Generating Diophantine Sets by Virus Machines

Álvaro Romero-Jiménez, Luis Valencia-Cabrera,
and Mario J. Pérez-Jiménez

Research Group on Natural Computing Department of Computer Science
and Artificial Intelligence, University of Seville,
Avda. Reina Mercedes s/n, 41012 Seville, Spain
{romero.alvaro,lvalencia,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 set generating devices are considered. Then, the universality of non-restricted virus machines is proved by showing that they can compute all diophantine sets, which the MRDP theorem proves that coincide with the recursively enumerable sets.

Keywords: Virus machines · Computational completeness · Diophantine sets · MRDP theorem

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 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, new virus machines as set generating devices are introduced. The universality of non-restricted virus machines working in this mode is proved by showing that they can generate all diophantine sets. The celebrated MRDP theorem assures that these sets are exactly the same as the recursively enumerable sets [4].

This paper is structured as follows. First, the computing model of virus machines is formally defined. Then, in Sect. 3 the computational completeness of non-restricted virus machines is stated. Finally, in Sect. 4 the main conclusions of this work are summarized and some suggestions for possible lines of future research are outlined.

2 Virus Machines

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

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 [2] 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.

Definition 1. *A* Virus Machine Π *of degree* (p, q)*, with* $p \geq 1, q \geq 1$*, is a tuple* $(\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{\text{out}})$*, where:*

- $\Gamma = \{v\}$ *is the singleton alphabet;*
- $H = \{h_1, \ldots, h_p\}$ *and* $I = \{i_1, \ldots, i_q\}$ *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* $h \in H$*,* 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* $i_1 \in I$;
- $h_{\text{out}} \notin I \cup \{v\}$ *and* h_{out} *is denoted by* h_0 *in the case that* $h_{\text{out}} \notin H$.

A Virus Machine $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{\text{out}})$ of degree (p, q) can be viewed as an ordered set of p *hosts* labelled with $h_1, \ldots, h_p$ (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 $i_1, \ldots, i_q$. The 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 $(a_{1,t}, \ldots, a_{p,t}, u_t, e_t)$, where $a_{1,t}, \ldots, a_{p,t}$ and e_t are non-negative integers and $u_t \in I \cup \{\#\}$, with $\# \notin H \cup \{h_0\} \cup I$. 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 $\mathcal{C}_0 = (n_1, \ldots, n_p, i_1, 0)$.

A configuration $\mathcal{C}_t = (a_{1,t}, \ldots, a_{p,t}, u_t, e_t)$ is a *halting configuration* if and only if u_t is the object $\#$. A non-halting configuration $\mathcal{C}_t = (a_{1,t}, \ldots, a_{p,t}, u_t, e_t)$ yields configuration $\mathcal{C}_{t+1} = (a_{1,t+1}, \ldots, a_{p,t+1}, u_{t+1}, e_{t+1})$ 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 virus.

3. Let us assume that instruction u_t is not attached to any channel $(h_s, h_{s'})$. Then there is no transmission of virus.
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.

In this paper we consider virus machines working in the *generating mode*. That is, we think of the result of a computation of a virus machine Π as the total number n of viruses sent to the output region during the computation. We say that $A \subseteq \mathbb{N}$ is the set *generated* by Π if it is verified that $n \in A$ if and only if there exists a halting computation of Π that outputs n.

3 The Universality of Non-Restricted Virus Machines

A *non-restricted Virus Machine* is a virus machine for which 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 $\mathbb{N}$ generated 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 parameters p, q, n is not bounded, then it is replaced with $*$. In particular, $NVM(*, *, *)$ denotes the family of all subsets of natural numbers generated by non-restricted virus machines.

3.1 Generating Diophantine Sets by Virus Machines

In this section, the computational completeness of non-restricted virus machines working in the generating mode is established. Specifically, we prove that they can generate all diophantine sets of natural numbers. Indeed, we will design non-restricted virus machines that, given a polynomial $P(x, y_1, \ldots, y_k)$ with integer coefficients:

1. Generate, in a non-deterministic manner, any tuple $(x, y_1, \ldots, y_k)$ of natural numbers.
2. Compute the value of P over the tuple $(x, y_1, \ldots, y_k)$.
3. If the computed value is zero, then halt and output x.
4. If the computed value is non-zero, then do not halt.

3.2 Modules

In order to ease the design of the virus machines generating any diophantine set, 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 out-degree less than two, so that they can still be connected to another instruction. In 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.

