Fundamenta Informaticae XXI (2001) 1001–1015
IOS Press

On the computational power of querying the history

Alexei Lisitsa
Computer Science Department
University of Liverpool
Ashton Building, Ashton Street
Liverpool, L69 3BX, UK
Lisitsa@liverpool.ac.uk

Igor Potapov
Computer Science Department
University of Liverpool
Ashton Building, Ashton Street
Liverpool, L69 3BX, UK
Potapov@liverpool.ac.uk

Abstract. Querying its own history is an important mechanism in the computations, especially those interacting with people or other computations such as transaction processing, electronic data interchange. John McCarthy in his Elephant programming language proposal suggested exploiting the referring to the past as the main programming primitive. In this paper we study the computational power of such primitive. In order to do that we propose a refined formal model, History Dependent Machine (HDM), which uses querying the history as its sole computational primitive. Our main result may be spelled in general terms as: a model with a single agent wandering around a pool of resources and having ability to check its own history for simple temporal properties has a universal computational power. Moreover, HDM can simulate any multicounter machine in real time. Then we show that the computations of HDM may be specified in the extension of propositional linear temporal logic by flexible constants, the abstraction operator and equality. We use then universality of HDM model to show that the above extension with a single flexible constant is not recursively axiomatizable.

Keywords: History-dependent computations, executable temporal logic, models of computations, universality.

Address for correspondence: Computer Science Department, University of Liverpool, Ashton Building, Ashton Street, L69 3BX, UK

1. Introduction

Computation is often considered as a process generating a sequence of *states*. In very general terms such a process can be described as follows. Starting with an initial state s_0 deterministic computation proceeds by application of a *transition function* $\sigma : S \to S$, where S is a set of possible states, and generates a sequence of states (which can be called a history of computation) by taking $s_{i+1} = \sigma(s_i)$. One may have either a process generating an infinite sequence, or the process may finish in one of the *terminal* states $t \in T \subseteq S$. Obviously this description is very general and abstracts away the nature of states and transition functions. But it captures the idea that the next state in the history of computation *depends solely* on the current state. For some computations however, it is more natural to consider the generation of the next state (say, s_{n+1}) as depending not only on the current state s_n but on the whole history up to the moment n that is $s_0, s_1, \ldots, s_n$. In that case one may assume a transition function $\sigma_h : S^* \to S$ where S^* is the set of all finite sequences of the elements of S. In a sense, in that case the computation refers to its own past to generate the next state. John McCarthy in his proposal of Elephant programming language [11] suggested to exploit the referring to the past as the main programming primitive, especially suitable for implementing transaction processing and electronic data interchange[11]:

> *Elephant programs may not need data structures, because program statements can refer directly to past. Thus, a program can say that an airline passenger has a reservation if he has made one and hasn't cancelled it. . . . Algolic programs refer to the past via variables, arrays and other data structures. Elephant 2000 programs can refer to the past directly.*

Theoretically, referring to the past does not increase computational power of sufficiently rich computational models for one can implement it just by keeping the records of all relevant information about the past in the current state (in the *single virtual history list or "journal"* in terms of [11]). This observation leaves, however the question on computational power of such primitive itself open. To address such a question we propose the model of history dependent computations with querying the history (or referring to the past) as its *sole* primitive and demonstrate its computational universality. In a sense, it supports McCarthy's assumption that referring to the past is a powerful facility which can be used as the main programming primitive. This support is still partial because we consider in this paper only non-interactive sequential computations of history dependent machines (HDMs), leaving the investigation of behavioural properties and expressive power of interactive history dependent machines to further research.

Informally, the computations of history dependent machines may be described as follows: being in some state the machine checks the history of computation up to the current moment, and, subject to the conditions satisfied in the history, either generates new state in which it never been before, or it returns to the one of the states it has been before and which has been *singled out* by some condition. We show that very restricted set of temporal queries is enough to make the model universal by using not-trivial encoding of multidimensional structure into several classes of time slices.

Our HDM model, history-dependent machines although being in part motivated by Elephant proposal, is more general and abstract and has nothing to do, for example, with the theory of Speech Acts on which Elephant is based. To keep the model as general as possible we do not assume any structure on the states and rely only on the ability to distinguish between different states and on detection of the order in which they appear in the history.

