Computing Partial Recursive Functions by Virus Machines

Álvaro Romero-Jiménez(✉), Luis Valencia-Cabrera, Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez

Research Group on Natural Computing,
Department of Computer Science and Artificial Intelligence,
Universidad de Sevilla, Avda. Reina Mercedes S/n, 41012 Seville, Spain
{romero.alvaro,lvalencia,ariscosn,marper}@us.es

Abstract. Virus Machines are a computational paradigm inspired by the manner in which viruses replicate and transmit from one host cell to another. This paradigm provides non-deterministic sequential devices. Non-restricted Virus Machines are unbounded Virus Machines, in the sense that no restriction on the number of hosts, the number of instructions and the number of viruses contained in any host along any computation is placed on them. The computational completeness of these machines has been obtained by simulating register machines. In this paper, Virus Machines as function computing devices are considered. Then, the universality of non-restricted virus machines is proved by showing that they can compute all partial recursive functions.

1 Introduction

A new computational paradigm inspired by the replications and transmissions of viruses was introduced in [1]. The computational devices in this paradigm are called *Virus Machines* and they consist of several processing units, called *hosts*, connected to each other by *transmission channels*. A host can be viewed as a group of cells (being part of a colony, organism, system, organ or tissue). Each cell in the group will contain at most one virus, but we will not take into account the number of cells in the group, we will only focus on the number of viruses that are present in some of the cells of that group (not every cell in the group does necessarily hold a virus). Only one type of viruses is considered. Channels allow viruses to be transmitted from one host to another or to the environment of the system. Each channel has a natural number (the *weight* of the channel) associated with it, indicating the number of copies of the virus that will be generated and transmitted from an original one (i.e., one virus may replicate, generating a number of copies to be transmitted to the target host group of cells). Each transmission channel is closed by default and it can be opened by a control instruction unit. Specifically, there is an *instruction-channel control network* that allows opening a channel by means of an activated instruction. In that moment, the opened channel allows a virus (only one virus) to replicate

G. Rozenberg et al. (Eds.): CMC 2015, LNCS 9504, pp. 353–368, 2015.
DOI: 10.1007/978-3-319-28475-0_24

and transmit through it. Instructions are activated individually according to a protocol given by an *instruction transfer network*, so that only one instruction is enabled in each computation step. That is, an instruction activation signal is transferred to the network to activate instructions in sequence.

In this work, Virus Machines as computing function devices are introduced. For this purpose, we deal with Virus Machines having input hosts, allowing us to introduce some additional numbers of viruses (encoding the information) in certain distinguished hosts as an input to the Virus Machine. The universality of non-restricted Virus Machines (Virus Machines where there is no restriction on the number of hosts, the number of instructions and the number of viruses contained in any host along any computation) working in the computing mode is proved by showing that they can compute all partial recursive functions.

This paper is structured as follows. First, some preliminaries are briefly introduced in order to make the work self-contained. Then, in Sect. 3, we formally define the computing model of virus machines. Section 4 is devoted to discuss the power of non-restricted Virus Machines, and their computational completeness (via computing partial recursive functions) is stated. Finally, in Sect. 5 the main conclusions of this work are summarized and some suggestions for possible lines of future research are outlined.

2 Preliminaries

In this section some basic concepts needed throughout this paper are introduced, thus making it self-contained.

2.1 Sets and Functions

In this paper $\mathbb{Z}$ denotes the set of integer numbers, $\mathbb{Z}_{>0}$ the set of positive integers, and $\mathbb{N} = \mathbb{Z}_{\geq 0}$ the set of non-negative integers or natural numbers.

A function from a set A to a set B is a subset of $A \times B$ such that every element of A is related through f with at most one element of B. The domain of f, $\mathrm{dom}(f)$, is the subset of A consisting of all the elements for which f is defined. If $\mathrm{dom}(f) = A$ we say that the function is total, and denote it by $f : A \to B$. Otherwise, we say that the function is partial, and denote it by $f : A \dashrightarrow B$.

2.2 Graphs

An *undirected graph* G is a pair (V, E), where V is a finite set and E is a subset of $\big\{\{x, y\} \mid x \in V, y \in V, x \neq y\big\}$. The set V is called the *vertex set* of G, and its elements are called *vertices*. The set E is called the *edge set* of G, and its elements are called *edges*. If $e = \{x, y\} \in E$ is an edge of G, then we say that edge e is incident on vertices x and y. In an undirected graph, the *degree* of a vertex x is the number of edges incident on it. A *bipartite graph* G is an undirected graph (V, E) in which V can be partitioned into two sets V_1, V_2 such that $\{u, v\} \in E$

implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$; that is, all edges are arranged between the two sets V_1 and V_2 (see [3] for details).

A *directed graph* G is a pair (V, E), where V is a finite set and E is a subset of $V \times V$. The set V is called the vertex set of G, and its elements are called vertices. The set E is called the *arc set* of G, and its elements are called *arcs*. In a directed graph, the *out-degree* of a vertex is the number of arcs leaving it, and the *in-degree* of a vertex is the number of arcs entering it.

3 Virus Machines

In what follows we formally define the syntax of the Virus Machines (see [1] for more details).