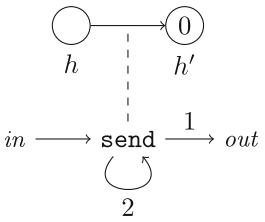
The layout of a module must be carefully done to avoid conflicts with other modules and to allow the module to be executed any number of times. To achieve the first condition, we will consider that all the hosts (with the only exception of the parameters of the module) and instructions of a module are individualized for that module, being distinct from the ones of any other module or virus machine. There are several ways to meet the second condition: for example, we can 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.

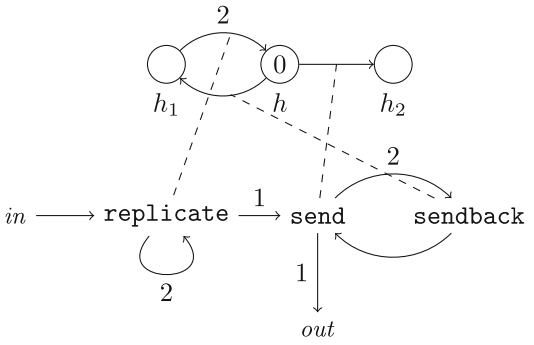
The library of modules used in this paper consists of the following modules (we name the action modules as verbs and the predicate modules as questions):

– `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.



– $\texttt{ADD}(h_1, h_2)$: 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.

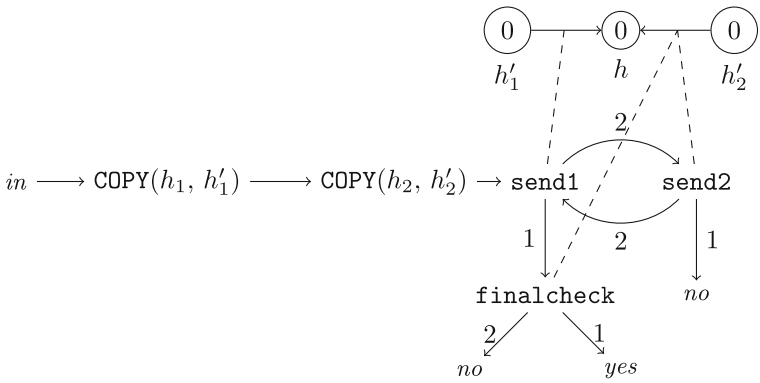
– $\texttt{COPY}(h_1, h_2)$: action module that sets the number of viruses in h_2 the same as in h_1, without modifying the number of viruses in h_1.
This module is implemented by the following concatenation of modules:

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

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 $\texttt{ADD}(h_1, h_2)$ does not modify the number of viruses in h_1, what will be important later.

– $\texttt{SET}(h, n)$: 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 $\texttt{COPY}(h', h)$.

– 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.
– MULTIPLY(h_1, h_2): action module that multiplies the number of viruses in host h_2 by the number of viruses in host h_1, without modifying the number of viruses in h_1.

This module is implemented in two stages:

1. An initialization stage, where the contents of an internal host h'_1, which will be used as a counter, is set to zero. Also the number of viruses in h_2 is saved in an internal host h'_2. This is because host h_2 needs to be emptied, so that it can be used as the accumulator in a standard implementation of the multiplication.

$$in \rightarrow \texttt{EMPTY}(h'_1) \rightarrow \texttt{COPY}(h_2, h'_2) \rightarrow \texttt{EMPTY}(h_2) \rightarrow$$

2. The second stage iteratively adds the contents of h'_2 to h_2, until the counter h'_1 reaches the number of viruses in h_1. The counter is incremented in each step by adding to it the contents of an internal host h_{one} that has only one virus within.

$$\rightarrow \texttt{AREEQUAL?}(h_1, h'_1) \overset{no}{\rightarrow} \texttt{ADD}(h'_2, h_2) \rightarrow \texttt{ADD}(h_{\text{one}}, h'_1) \rightarrow \text{back to stage 2}$$
$$\downarrow yes$$
$$out$$

It is clear that when the module ends, the host h_1 retains its initial number of viruses and the host h_2 has a number of viruses equal to the product of the initial number of viruses in h_1 and h_2. The internal host h_{one} is never modified and both internal hosts h'_1 and h'_2 are initialized in stage 1, what allows the module to be reused.