Related models of computation with similar motivations have appeared in the area of executable temporal logics, and in particular, in METATEM paradigm[8]. In METATEM the temporal formulae are given an operational semantics and considered as the programs themselves, operating according to the principle "declarative past implies imperative future". Although executable temporal logic approach and our history-dependent machines are close to each other, at least at the ideological level, we do not do detailed technical comparison in this paper. Instead, in the Section 4 we demonstrate connections with temporal logic, further development of which would lead to clarification of relationships between both approaches.

Aspects of history depended computations appeared also in the research on temporal and active databases. Both checking temporal integrity conditions [4] and maintaining triggers [14] involve computations referring to the (finite) history of a database. To show that *potential satisfiability* of integrity contraints is undecidable, authors of [4] demonstrate how to encode Turing machine computations in a fragment of first-order temporal logic. On the one hand that fragment is more powerful that our $LTL_{\lambda=}$ because it uses standard quantifiers. On the other hand in the encoding from [4], unlike in our modelling, only referring to the *immediate* past is needed. One can see it as the indication of trade-off between power of history querying primitives and non-temporal primitives nedeed to get Turing complete computational models.

Maintaining temporal triggers in active databases involve computations which check some conditions on the database history and fire the triggers. In [14] temporal logic languages for specifying temporal triggers are proposed. The only quantifier allowed in these languages is the *assignment* operator, whose semantics is the same as of freeze quantifier or, indeed, of λ-abstraction operator [6, 10]. The expressive and computational power of the languages from [14] depends on underlying database query languages and assumed built-in domains. We notice that our HDM model can be specified in a small fragment of a trigger specification language from [14] modified to allow both past and future time modalities.

The paper is organised as follows. In the next section we introduce our History Dependent Machine model and discuss possible variations of the model. The main result of the paper is presented in Section 3, where we demonstrated a universality of HDM model. Section 4 introduces a fragment of temporal logic suitable for specification of HDM computations and, as application of the main result, we prove its non-axiomatizability. We conclude the paper with the Section 5.

2. History Dependent Machines

Before we give the definition of our History Dependent Machine model we consider less formal, but still idealized scenario which gives an intuition behind the model. We found it intuitive to think about history dependent computations as a behaviour of a one-pebble system. Let S be a set of states. Any evolution of a system over states from S defines a function $s : \mathbb{N} \to S$. If at some moment of time t, the state of a system M is r, i.e. $s(t) = r$ we interpret it as a pebble being put on the element r of S. Thus, the whole evolution of M may be though of as the movement of a single pebble over the abstract set S. Such metaphor gives the very abstract model of computational process (agent) using some resources. In such a model a pebble may be thought of as a computational process and elements of the domain S as the abstract resources. Then, if at some moment of time a pebble is on an element x of the domain, one may understand it as *"process (agent) uses the resource x"*. The model can be easily extended to the case of several processes possibly using several resources at the same time. Some relevant discussion can be found in [1, 9, 10].

Our main result may be spelled in these terms as: even a model with a *single* agent wandering around a pool of resources and having ability to check its own path for simple temporal properties has a universal computational power, and is able to model multicounter machines in *real time*. From the results of [10] one may easily extract a similar result but for the case of several agents.

Now we introduce a formal model of History Dependent Machine. Let S be an arbitrary set, which in general may be infinite and elements of which we call *states*. A *history* over S is a finite sequence of states $s_0, \dots, s_n$ from S with the last state being labelled by a constant now. Formally, we represent the history h as a first-order two-sorted structure

$$\langle T, S, s, \leq, next, 0, now \rangle$$

where T is a *finite* sort of moments of time; S is a sort of states; $s : T \to S$ is *state* function; $\leq$ is a linear order on T; 0 and now are constants interpreted by minimal, respectively, maximal element of T wrt $\leq$; $next : T \to T$ is the successor (partial) function giving a next element of T wrt $\leq$ and $next(now)$ is undefined. The set of all histories over S we denote by $\mathcal{H}(S)$.