Definition 1. *A* Virus Machine Π *of degree* (p, q)*, with* $p \geq 1, q \geq 1$*, is a tuple* $(\Gamma, H, I, D_H, D_I, G_C, n_1, \dots, n_p, i_{\text{start}}, h_{\text{out}})$*, where:*

- $\Gamma = \{v\}$ *is the singleton alphabet;*
- $H = \{h_1, \dots, h_p\}$ *and* $I = \{i_1, \dots, i_q\}$ *are ordered sets such that* $v \notin H \cup I$ *and* $H \cap I = \emptyset$*;*
- $D_H = (H \cup \{h_{\text{out}}\}, E_H, w_H)$ *is a weighted directed graph, verifying that* $E_H \subseteq H \times (H \cup \{h_{\text{out}}\})$*,* $(h, h) \notin E_H$ *for each* $h \in H$*,* $\text{out-degree}(h_{\text{out}}) = 0$*, and* w_H *is a mapping from* E_H *to* $\mathbb{Z}_{>0}$*;*
- $D_I = (I, E_I, w_I)$ *is a weighted directed graph, where* $E_I \subseteq I \times I$*,* w_I *is a mapping from* E_I *to* $\mathbb{Z}_{>0}$ *and, for each vertex* $i_j \in I$*, the out-degree of* i_j *is less than or equal to 2;*
- $G_C = (V_C, E_C)$ *is an undirected bipartite graph, where* $V_C = I \cup E_H$*, being* $\{I, E_H\}$ *the partition associated with it (i.e., all edges go between the two sets* I *and* E_H*). In addition, for each vertex* $i_j \in I$*, the degree of* i_j *in* G_C *is less than or equal to 1;*
- $n_j \in \mathbb{N}$ *(*$1 \leq j \leq p$*) and* $i_{\text{start}} \in I$*;*
- $h_{\text{out}} \notin I \cup \{v\}$ *and* h_{out} *is denoted by* h_0 *in the case that* $h_{\text{out}} \notin H$*.*

A Virus Machine $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \dots, n_p, i_{\text{start}}, h_{\text{out}})$ of degree (p, q) can be viewed as an ordered set of p *hosts* labelled with $h_1, \dots, h_p$ (where each host h_j, $1 \leq j \leq p$, initially contains exactly n_j *viruses* –copies of the symbol v–), and an ordered set of q *control instruction units* labelled with $i_1, \dots, i_q$. Symbol h_{out} represents the *output region* of the system (we use the term *region* to refer to host h_{out} in the case that $h_{\text{out}} \in H$ and to refer to the environment in the case that $h_{\text{out}} = h_0$). Arcs $(h_s, h_{s'})$ from D_H represent *transmission channels* through which viruses can travel from host h_s to $h_{s'}$.

Each channel is *closed* by default, and so it remains until it is opened by a control instruction (which is attached to the channel by means of an edge in graph G_C) when that instruction is *activated*. Furthermore, each channel $(h_s, h_{s'})$ is assigned with a positive integer weight, denoted by $w_{s,s'}$, which indicates the number of viruses that will be transmitted/replicated to the receiving host of the channel.

Arcs $(i_j, i_{j'})$ from D_I represent *instruction transfer paths*, and they have a weight, denoted by $w_{j,j'}$, associated with it. Finally, the undirected bipartite graph G_C represents the *instruction-channel network* by which an edge $\{i_j, (h_s, h_{s'})\}$ indicates a control relationship between instruction i_j and channel $(h_s, h_{s'})$: when instruction i_j is activated, the channel $(h_s, h_{s'})$ is opened.

A *configuration* $\mathcal{C}_t$ of a Virus Machine at an instant t is described by a tuple $(a_{1,t}, \dots, a_{p,t}, u_t, e_t)$, where $a_{1,t}, \dots, a_{p,t}$ and e_t are non-negative integers and $u_t \in I \cup \{\#\}$, with $\# \notin \{v\} \cup H \cup \{h_0\} \cup I$. The meaning is the following: at instant t the host h_s of the system contains exactly $a_{s,t}$ viruses, the output region h_{out} contains exactly e_t viruses and, if $u_t \in I$, then the control instruction unit u_t will be activated at step $t+1$. Otherwise, if $u_t = \#$, then no further instruction will be activated. The *initial configuration* of the system is the configuration $\mathcal{C}_0 = (n_1, \dots, n_p, i_{\text{start}}, 0)$.

A configuration $\mathcal{C}_t = (a_{1,t}, \dots, a_{p,t}, u_t, e_t)$ is a *halting configuration* if and only if u_t is the object $\#$. A non-halting configuration $\mathcal{C}_t = (a_{1,t}, \dots, a_{p,t}, u_t, e_t)$ yields configuration $\mathcal{C}_{t+1} = (a_{1,t+1}, \dots, a_{p,t+1}, u_{t+1}, e_{t+1})$ in one *transition step*, denoted by $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$, if we can pass from $\mathcal{C}_t$ to $\mathcal{C}_{t+1}$ as follows:

1. First, given that $\mathcal{C}_t$ is a non-halting configuration, we have $u_t \in I$. So the control instruction unit u_t is activated.
2. Let us assume that instruction u_t is attached to channel $(h_s, h_{s'})$. Then this channel will be opened and:
   - If $a_{s,t} \geq 1$, then a virus (only one virus) is consumed from host h_s and $w_{s,s'}$ copies of v are produced in host $h_{s'}$ (if $s' \neq out$) or in the output region h_{out}.
   - If $a_{s,t} = 0$, then there is no transmission of viruses.
3. Let us assume that instruction u_t is not attached to any channel $(h_s, h_{s'})$. Then there is no transmission of viruses.
4. Object $u_{t+1} \in I \cup \{\#\}$ is obtained as follows:
   - Let us suppose that out-degree$(u_t) = 2$, that is, there are two different instructions $u_{t'}$ and $u_{t''}$ such that $(u_t, u_{t'}) \in E_I$ and $(u_t, u_{t''}) \in E_I$.
     - If instruction u_t is attached to a channel $(h_s, h_{s'})$ and $a_{s,t} \geq 1$ then u_{t+1} is the instruction corresponding to the *highest* weight path.
     - If instruction u_t is attached to a channel $(h_s, h_{s'})$ and $a_{s,t} = 0$ then u_{t+1} is the instruction corresponding to the *lowest* weight path.
     - If both weights are equal or if instruction u_t is not attached to a channel, then the next instruction u_{t+1} is either $u_{t'}$ or $u_{t''}$, selected in a non-deterministic way.
   - If out-degree$(u_t) = 1$ then the system behaves deterministically and u_{t+1} is the instruction that verifies $(u_t, u_{t+1}) \in E_I$.
   - If out-degree$(u_t) = 0$ then u_{t+1} is object $\#$ and configuration $\mathcal{C}_{t+1}$ is a halting configuration.

A *computation* of a Virus Machine Π is a (finite or infinite) sequence of configurations such that: (a) the first element is the initial configuration $\mathcal{C}_0$ of the system; (b) for each $n \geq 1$, the n-th element of the sequence is obtained from the previous element in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last element is a halting configuration. All the computations start from the initial configuration and proceed as stated above; only halting computations give a result, which is encoded in the contents of the output region for the halting configuration.

Definition 2. *A Virus Machine Π* with input *of degree (p, q, r), $p \geq 1$, $q \geq 1$, $r \geq 1$, is a tuple $(\Gamma, H, H_r, I, D_H, D_I, G_C, n_1, \dots, n_p, i_{\text{start}}, h_{\text{out}})$, where:*

- *$(\Gamma, H, I, D_H, D_I, G_C, n_1, \dots, n_p, i_{\text{start}}, h_{\text{out}})$ represents a Virus Machine of degree (p, q).*
- *$H_r = \{h_{j_1}, \dots, h_{j_r}\} \subseteq H$ is the ordered set of r input hosts and $h_{\text{out}} \notin H_r$.*

The *initial configuration* of Π with input $(\alpha_1, \dots, \alpha_r)$ is the configuration $(m_1, \dots, m_p, i_{start}, 0)$, where $m_j = n_j + \alpha_j$, if $j \in \{j_1, \dots, j_r\}$, and $m_j = n_j$ otherwise. Therefore, in a Virus Machine with input we have an initial configuration associated with each $(\alpha_1, \dots, \alpha_r) \in \mathbb{N}^r$. A computation of a Virus Machine Π with input $(\alpha_1, \dots, \alpha_r)$, denoted by $\Pi + (\alpha_1, \dots, \alpha_r)$, starts with the initial configuration $(m_1, \dots, m_p, i_{\text{start}}, 0)$ and proceeds as stated above.

In this paper we work with Virus Machines working in the *computing mode.* That is, the result of a computation of a Virus Machine Π with input $(\alpha_1, \dots, \alpha_r)$ is the total number n of viruses sent to the output region during the computation. We say that number n is *computed* by the Virus Machine $\Pi + (\alpha_1, \dots, \alpha_r)$. We denote by $N\big(\Pi + (\alpha_1, \dots, \alpha_r)\big)$ the set of all natural numbers computed by $\Pi + (\alpha_1, \dots, \alpha_r)$.

Throughout this paper, due to technical reasons, we consider $h_{\text{out}} \in H$, that is, the output region of a Virus Machine will be a host.

3.1 Virus Machines as Function Computing Devices

Virus Machines can work in several modes. In this section we introduce a particular kind of virus machines working in the computing mode providing function computing devices.

Definition 3. *Let $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ be a partial function. We say that f is computable by a Virus Machine Π with k input hosts working in the computing mode if the following holds: for each $(x_1, \dots, x_k) \in \mathbb{N}^k$,*

- *If $(x_1, \dots, x_k) \in \text{dom}(f)$ and $f(x_1, \dots, x_k) = z$, then every computation $\Pi + (x_1, \dots, x_k)$ is a halting computation with output z.*
- *If $(x_1, \dots, x_k) \notin \text{dom}(f)$, then every computation $\Pi + (x_1, \dots, x_k)$ is a non-halting computation.*

The concept of computation of a subset of $\mathbb{N}^k$ is introduced below, via function computing Virus Machines.

Definition 4. *Let $A \subseteq \mathbb{N}^k$ be a set of k-tuples of natural numbers. We say that A is computed by a Virus Machine Π with k input hosts working in the computing mode if Π computes the partial characteristic function $\mathcal{C}_A^*$ of A, defined as follows:*

$$\mathcal{C}_A^*(x_1, \ldots, x_k) = \begin{cases} 1, & \textit{if } (x_1, \ldots, x_k) \in A \\ \textit{undefined}, & \textit{otherwise} \end{cases}$$

4 The Universality of Non-restricted Virus Machines

A *non-restricted Virus Machine* is a Virus Machine such that there is no restriction on the number of hosts, the number of instructions and the number of viruses contained in any host along any computation.

For each $p, q, n \geq 1$, we denote by $NVM(p, q, n)$ the family of all subsets of $\mathbb{N}$ computed by Virus Machines with at most p hosts, q instructions, and all hosts having at most n viruses at any instant of each computation. If one of the numbers p, q, n is not bounded, then it is replaced with $*$. In particular, $NVM(*, *, *)$ denotes the family of all subsets of natural numbers computed by non-restricted Virus Machines.