– $\texttt{RAISE}(h_1, h_2)$: action module that raises the number of viruses in host h_2 to the power of the number of viruses in host h_1, without modifying the number of viruses in h_1.
This module is implemented in two stages:
1. An initialization stage, where the contents of an internal host h'_1, which will be used as a counter, is set to zero. Also the number of viruses in h_2 is saved in an internal host h'_2. This is because the contents of host h_2 needs to be set to one virus, so that it can be used as the accumulator in a standard implementation of the exponentiation.

$$in \rightarrow \texttt{EMPTY}(h'_1) \rightarrow \texttt{COPY}(h_2, h'_2) \rightarrow \texttt{SET}(h_2, 1) \rightarrow$$

2. The second stage iteratively multiplies the contents of h_2 by the contents of h'_2, until the counter h'_1 reaches the number of viruses in h_1. The counter is incremented in each step by adding to it the contents of an internal host h_{one} that has only one virus within.

$$\rightarrow \texttt{AREEQUAL?}(h_1, h'_1) \overset{no}{\rightarrow} \texttt{MULTIPLY}(h'_2, h_2) \rightarrow \texttt{ADD}(h_{\text{one}}, h'_1) \rightarrow \begin{matrix}\text{back to}\\ \text{stage 2}\end{matrix}$$
$$\downarrow yes$$
$$out$$

It is clear that when the module ends, the host h_1 retains its initial number of viruses and the host h_2 has a number of viruses equal to the initial number of viruses in h_2 raised to the initial number of viruses in h_1. The internal host h_{one} is never modified and both internal hosts h'_1 and h'_2 are initialized in stage 1, what allows the module to be reused.

– $\texttt{EXPT}(h, n)$: action module that raises the number of viruses in h to the power of n.
This module is implemented simply by introducing an internal host h' with initial number of viruses n and using the module $\texttt{RAISE}(h', h)$.

3.3 Generation of a Diophantine Set

In what follows we show how to design, given a polynomial $P(x, y_1, \ldots y_k)$ with integer coefficients, a virus machine Π_P that generates the diophantine set characterized by that polynomial.

– The hosts are

$$\begin{aligned} H = &\{h_x, h'_x, h_{y_1}, h'_{y_1}, \ldots, h_{y_k}, h'_{y_k}, h_+, h_-, h_{\text{one}}, h_{\text{out}}\} \cup \\ &\{h_c \,|\, c > 0 \text{ and there exists } \alpha, \beta_1, \ldots, \beta_k \in \mathbb{N} \text{ such that} \\ &\qquad c\, x^\alpha y_1^{\beta_1} \cdots y_k^{\beta_k} \text{ or } -c\, x^\alpha y_1^{\beta_1} \cdots y_k^{\beta_k} \text{ is a monomial of } P\} \end{aligned}$$

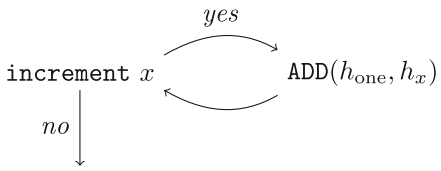
together with the internal hosts of the modules.

– The initial contents of h_{one} is 1, and of h_c is c. The initial contents of the rest of hosts is 0, except for the internal hosts of the modules, which have their specific initial contents.
– The output host is h_{out}.
– The instructions are

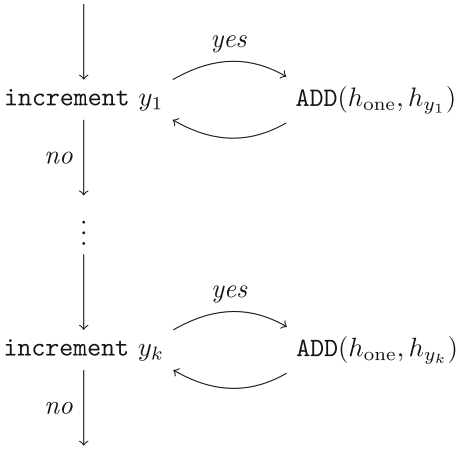
$$I = \{\texttt{increment}\, x, \texttt{increment}\, y_1, \ldots, \texttt{increment}\, y_k, \\ \texttt{halt}, \texttt{infinite loop}\}$$

together with the individualized instructions of the modules.

– The initial instruction is `increment` x.
– The functioning of the virus machine is given by the following sequence of concatenated instructions and modules, which determines the graphs D_H, D_I and G_C:
  1. First a value for x is generated, in a non-deterministic manner.



  The instruction transfer paths labelled by *yes* and *no* are set to have the same weight (for example, weight 1) so, according to the semantics of the model, it is non-deterministically chosen to add or not the contents of h_{one}, one virus, to h_x. In the former case, the machine comes back to instruction `increment x` to make the choice again. In the latter case, it has finished generating a value for x.
  2. Analogously, a value for each of y_1 to y_k is generated in a non-deterministic manner.