The language of terms to encode some formulae satisfied at a single state of the history uses two new functional symbols $first : T \to T$ and $last : T \to T$. We also define a *term* of the language as follows: the constant now is a term and if t is a term then $first(t)$ and $last(t)$ are terms. Two functional symbols are interpreted in a history $h = \langle T, S, \leq, next, 0, now \rangle$ as follows:

- - if $s(t) = s(0)$ then $first(t) = 0$ for all t $(0 \leq t \leq now)$;
  - if $s(t) \neq s(0)$ then $first(t) = t'$, such that $s(next(t')) = s(t) \wedge \forall z < t'\ s(next(z)) \neq s(t)$ for all t $(0 < t \leq now)$;
- - $last(0) = 0$;
  - $last(t) = t'$, such that $next(t') = t$ for all t $(0 \leq t \leq now)$.

Thus, for a moment of time y from a history, $first(y)$ and $last(y)$ are referring to the moments (in the past wrt y) when the machine made a transition to the state $s(y)$ for the *first*,respectively, *last* time. The interpretation of $first$ and $last$ is extended to the interpretation of all terms in a usual way.

A *rule* of a History Dependent Machine is any expression of one of two possible forms:

- $\varphi \to move(new)$
- $\psi \to move(s(t))$

where t is a term, φ and ψ are arbitrary conjunctions of the atomic formulae of one of the following forms: $s(t_i) = s(t_j)$ or $s(t_i) \neq s(t_j)$, where t_i and t_j are terms.

Definition 2.1. Let R be a rule and h is a history. The rule $R = \varphi \to move(\_)$ is *applicable* to h iff $h \models \varphi$.

Let R be a rule applicable to a history $h = \langle T, S, s, \leq, next, 0, now \rangle$. Then by $R(h)$ we denote the result of applying R to h, which is a new (extended) history $h' = \langle T', S', s', \leq', next', 0', now' \rangle$ defined as follows:

- if R is of the form $\varphi \rightarrow move(new)$ then $T' = T \cup \{a\}$, where a is a fresh element, i.e. $a \notin T$, $\leq'$ coincides with $\leq$ on all pairs of elements from T and $\forall t \in T(t \leq a)$. A constant now' is interpreted as a newly added element a (maximal in T'); 0'=0; $S' = S$; $s'(now') = b$, where b is *any* element of S such that $\forall x \in T\ s(x) \neq b$; $next'(t) = next(t)$ for all $t < now$, $next'(now) = now'$.

- if R is of the form $\psi \rightarrow move(s(t))$ then h' is defined as above except $s'(now') = s(t)$.

Definition 2.2. *A* program *is any finite set of rules.*

Definition 2.3. *A* history dependent machine M *over set of states* S *is a pair* $\langle h, P\rangle$*, where* $h \in \mathcal{H}(S)$ *is an* initial history *and* P *is a program.*

Let h be a history and Π a program. The result of applying Π to h is a *partial* function $\Pi : \mathcal{H}(S) \rightarrow \mathcal{H}(S)$ defined as follows:

- if for all applicable $R_1, \ldots R_k \in \Pi$ $R_1(h) = \ldots = R_k(h) = h'$ then $\Pi(h) = h'$;

- as a particular case of above, If there is a single applicable $R \in \Pi$ then $\Pi(h) = R(h)$;

- if there are no applicable rules then the result of $\Pi(h)$ is not defined;

- if there are two applicable rules R_1 and R_2 such that $R_1(h) \neq R_2(h)$ then the result $\Pi(h)$ is not defined.

Let $M = \langle h, P\rangle$ be a history dependent machine. The *computation*, or *run* of M is defined as a sequence of histories $h_0, \ldots, h_i, \ldots$ where $h_0 = h$ and $h_{i+1} = \Pi(h_i)$. If for all $i \in N$ $\Pi(h_i)$ is defined then machine generates an infinite computation $h_0, \ldots, h_i, \ldots$. The assumption about the termination condition can be defined in the following way. If for some i $R(h_i)$ is undefined and for all $j < i$ $R(h_j)$ is defined then the machine generates a finite computation sequence $h_0, \ldots, h_i$.

A sequence of states $[s_0, s_1, \ldots, s_k], s_{k+1}, \ldots, s_i, \ldots$ generated by a computation is called the history of computation. Here $[s_0, s_1, \ldots, s_k]$ is an initial history and for $j \geq k$ s_{j+1} is a state generated at the step $j - k + 1$.