4.1 Computing Partial Recursive Functions by Virus Machines

In this section, the computational completeness of non-restricted Virus Machines working in the computing mode is established. Specifically, we prove that they can compute all partial recursive functions. Indeed, we will design non-restricted Virus Machines that:

1. Compute the *basic or initial functions*: constant zero function, successor function and projection functions.
2. Compute the *composition* of functions, from Virus Machines computing the functions to be composed.
3. Compute the *primitive recursion* of functions, from Virus Machines computing the functions that participate in the recursion.
4. Compute the *unbounded minimization* of functions, from a Virus Machine computing the function to be minimized.

4.2 Modules

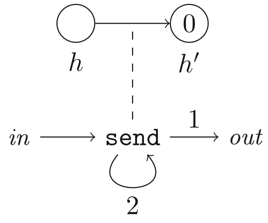
In order to ease the design of the Virus Machines computing any partial recursive function, the construction of such Virus Machines will be made in a modular manner. A *module* can be seen as a Virus Machine without output host, with the initial instruction marked as the *in* instruction and with at least one instruction marked as an *out* instruction. The *out* instructions must have outdegree less than two, so that they can still be connected to another instruction. This way, a module m_1 can be plugged in before another module m_2 or Virus Machine instruction i by simply connecting the *out* instructions of m_1 with the *in* instruction of m_2 or with the instruction i.

The layout of a module must be carefully design to avoid conflicts with other modules and to allow the module to be executed any number of times. To achieve the first condition, we will consider that all the hosts (with the only exception of those belonging to the parameters of the module) and instructions of a module are individualized for that module, being distinct from the ones of any other module or Virus Machine. The second condition is met if we ensure that, after the execution of the module, all its hosts except its parameters contain the same number of viruses as before the execution.

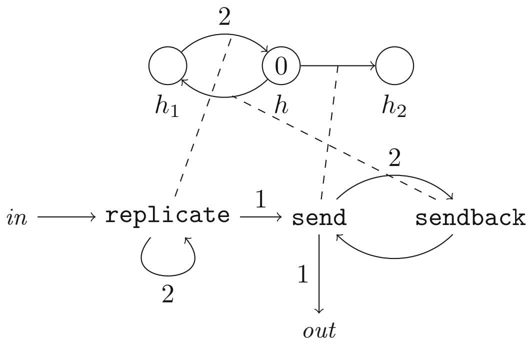
In this paper we consider two types of modules: action modules and predicate modules. For the action modules we require all of its *out* instructions to be connected to the *in* instruction of the following module, or to the following instruction of the Virus Machine. For the predicate modules we consider its *out* instructions to be divided in two subsets: the *out* instructions representing a *yes* answer and the *out* instructions representing a *no* answer of the predicate. For each of these subsets, all of its instructions have to be connected to the same module *in* instruction or Virus Machine instruction.

The library of modules used in this paper consists of the following modules (we name the action modules as verbs and the predicate modules as questions):

- `EMPTY`(h): action module that sets to zero the number of viruses in host h.
  To implement this module we only need to introduce an internal host h', initially with zero viruses, and associate with the channel from h to h' an action that transfers all the viruses from h. Note that host h' may end with a nonzero number of viruses, but this does not prevent the module to be reused, because h' plays a passive role.



- `ADD`(h_1, h_2): action module that adds to host h_2 the number of viruses in host h_1, without modifying the number of viruses in h_1.
  This module is implemented as follows:

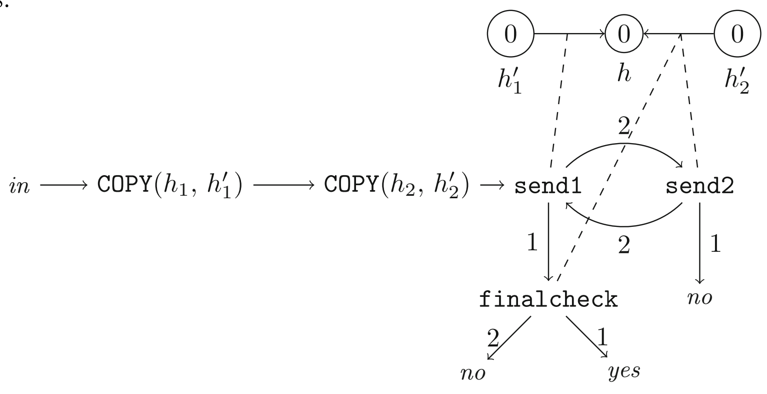


This way, the module starts by transferring one by one all the viruses from h_1 to h, duplicating them along the way. Then it sends, again and again, one virus from h to h_2 and another one from h to h_1, until there are no more viruses left. It is clear then that when the module ends, the host h_1 retains its initial number of viruses, the host h is empty (thus allowing the module to be reused), and the host h_2 has a number of viruses equal to the sum of the initial number of viruses in h_1 and h_2.

