X_i)$ | receive $X_i$ from channel $c$ |
| | \| | $(\prime_{i\in I}\ P_i)$ | suppress evaluation of $P_i$ |
| | \| | $(a_{i\in I}\ P_i)$ | defined call of simple \$-expr. $a$ with parameters $P_i$ |
| | \| | $(\bar{a}_{i\in I}\ P_i)$ | negation of defined call of simple \$-expression $a$ |

| | | | |
|---|---|---|---|
| $P$ | ::= | $(\circ_{i\in I}\ \alpha\ P_i)$ | sequential composition |
| | \| | $(\parallel_{i\in I}\ P_i)$ | parallel composition |
| | \| | $(\uplus_{i\in I}\ P_i)$ | cost choice |
| | \| | $(\biguplus_{i\in I}\ P_i)$ | adversary choice |
| | \| | $(\sqcup_{i\in I}\ P_i)$ | general choice |
| | \| | $(f_{i\in I}\ P_i)$ | defined process call $f$ with param. $P_i$, and its associated definition $(:=\ (f_{i\in I}\ X_i)\ R)$ with body $R$ |

The indexing set $I$ is a possibly countably infinite. In the case when $I$ is empty, we write empty parallel composition, general, cost and adversary choices as $\perp$ (blocking), and empty sequential composition ($I$ empty and $\alpha = \varepsilon$) as $\varepsilon$ (invisible transparent action, which is used to mask, make invisible parts of \$-expressions). Adaptation (evolution/upgrade) is an essential part of \$-calculus, and all \$-calculus operators are infinite (an indexing set $I$ is unbounded). The \$-calculus agents interact through send-receive pair as the essential primitives of the model.

Sequential composition is used when \$-expressions are evaluated in a textual order. Parallel composition is used when expressions run in parallel and it picks a subset of non-blocked elements at random. Cost choice is used to select the cheapest alternative according to a cost metric. Adversary choice is used to select the most expensive alternative according to a cost metric. General choice picks one non-blocked element at random. General choice is different from cost and adversary choices. It uses guards satisfiability. Cost and adversary choices are based on cost functions. Call and definition encapsulate expressions in a more complex form (like procedure or function definitions in programming languages). In particular, they specify recursive or iterative repetition of \$-expressions.

Simple cost expressions execute in one atomic step. Cost functions are used for optimization and adaptation. The user is free to define his/her own cost metrics. Send and receive perform handshaking message-passing communication, and inferencing. The suppression operator suppresses evaluation of the underlying \$-expressions. Additionally, a user is free to define her/his own simple \$-expressions, which may or may not be negated.

We define the operational semantics of the \$-calculus using the $k\Omega$-search that captures the dynamic nature and incomplete knowledge associated with the construction of the problem solving tree.

The basic $-calculus problem solving method, the $k\Omega$-optimization, is a very general search method providing meta-control, and allowing to simulate many other search algorithms, including A*, minimax, dynamic programming, tabu search, or evolutionary algorithms [19]. The problem solving works iteratively: through select, examine and execute phases. In the select phase the tree of possible solutions is generated up to $k$ steps ahead, and agent identifies its alphabet of interest for optimization $\Omega$. This means that the tree of solutions may be incomplete in width and depth (to deal with complexity). However, incomplete (missing) parts of the tree are modeled by silent $-expressions $\varepsilon$, and their cost estimated (i.e., not all information is lost). The above means that $k\Omega$-optimization may be (if some conditions are satisfied) complete and optimal. In the examine phase the trees of possible solutions are pruned minimizing cost of solutions, and in the execute phase up to $n$ instructions are executed. Moreover, because the $ operator may capture not only the cost of solutions, but the cost of resources used to find a solution, we obtain a powerful tool to avoid methods that are too costly, i.e., the $-calculus directly minimizes search cost. This basic feature, inherited from anytime algorithms, is needed to tackle directly hard optimization problems, and allows to solve total optimization problems (the best quality solutions with minimal search costs). The variable $k$ refers to the limited horizon for optimization, necessary due to the unpredictable dynamic nature of the environment. The variable $\Omega$ refers to a reduced alphabet of information. No agent ever has reliable information about all factors that influence all agents behavior. To compensate for this, we mask factors where information is not available from consideration; reducing the alphabet of variables used by the $-function. By using the $k\Omega$-optimization to find the strategy with the lowest $-function, meta-system finds a satisficing solution, and sometimes the optimal one. This avoids wasting time trying to optimize behavior beyond the foreseeable future. It also limits consideration to those issues where relevant information is available. Thus the $k\Omega$ optimization provides a flexible approach to local and/or global optimization in time or space. Technically this is done by replacing parts of $-expressions with invisible $-expressions $\varepsilon$, which remove part of the world from consideration (however, they are not ignored entirely - the cost of invisible actions is estimated).

### 5.1 Polymorphic Viruses in $-Calculus

In terms of the $-calculus, infected by polymorphic viruses programs and their environemnt are a parallel composition of component programs

$$( \parallel_i (k\Omega_i[t] \;\; x_i[t])),$$

where $k\Omega_i[t]$ represents a viral part operating on a healthy program (its host) $x_i[t]$ in time $t = 0, 1, 2, ....$ For healthy programs their viral part is empty (or, more precisely, the identity function). Each infected component performs the $k\Omega$-optimization looking for the minimum of the cost function

$$\$(k\Omega_i[t], x_i[t]) = \$_1(\$_2(k\Omega_i[t]), \$_3(x_i[t])),$$

where $\$_1$ is an aggregating cost funtion, $\$_2$ is a polymorhic virus cost function capturing its self-reproduction and damaging other programs payload capabilities, and $\$_3$ is the problem-specific cost funtion capturing the goal of computation of the host program.