2.1. Possible variations of the model

The above definition of a program applying to the history makes the model *deterministic*. Alternatively, one may skip last item in the definition and allow the computation to proceed even if there are conflicting rules. It leads either to non-deterministic models or to deterministic models with the specified choice between possible alternatives. This rises the question of comparing such models in terms their computational strength. Further, one may include the explicit notion of the *final* states in the model assuming that the computation stops once a final state appeared in the history.

What we show here that even a restricted version of the simple deterministic history dependent machines is a computationally universal model.

3. Universality of the History Dependent Machines

The main aim of this section is to demonstrate the computational universality of the HDM by simulating a well-known model of counters automata. The task is quite challenging since a single player accesses its own history with a very limited set of temporal queries (compositions of $first$, $last$ and now).

The universality result is based on the idea that the movement of a single pebble over elements of the infinite domain can simulate basic type of memory such as integer value counters. In fact we use different time slices to encode several counters in the the history of pebble moves. Using that we then can simulate the behaviour of multi-counters automata with a single state. We start with showing that the latter model is equivalent to Minsky machines (2-counter automata with states).

3.1. Multi-counter automata with a single state

Let us consider the restricted, a *single-state* variant of the multi-counter automaton, which differs from the general model [5] as follows:

Definition 3.1. A multi-counters automaton with a single state is a pair (C, T) where

- C is finite set of counter names;
- T is the set of rules built on the alphabet C, A member of T is an expression of the form **if** Φ $\mathbf{then}$ U, where Φ is either $\mathbf{true}$ or a conjunction of atomic formulas of the form $x = 0$ or $x > 0$ with $x \in C$ and U is a set of updates of the form $x := x + c$, where $x \in C$ and $c \in \{-1, 0, 1\}$.

A configuration of the automaton is an evaluation $v : C \to \mathbb{N}$ of counters names. If $C = \{c_1, \ldots c_n\}$ we represent configuration as an integer-valued vector $v = (v_1, \ldots, v_n) = (v(c_1), \ldots, v(c_n))$. The automaton may make a transition from a configuration v to a configuration v', if there is a rule $r \in T$ such that $r =$ **if** Φ $\mathbf{then}$ U, $v(C) \models \Phi$ and v' is obtained by application of all updates from U to v. If $\Phi \equiv true$ than the corresponding transition can be made from any configuration. If the set of updates U is inconsistent (e.g. containing updates $x := x + 1$ and $x := x - 1$) then corresponding transition can not be made. As an example consider the rule **if** $c_1 = 0 \ \wedge \ c_2 > 0$ **then** $c_1 := c_1 + 1; c_2 := c_2 - 1$ This rule is applicable for the configurations where $c_1 = 0$ and $c_2 > 0$ and when applied the value of c_1 will be increased by 1, the value of c_2 will be decreased by 1 and the values of all other counters (if any) will be preserved.

This model is a particular case of multiple counters automata defined, e.g. in [5]. The difference is that in our model we do not use a finite state control and the form of possible guards is restricted. Still, it is not difficult to show that multi-counter automaton with a single state is an universal model of computation, e.g. Minsky machines [13] can be embedded easily into that model.

Proposition 3.1. The model of a multi-counter automaton with a single state is a universal model of computation.

Proof:
A Minsky machine may be defined as a simple imperative program M which is operating on two non-negative counters and consisting of a sequence of instructions labelled by natural numbers from 1 to some L. Any instruction is one of the following forms:

l: ADD 1 to S_k; GOTO l';

l: IF $S_k \neq 0$ THEN SUBTRACT 1 FROM S_k; GOTO l' ELSE GOTO l'';

l: STOP.

where $k \in \{1, 2\}$ and $l, l', l'' \in \{1, \ldots, L\}$.

Any Minsky machine M can be translated into an equivalent multi-counter automaton with one state using the following simple idea. If M has L instructions then we take set of counters names $C = \{c_1, \ldots, c_L, c_{L+1}, c_{L+2}\}$. The counters c_{L+1}, c_{L+2} represent the counters S_1 and S_2 of M, while the remaining counters $c_1, \ldots, c_L$ represent finite state control. If the machine executes an instruction with the label (number) i then we model this by setting $c_i = 1$ and $c_j = 0$ for all j such that $1 \leq j \leq L$ and $j \neq i$. We leave design of suitable translation and checking of all obvious details to the reader. $\square$

Now we show that the model of HDM can simulate any multi-counter automaton with a single state defined above. We start with showing how the movement of a single pebble over elements of the infinite domain can simulate basic type of memory such as integer-valued counter. Then we extend it to the more sophisticated system of single pebble movements simulating two (and any finite number) of counters.