- `COPY`(h_1, h_2): action module that sets the number of viruses in h_2 the same as in h_1, without modifying the number of viruses in h_1.
  This module is implemented by the following concatenation of modules:

$$\mathit{in} \to \texttt{EMPTY}(h_2) \to \texttt{ADD}(h_1, h_2) \to \mathit{out}$$

  That is, we first get rid of all the viruses from h_2, and then add the viruses from h_1, so h_2 ends with the same number of viruses as h_1. Also observe that the module `ADD`(h_1, h_2) does not modify the number of viruses in h_1, what will be important later.
- `SET`(h, n): action module that sets to n the number of viruses in host h. This module is implemented simply by introducing an internal host h' with initial number of viruses n and using the module `COPY`(h', h).
- `AREEQUAL?`(h_1, h_2): predicate module that checks if the number of viruses in hosts h_1 and h_2 coincides.
  This module is implemented as follows, where h'_1, h'_2 and h are new internal hosts:



  We first copy the contents of h_1 and h_2 into the internal hosts h'_1 and h'_2, so that they do not get modified. Then, in turns, we send one virus from h'_1 to h and then another one from h'_2 to h. If the latter can not be done, this is because the contents of h_1 were greater than the contents of h_2 and the answer is no. If the former can not be done, we must try once more to send a virus from h'_2 to h to determine if the contents were or not equal.
  Notice that the contents of h'_1, h'_2 and h get modified, but this does not prevent the module to be reused, because the first two get initialized by the first two `COPY` modules and the latter plays a passive role.
- `ISZERO?`(h): predicate module that checks if the number of viruses in host h is zero.

This module is simply implemented by introducing an empty internal host h' and using the module AREEQUAL?(h, h').

Finally, notice that we can consider any Virus Machine Π as an action module without parameters, where the initial instruction is the *in* instruction and any instruction with out-degree zero is an *out* instruction. The only problem is that Π would be a module of one use, because it is not guaranteed that the contents of its hosts are the same before and after execution. If we wanted to reuse it, we would need to set Π to its initial state, by means of the following module:

– RESTART(Π): action module that sets the number of viruses of each host h_i of Π to its initial contents n_i.
  This module is implemented by the following concatenation of modules:

$$in \rightarrow \texttt{SET}(h_1, n_1) \rightarrow \cdots \rightarrow \texttt{SET}(h_p, n_p) \rightarrow out$$

where $h_1, \ldots, h_p$ are the hosts of Π and $n_1, \ldots, n_p$ are their initial contents.

4.3 Basic or Initial Functions

We begin by describing function computing Virus Machines that allow us to compute the basic functions.

– The *constant zero function*, $\mathcal{O} : \mathbb{N} \rightarrow \mathbb{N}$, defined by $\mathcal{O}(x) = 0$, for every $x \in \mathbb{N}$, can be computed by the following virus machine $\Pi_{\mathcal{O}}$ with input working in the computing mode:
  - The hosts are $H_{\mathcal{O}} = \{h, h_{\text{zero}}\}$, each of them initially empty.
  - The input host is h and the output host is h_{zero}.
  - The initial and only instruction is `halt`.
  - Each of the three graphs $D_{H_{\mathcal{O}}}$, $D_{I_{\mathcal{O}}}$ and $G_{C_{\mathcal{O}}}$ determining the functioning of the machine has an empty set of edges.

  This way, for any input the Virus Machine $\Pi_{\mathcal{O}}$ halts in the very first step, and the output host h_{zero} remains empty. So the output of this machine is always zero.
– The *successor function*, $\mathcal{S} : \mathbb{N} \rightarrow \mathbb{N}$, defined by $\mathcal{S}(x) = x + 1$, for every $x \in \mathbb{N}$, can be computed by the following virus machine $\Pi_{\mathcal{S}}$ with input working in the computing mode:
  - The hosts are $H_{\mathcal{S}} = \{h, h_{\text{one}}\}$, together with the internal hosts of the module ADD(h, h_{one}).
  - The initial contents are zero for the host h and one for the host h_{one}, together with the initial contents of the internal hosts of the module ADD(h, h_{one}).
  - The input host is h and the output host is h_{one}.
  - The instructions are $I_{\mathcal{S}} = \{\texttt{halt}\}$, together with the instructions of the module ADD(h, h_{one}).
  - The initial instruction is the *in* instruction of the module ADD(h, h_{one}).

- The functioning of the Virus Machine is given by the following sequence, which determines the graphs $D_{H_{\mathcal{S}}}$, $D_{I_{\mathcal{S}}}$ and $G_{C_{\mathcal{S}}}$:

$$\texttt{ADD}(h, h_{\text{one}}) \rightarrow \texttt{halt}$$

This way, for any input the Virus Machine $\Pi_{\mathcal{S}}$ adds it from h to h_{one} and halts. Since host h_{one} contained one virus, the output of this machine is equal to the input plus one, as required.

– The *projection functions*, $\Pi_i^m : \mathbb{N}^m \rightarrow \mathbb{N}$, with $m \geq 1$ and $1 \leq i \leq m$, defined by $\Pi_i^m(x_1, \dots, x_m) = x_i$, for every $(x_1, \dots, x_m) \in \mathbb{N}^m$, can be computed by the following Virus Machine $\Pi_{\Pi_i^m}$ with input working in the computing mode:
  - The hosts are $H_{\Pi_i^m} = \{h_1, \dots, h_m, h_{\text{out}}\}$, together with the internal hosts of the module $\texttt{COPY}(h_i,\ h_{\text{out}})$.
  - The initial contents are zero for the hosts $h_1, \dots, h_m, h_{\text{out}}$, together with the initial contents of the internal hosts of the module $\texttt{COPY}(h_i,\ h_{\text{out}})$.
  - The input hosts are $h_1, \dots, h_m$ and the output host is h_{out}.
  - The instructions are $I_{\Pi_i^m} = \{\texttt{halt}\}$, together with the instructions of the module $\texttt{COPY}(h_i,\ h_{\text{out}})$.
  - The initial instruction is the *in* instruction of the module $\texttt{COPY}(h_i,\ h_{\text{out}})$.
  - The functioning of the Virus Machine is given by the following sequence, which determines the graphs $D_{H_{\Pi_i^m}}$, $D_{I_{\Pi_i^m}}$ and $G_{C_{\Pi_i^m}}$:

$$\texttt{COPY}(h_i, h_{\text{out}}) \rightarrow \texttt{halt}$$

This way, for any input the Virus Machine $\Pi_{\Pi_i^m}$ copies the i-th component from h_i to h_{out} and halts, so the output of the machine is that component.