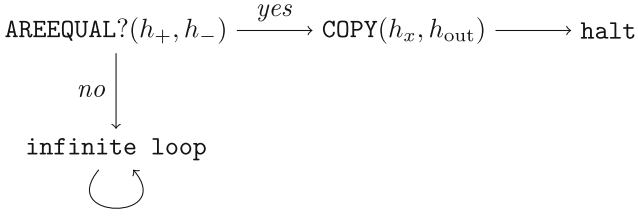
3. For each monomial $c\,x^{\alpha}y_1^{\beta_1}\cdots y_k^{\beta_k}$ of P, with $c > 0$, its value over the arguments $x, y_1, \ldots, y_k$ previously generated is computed and accumulated in h_+.

$$\begin{aligned}&\rightarrow \texttt{COPY}(h_x, h'_x) \rightarrow \texttt{COPY}(h_{y_1}, h'_{y_1}) \rightarrow \cdots \rightarrow \texttt{COPY}(h_{y_k}, h'_{y_k}) \rightarrow\\ &\texttt{EXPT}(h'_x, \alpha) \rightarrow \texttt{EXPT}(h'_{y_1}, \beta_1) \rightarrow \cdots \rightarrow \texttt{EXPT}(h'_{y_k}, \beta_k) \rightarrow\\ &\texttt{MULTIPLY}(h'_{y_1}, h'_x) \rightarrow \cdots \rightarrow \texttt{MULTIPLY}(h'_{y_k}, h'_x) \rightarrow \texttt{MULTIPLY}(h_c, h'_x) \rightarrow\\ &\texttt{ADD}(h'_x, h_+) \rightarrow\end{aligned}$$

4. For each monomial $c\,x^{\alpha}y_1^{\beta_1}\cdots y_k^{\beta_k}$ of P, with $c < 0$, its absolute value over the arguments $x, y_1, \ldots, y_k$ previously generated is computed and accumulated in h_-.

$$\begin{aligned}&\rightarrow \texttt{COPY}(h_x, h'_x) \rightarrow \texttt{COPY}(h_{y_1}, h'_{y_1}) \rightarrow \cdots \rightarrow \texttt{COPY}(h_{y_k}, h'_{y_k}) \rightarrow\\ &\texttt{EXPT}(h'_x, \alpha) \rightarrow \texttt{EXPT}(h'_{y_1}, \beta_1) \rightarrow \cdots \rightarrow \texttt{EXPT}(h'_{y_k}, \beta_k) \rightarrow\\ &\texttt{MULTIPLY}(h'_{y_1}, h'_x) \rightarrow \cdots \rightarrow \texttt{MULTIPLY}(h'_{y_k}, h'_x) \rightarrow \texttt{MULTIPLY}(h_{|c|}, h'_x) \rightarrow\\ &\texttt{ADD}(h'_x, h_-) \rightarrow\end{aligned}$$

5. Finally, in order to check if the arguments $x, y_1, \ldots, y_k$ constitute a solution of the polynomial P, the contents of h_+ and h_- are compared. If they are equal, the value of the argument x is copied from h_x to h_{out} and the computation halts. Otherwise, an infinite loop is started to make the computation non-halting.



3.4 Main Result

Taking into account that, by virtue of the MRDP theorem every recursively enumerable set is diophantine, it is guaranteed that it is possible to construct virus machines that compute any such set. Then, we have the following result.

Theorem 1. *NVM(*, *, *) = NRE.*

4 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, a new variant of virus machines able to generate sets of natural numbers is introduced. The universality of non-restricted virus machines is then proved by showing that they can generate all diophantine sets.

Being shown the computational completeness of virus machines (in the unrestricted form) working in several modes, we can turn our attention to their computational efficiency. A computational complexity theory for these devices is therefore required, so that the resources needed to solve (hard) problems can be rigorously measured.

To this respect, it is convenient to point out that, according to the formalization given in this paper, virus machines are inherently sequential devices. To increase their efficiency it could be interesting to consider variants of the model where the instructions are activated in parallel.

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. International Journal of Bioinspired Computation, submitted, **7**(3) 176-182 (Submitted 2015)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: An Introduction to Algorithms. MIT Press, Cambridge (1994)
3. Dimmock, N.J., Easton, A.J., Leppard, K.: Introduction to Modern Virology, 6th edn. Blackwell Publishing, Malden (2007)
4. Matijasevich, Y.: Hilbert's Tenth Problem. MIT Press, Cambridge (1993)
5. Rozenberg, G., Bäck, T., Kok, J.N.: Handbook of Natural Computing. Springer, Heidelberg (2012)