3.2. Modeling of a single counter

We show in this section how the movement of a single pebble over elements of the infinite domain can simulate basic type of memory such as integer-valued counter. that stores a single natural number.

Let S be a set of states and $H = s_0, \ldots, s_i, \ldots$ is a history of computation of some HDM over S, i.e., in other words, it is a sequence of pebble positions. Let us associate with H directed, edge labelled, possibly infinite graph G:

- the set of vertices V of G is the set of all visited states in the history of computation, i.e. $V = \{s_0, \ldots, s_i, \ldots\}$.
- a triple $\langle a, t, b \rangle \in V \times \mathbb{N} \times V$ is an edge in G labelled by a natural number t iff the pebble has moved from the element a to the element b at the moment of time t.

We can consider now the *length of the shortest path* from the *initial vertex* s_0 to some vertex s_t in G as the value of an integer-valued counter at the moment t. Assuming that, incrementing the counter by one corresponds to moving a pebble to a *new* (not visited before) element. Decrementing the counter corresponds to returning the pebble to the position from where the pebble was moved to the current position *for the first time*. Example 3.1 illustrates the idea.

Example 3.1. To illustrate the idea of counter simulation consider as an example the sequence of the updates shown in the Table 1. The trace of the pebble movements simulating these updates is shown on the Figure 1 as a graph G. The number inside of each vertex shows the distance from the initial vertex. Note, that we can easily test whether the counter have a zero or non-zero value by checking that $s(now) = s(first(now))$ or $s(now) \neq s(first(now))$. In other words, we check that the current element coincides or does not coincide with an element from which we moved to it for the first time. It is easy to see that only unique element satisfies the condition $s(now) = s(first(now))$ that is initial state s_0.

$Time$	$Update$	$Value$	$Rule\ applied$
1	$c := c+1$	1	$true \rightarrow move(new)$
2	$c := c+1$	2	$true \rightarrow move(new)$
3	$c := c$	2	$true \rightarrow move(s(now))$
4	$c := c-1$	1	$true \rightarrow move(s(first(now)))$
5	$c := c+1$	2	$true \rightarrow move(new)$
6	$c := c+1$	3	$true \rightarrow move(new)$
7	$c := c-1$	2	$true \rightarrow move(s(first(now)))$
8	$c := c-1$	1	$true \rightarrow move(s(first(now)))$
9	$c := c-1$	0	$true \rightarrow move(s(first(now)))$
10	$c := c+1$	1	$true \rightarrow move(new)$

Table 1. Sequence of counter updates during 10 time steps.

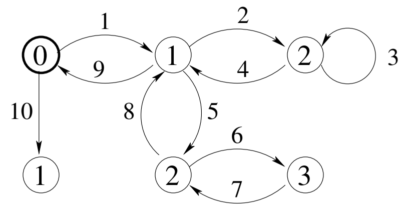

Figure 1. The directed edge-labelled graph that represents the dynamics of counter updates.

3.3. Modelling of several counters

Applying more sophisticated encoding one can keep even more information than one integer value in the trace of the moving pebble. We start with detailed explanation of two counters simulation that can be easy extended for a more general case.

Let $H = s_0, s_1, \ldots, s_i, \ldots$ be a history over S, i.e. be a sequence of pebble positions and its graphical interpretation is defined by a graph G. Split H into two disjoint subsequences: $H_0 = s_0, s_2, \ldots, s_{2i}, \ldots$ that corresponds to visited positions in *even* moments of time including initial position s_0 and $H_1 = s_1, s_3, \ldots, s_{2i+1}, \ldots$ that corresponds to visited positions in *odd* moments of time time.

Now we use H_0 and H_1 for modelling two counters in the same way as we did it for one counter simulation. We associate independently constructed G_0 and G_1 with sequences H_0 and H_1 that we can easy extract from the original trace of a single pebble in graph G. In this case the length of a shortest path in G_0 from initial to the last visited vertex keeps the first counter value and the length of a shortest path in G_1 from initial to the last visited vertex keeps a value for the second counter.