4.4 Composition of Functions

We show now how the composition of functions can be simulated by Virus Machines with input working in the computing mode.

Definition 5. *Let $f : \mathbb{N}^m \dashrightarrow \mathbb{N}$ and $g_1 : \mathbb{N}^n \dashrightarrow \mathbb{N}, \dots, g_m : \mathbb{N}^n \dashrightarrow \mathbb{N}$. Then, the composition of f with g_1 to g_m, denoted $C(f; g_1, \dots, g_m)$, is a partial function from $\mathbb{N}^n$ to $\mathbb{N}$ defined as follows:*

$$C(f; g_1, \dots, g_m)(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

for each $(x_1, \dots, x_n) \in \mathbb{N}^n$.

Let $\Pi_f, \Pi_{g_1}, \dots, \Pi_{g_m}$ be Virus Machines with input, computing the functions $f, g_1, \dots, g_m$, respectively. Let us assume that for each $x \in \{f, g_1, \dots, g_m\}$ the elements of the Virus Machine Π_x are the following:

– The hosts are $H_x = \{h_1^x, \dots, h_{p_x}^x\}$.
– The initial contents of the hosts are $n_1^x, \dots, n_{p_x}^x$.
– The input hosts are $h_1^f, \dots, h_m^f$ and $h_1^x, \dots, h_n^x$ for $x \in \{g_1, \dots, g_m\}$.
– The output host is h_{out}^x.

- The instructions are $I_x = \{i_1^x, \dots, i_{q_x}^x\}$.
- The initial instruction is i_{start}^x.
- The functioning of the Virus Machine is determined by the directed graphs D_{H_x}, D_{I_x} and the bipartite graph G_{C_x}.

Then, the composition of f with $g_1, \dots, g_m$ can be computed by the following Virus Machine $\Pi_{C(f;g_1,\dots,g_m)}$ with input:

- The hosts are $H = \{h_1, \dots, h_n\} \cup H_f \cup H_{g_1} \cup \dots \cup H_{g_m}$, together with the internal hosts of the modules.
- The initial contents of the hosts are
$$(0, \dots, 0, n_1^f, \dots, n_{p_f}^f, n_1^{g_1}, \dots, n_{p_{g_1}}^{g_1}, \dots, n_1^{g_m}, \dots, n_{p_{g_m}}^{g_m})$$
together with the initial contents of the internal hosts of the modules.
- The input hosts are $\{h_1, \dots, h_n\}$.
- The output host is h_{out}^f.
- The instructions are $I_f \cup I_{g_1} \cup \dots \cup I_{g_m} \cup \{\texttt{halt}\}$, together with the individualized instructions of the modules.
- The initial instruction is the *in* instruction of the first module.
- The functioning of the Virus Machine is given by the following sequence of concatenated modules, which determines the graphs D_H, D_I and G_C:
  1. First we simulate the introduction of the input into the input hosts of Π_{g_1}. Recall that the module `ADD(`h_1, h_2`)` does not change the content of host h_1.
  $$\texttt{ADD}(h_1, h_1^{g_1}) \rightarrow \dots \rightarrow \texttt{ADD}(h_n, h_n^{g_1}) \rightarrow$$
  2. We do the same for the machines $\Pi_{g_2}, \dots, \Pi_{g_m}$.
  $$\rightarrow \texttt{ADD}(h_1, h_1^{g_2}) \rightarrow \dots \rightarrow \texttt{ADD}(h_n, h_n^{g_2}) \rightarrow$$
  $$\vdots$$
  $$\rightarrow \texttt{ADD}(h_1, h_1^{g_m}) \rightarrow \dots \rightarrow \texttt{ADD}(h_n, h_n^{g_m}) \rightarrow$$
  3. Now we can simulate the functions $g_1, \dots, g_m$ over the received input.
  $$\rightarrow \Pi_{g_1} \rightarrow \dots \rightarrow \Pi_{g_m} \rightarrow$$
  4. Finally, we introduce the outputs of the previous simulations as input for Π_f, simulate f and finish the execution.
  $$\rightarrow \texttt{ADD}(h_{\text{out}}^{g_1}, h_1^f) \rightarrow \dots \rightarrow \texttt{ADD}(h_{\text{out}}^{g_m}, h_m^f) \rightarrow \Pi_f \rightarrow \texttt{halt}$$

4.5 Primitive Recursion of Functions

We show now how the primitive recursion of functions can be simulated by Virus Machines with input working in the computing mode.

Definition 6. *Let $f : \mathbb{N}^m \dashrightarrow \mathbb{N}$ and $g : \mathbb{N}^{m+2} \dashrightarrow \mathbb{N}$. Then, the function obtained by primitive recursion from f and g, denoted $Rec(f;g)$, is a partial function from $\mathbb{N}^{m+1}$ to $\mathbb{N}$ defined as follows:*

$$Rec(f;g)(x_1,\dots,x_m,x_{m+1}) = \begin{cases} f(x_1,\dots,x_m), & \textit{if } x_{m+1}=0 \\ g(x_1,\dots,x_m,x_{m+1},y), & \textit{otherwise} \end{cases}$$

$$\textit{where } y = Rec(f;g)(x_1,\dots,x_m,x_{m+1}-1)$$

for each $(x_1,\dots,x_m,x_{m+1}) \in \mathbb{N}^{m+1}$.