Once again from the point of view of the anti-virus software the desirable situation is when $\$ = \$_3$, and from the point of view of computer virus the desirable goal if $\$ = \$_2$.

## 6 Conclusions and Future Work

This paper outlined three models of polymorphic viruses allowing to capture their self-reproduction, damaging payload, and their host program environment. These models have been derived using Evolutionary Turing Machine, random automata networks, and the $-calculus. Such result seems be very encouraging in the situation, when no models of polymorphic viruses existed, and their nature is not well understood. This was because conventional algorithms and Turing Machine do not capture properly self-reproduction and potential creation of new types of viruses. This was the main reason why von Neumann had to use a new model of cellular automata instead of Turing Machine, when he wanted to design his universal constructor. Of course, these results are preliminary and much more work is needed. The follow up work, including some simulation is under way [18].

## Bibliography


[1] Cohen F., Computational Aspects of Computer Viruses, Computers & Security, vol.8, no.4, 1989, 325-344.

[2] Dorigo M., Gambardella L.M., Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, IEEE Trans. on Evolutionary Computation, vol.1, no.1, 1997, 53-66.

[3] Eberbach E., Expressiveness of $-Calculus: What Matters?, in: Advances in Soft Computing, in: (eds. M.Klopotek, M.Michalewicz, S.T.Wierzchon) Adances in Soft Computing, Proc. of the 9th Intern. Symp. on Intelligent Information Systems IIS'2000, Bystra, Poland, Physica-Verlag, 2000, 145-157.

[4] Eberbach E., Goldin D., Wegner P., Turing's Ideas and Models of Computation, in: (ed. Ch.Teuscher) Alan Turing: Life and Legacy of a Great Thinker, Springer-Verlag, 2004, 159-194.

[5] Eberbach E., Toward a Theory of Evolutionary Computation, BioSystems, vol. 82, issue 1, 2005, 1-19.

[6] Eberbach E., $-Calculus of Bounded Rational Agents: Flexible Optimization as Search under Bounded Resources in Interactive Systems, Fundamenta Informaticae, vol.68, no.1-2, 2005, 47-102.

[7] Eberbach E., The Role of Completeness in Convergence of Evolutionary Algorithms, Proc. 2005 Congress on Evolutionary Computation CEC'2005, vol.2, Edinburgh, Scotland, 2005, 1706-1713.

[8] Fogel D.B., An Introduction to Evolutionary Computation, Tutorial, Congress on Evolutionary Computation CEC2001, Seoul, Korea, 2001.

[9] Garzon M., Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks, An EATCS series, Springer-Verlag, 1995.

[10] Kennedy J., Eberhart R., Tutorial on Particle Swarm Optimization, 2002 World Congress on Computational Intelligence WCCI'2002, Honolulu, HI, 2002.

[11] Kuratowski K., Introduction to Set Theory and Topology, PWN, Warsaw, 1977.

[12] Langton Ch., Self-Reproduction in Cellular Automata, Physica 10D, 1984, 135-144.

[13] Langton Ch., Artificial Life: An Overview, The MIT Press, 1996.

[14] Leitold F., Mathematical Model of Computer Viruses, EICAR Best Paper Proceedings, 2000, 194-217.

[15] Lohn J., Self-Replication Systems in Cellular Space Models Tutorial, Genetic Programming Conference GP97, Stanford University, 1997.

[16] Michalewicz Z., Fogel D.B., How to Solve It: Modern Heuristics, Springer-Verlag, 2000.

[17] Milner R., Parrow J., Walker D., A Calculus of Mobile Processes, I & II, Information and Computation 100, 1992, 1-77.

[18] Ramanadham S., Study of Polymorphic Computer Viruses, Master Project, CIS Dept., Univ. of Mass. Dartmouth, 2006.

[19] Russell S., Norvig P., Artificial Intelligence: A Modern Approach, Prentice Hall, 1995 (2nd ed. 2003).

[20] Sipper M., Cellular Programming: The Evolution of Parallel Cellular Machines, Genetic Programming Conf. GP-98, Tutorial, University of Wisconsin, Madison, WI, 1998.

[21] Spafford E.H., Computer Viruses as Artificial Life, in Ch. Langton (ed.), Artificial Life: An Overview, The MIT Press, 1996, 249-265.

[22] Tanenbaum A., Modern Operating Systems, Prentice Hall, 2nd edition, 2001.

[23] Thimbleby H., Anderson S., Cairns P., A Framework for Modelling Trojans and Computer Virus Infection, Computer Journal, 41(7), 1999, 444-458.

[24] Wegner P., Eberbach E., New Models of Computation, The Computer Journal, 47(1), The British Computer Society, Oxford University Press, 2004, 4-9.

[25] Von Neumann J., Theory of Self-Reproducing Automata, (edited and completed by Burks A.W.), Univ. of Illinois Prfess, 1966.