Let Π_f and Π_g be Virus Machines with input, computing the functions f and g, respectively. Let us suppose that for each function $x \in \{f,g\}$ the elements of the virus machine Π_x are the following:

- The hosts are $H_x = \{h_1^x,\dots,h_{p_x}^x\}$.
- The initial contents of the hosts are $(n_1^x,\dots,n_{p_x}^x)$.
- The input hosts are $h_1^f,\dots,h_m^f$ and $h_1^g,\dots,h_{m+2}^g$.
- The output host is h_{out}^x.
- The instructions are $I_x = \{i_1^x,\dots,i_{q_x}^x\}$.
- The initial instruction is i_{start}^x.
- The functioning of the Virus Machine is determined by the directed graphs D_{H_x}, D_{I_x} and the bipartite graph G_{C_x}.

Then, the function $Rec(f;g)$ can be computed by the following Virus Machine with input $\Pi_{Rec(f;g)}$:

- The hosts are $H = \{h_1,\dots,h_{m+1},h',h_{\text{one}},h_{\text{out}},h'_{\text{out}}\} \cup H_f \cup H_g$, together with the internal hosts of the modules.
- The initial contents of the hosts are $0,\dots,0,0,1,0,0,n_1^f,\dots,n_{p_f}^f,n_1^g,\dots,n_{p_g}^g$, together with the initial contents of the internal hosts of the modules.
- The input hosts are $\{h_1,\dots,h_{m+1}\}$.
- The output host is h_{out}.
- The instructions are $I_f \cup I_g \cup \{\texttt{halt}\}$, together with the individualized instructions of the modules.
- The initial instruction is the *in* instruction of the first module.
- The functioning of the Virus Machine is given by the following sequence of concatenated modules, which determines the graphs D_H, D_I and G_C:
  1. Observe that to compute the function $Rec(f;g)$ we have to repeatedly compute the function g as many times as indicated by the $(m+1)$-th argument, except for the first time in which the function f has to be computed instead.
  2. First we simulate the introduction of the input for the function f into the input hosts of Π_f.

$$\texttt{ADD}(h_1,h_1^f) \rightarrow \cdots \rightarrow \texttt{ADD}(h_m,h_m^f) \rightarrow$$

3. We now simulate the function f over its input and copy the result to h_{out} and h'_{out}. This is because if we are done, then the result has to be in h_{out}, but if we are not done, we must pass this result as the last argument to g. However, h_{out} is required to have out-degree zero, so we take the result from h'_{out} instead.

$$\to \mathit{\Pi}_f \to \texttt{COPY}(h^f_{\text{out}}, h_{\text{out}}) \to \texttt{COPY}(h^f_{\text{out}}, h'_{\text{out}}) \to$$

4. We check if we are done, in which case stop the execution.

$$\begin{array}{l} \to \texttt{AREEQUAL?}(h_{m+1}, h') \overset{yes}{\to} \texttt{halt} \\ \downarrow no \end{array}$$

5. If we are not done, one computation of g has to be simulated. For that, the $(m+1)$-th argument of g is updated by adding 1 to it and the appropriate input is introduced into the input hosts of $\mathit{\Pi}_g$. The input for the last argument is the result of the previous computation, that we will ensure is always within host h'_{out}.

$$\overset{no}{\to} \texttt{ADD}(h_{\text{one}}, h') \to \texttt{ADD}(h_1, h^g_1) \to \cdots \to \texttt{ADD}(h_m, h^g_m) \to$$
$$\texttt{ADD}(h', h^g_{m+1}) \to \texttt{ADD}(h'_{\text{out}}, h^g_{m+2}) \to$$

6. We simulate the function g and copy the result to both hosts h_{out} and h'_{out}. Before continuing to step 4, the machine $\mathit{\Pi}_g$ has to be restarted to its initial state, so that it can be used to simulate again the function g, if necessary.

$$\to \mathit{\Pi}_g \to \texttt{COPY}(h^g_{\text{out}}, h_{\text{out}}) \to \texttt{COPY}(h^g_{\text{out}}, h'_{\text{out}}) \to$$
$$\texttt{RESTART}(\mathit{\Pi}_g) \to \text{back to step 4}$$

4.6 Unbounded Minimization of Functions

We show now how the unbounded minimization of functions can be simulated by Virus Machines with input working in the computing mode.

Definition 7. *Let $f : \mathbb{N}^{m+1} \dashrightarrow \mathbb{N}$. Then, the function obtained by unbounded minimization from f, denoted $Min(f)$, is a partial function from $\mathbb{N}^m$ to $\mathbb{N}$ defined as follows:*

$$Min(f)(x_1, \ldots, x_m) = \begin{cases} y_{x_1,\ldots,x_m}, & \textit{if it exists} \\ \textit{undefined}, & \textit{otherwise} \end{cases}$$

where

$$y_{x_1,\ldots,x_m} = \min\{y \in \mathbb{N} \mid \forall z < y \left(f \textit{ is defined over } (x_1, \ldots, x_m, z)\right) \wedge f(x_1, \ldots, x_m, y) = 0\}$$

for each $(x_1, \ldots, x_m) \in \mathbb{N}^m$.

Let Π_f be a Virus Machine with input, computing the function f. Let us suppose that the elements of the Virus Machine Π_f are the following:

- The hosts are $H_f = \{h_1^f, \dots, h_{p_f}^f\}$.
- The initial contents of the hosts are $n_1^f, \dots, n_{p_f}^f$.
- The input hosts are $h_1^f, \dots, h_{m+1}^f$.
- The output host is h_{out}^f.
- The instructions are $I_f = \{i_1^f, \dots, i_{q_f}^f\}$.
- The initial instruction is i_{start}^f.
- The functioning of the Virus Machine is determined by the directed graphs D_{H_f}, D_{I_f} and the bipartite graph G_{C_f}.

Then, the function $Min(f)$ can be computed by the following Virus Machine with input $\Pi_{Min(f)}$:

- The hosts are $H = \{h_1, \dots, h_m, h_{m+1}, h_{\text{one}}, h_{\text{out}}\} \cup H_f$, together with the internal hosts of the modules.
- The initial contents of the hosts are $0, \dots, 0, 0, 1, 0, n_1^f, \dots, n_{p_f}^f$, together with the initial contents of the internal hosts of the modules.
- The input hosts are $\{h_1, \dots, h_m\}$.
- The output host is h_{out}.
- The instructions are $I_f \cup \{\texttt{halt}\}$, together with the individualized instructions of the modules.
- The initial instruction is the *in* instruction of the first module.
- The functioning of the Virus Machine is given by the following sequence of concatenated modules, which determines the graphs D_H, D_I and G_C:
  1. Observe that to compute the function $Min(f)$ we have to repeatedly compute the function f until we obtain a zero result.
  2. First we simulate the introduction of the input for the function f into the input hosts of Π_f.

$$\texttt{ADD}(h_1, h_1^f) \to \cdots \to \texttt{ADD}(h_{m+1}, h_{m+1}^f) \to$$

  3. We now simulate the function f over its input and check if the result is or not zero.

$$\begin{array}{c} \to \Pi_f \to \texttt{ISZERO?}(h_{\text{out}}^f) \overset{yes}{\to} \\ \downarrow no \end{array}$$

  4. In the case that the result obtained is zero, we copy the last argument to the output host and stop the execution.

$$\overset{yes}{\to} \texttt{COPY}(h_{m+1}, h_{\text{out}}) \to \texttt{halt}$$

  5. Otherwise, we add one to the last argument, restart the machine Π_f so that it can be used again to simulate f, and go back to step 2.

$$\overset{no}{\to} \texttt{ADD}(h_{one}, h_{m+1}) \to \texttt{RESTART}(\Pi_f) \to \text{back to step 2}$$

4.7 Main Result

Taking into account that the class of partial recursive functions coincides with the least class that contains the basic functions and is closed under composition, primitive recursion and unbounded minimization (see [2]), it is guaranteed that it is possible to construct virus machines that compute any partial recursive function. Then, we have the following result.

Theorem 1. *The family $NVM(*, *, *)$ equals to the family of all the recursively enumerable sets of natural numbers.*

5 Conclusions and Future Work

Virus Machines are a bio-inspired computational paradigm based on the transmissions and replications of viruses [1]. The computational completeness of Virus Machines having no restriction on the number of hosts, the number of instructions and the number of viruses contained in any host along any computation has been established by simulating register machines. However, when an upper bound on the number of viruses present in any host during a computation is set, the computational power of these systems decreases; in fact, a characterization of semi-linear sets of numbers is obtained [1].

The semantics of the model makes it easy to construct specific Virus Machines by assembling small components that carry out a part of the task to be solved. It is then convenient to develop a library of modules solving common problems such as comparisons or arithmetic operations between contents of hosts.

In this paper, Virus Machines able to compute partial functions on natural numbers are introduced. The universality of non-restricted Virus Machines is then proved by showing that they can compute all partial recursive functions.

In [5] Virus Machines working in the generating mode are considered, and it is shown how they can generate any diophantine set, providing, via the MRDP theorem, another proof of the universality of this model of computation. What is interesting is that the structure of the design of these systems has served as inspiration to defined a parallel variant of Virus Machines having several independent instruction transfer networks. It could be interesting to explore other means of introducing parallelism, such as considering more than one type of viruses or allowing more than one virus to be transmitted when a channel is opened.

To study the computational efficiency of this model of computation, for example to analyze if the parallel variants of Virus Machines represent an improvement over the sequential one, a computational complexity theory is required. This way, the resources needed to solve (hard) problems can be rigorously measured.

Acknowledgments. This work was supported by Project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain, cofinanced by FEDER funds.

References

1. Chen, X., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Wang, B., Zeng, X.: Computing with viruses. Int. J. Bioinspired Comput. (2015, submitted)
2. Cohen, D.E.: Computability and Logic. Ellis Horwood, Chichester (1987)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: An Introduction to Algorithms. The MIT Press, Cambridge, Massachussets (1994)
4. Dimmock, N.J., Easton, A.J., Leppard, K.: Introduction to Modern Virology. Blackwell Publishing, Malden (USA) (2007)
5. Romero-Jiménez, Á., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: Sequential and parallel generation of diophantine sets by virus machines. J. Comput. Theor. Nanosci. (2015, submitted)
6. Rozenberg, G., Bäck, T., Kok, J.N.: Handbook of Natural Computing, 1st edn. Springer, Heidelberg (2012)