In this case two consecutive moves of a single pebble in time $2i$ and $2i+1$ for $i \geq 1$ correspond to updates of two counters. We assume that a pebble uses the first moment of time for initialization, where it applies the rule $s(now) = s(first(now)) \rightarrow move(new)$. In other words it moves to a new state to meet initial conditions for the above coding, i.e. after applying this rule each of graphs (G_0 and G_1) contains a single node each, that corresponds to a zero value in counters.

To increase the value of the first or the second counter as in the case of one counter one has to apply a rule of the form $\phi \rightarrow move(new)$. The decreasing of the counter is also similar to the one-counter case, but requires more complex references in time to get or update an information in an appropriate slice (i.e

H_0 or H_1).

The Table 2 shows the set of rules that allow us to control these values and update the counters by incrementing they values by one, decrementing by one or keep they unchanged. In order to illustrate this

$Update$	$Rule\ applied$
$c_i := c_i + 1$	$true \rightarrow move(new)$
$c_i := c_i - 1$	$true \rightarrow move(s(last(first(last(now)))))$
$c_i := c_i$	$true \rightarrow move(s(last(now)))$

Table 2. Updates for counter c_i at the moment of time t, $i \equiv t \pmod 2$.

idea we suggest to consider the Example 3.2.

Example 3.2. Let us illustrate the computation of two-counter automata by HDM. The Table 3 shows the sequence of possible counter updates and the Figure 2 contains a single graph G that covers the updates of both counters simulated by a single pebble. Other two graphs G_0 and G_1 are given here to show independent updates of two counters.

$Time$	$Update$	$Rule\ applied$
1		$true \rightarrow move(new)$
2	$c_1 := c_1 + 1$	$true \rightarrow move(new)$
3	$c_2 := c_2$	$true \rightarrow move(s(last(now)))$
4	$c_1 := c_1 + 1$	$true \rightarrow move(new)$
5	$c_2 := c_2 + 1$	$true \rightarrow move(new)$
6	$c_1 := c_1 - 1$	$true \rightarrow move(s(last(first(last(now)))))$
7	$c_2 := c_2$	$true \rightarrow move(s(last(now)))$
8	$c_1 := c_1 + 1$	$true \rightarrow move(new)$
9	$c_2 := c_2 - 1$	$true \rightarrow move(s(last(first(last(now)))))$
10	$c_1 := c_1 - 1$	$true \rightarrow move(s(last(first(last(now)))))$
11	$c_2 = c_2 + 1$	$true \rightarrow move(new)$

Table 3. Sequence of counter updates and corresponding pebble rules

If the last move of the pebble updates the second counter the condition

$$s(last(now)) = s(first(last(now)))$$

is true when the first counter has zero value and the condition $s(first(now)) = s(last(first(now)))$ is true when the second counter has zero value. In a similar way we check non-emptiness of the counters. Using the same technique of allocating different counter updates to different time intervals we can simulate any finite number of positive integer value counters.

Now let us show how to translate a rule of a *deterministic* multi-counter automaton with a single state

if Φ **then** update of c_1,update of $c_2, \ldots,$ update of c_n

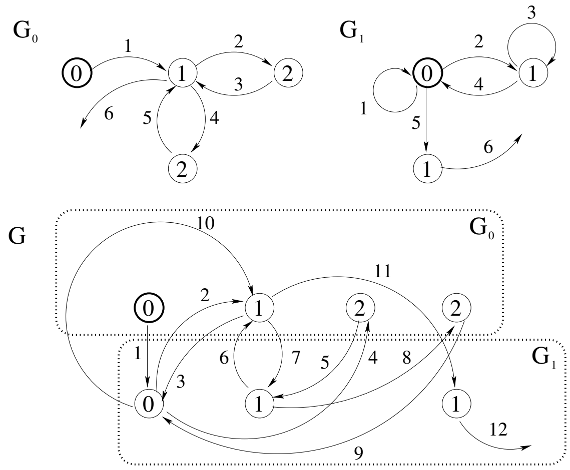

Figure 2. Modelling of two counters by a single pebble.

into the sequence of rules for the HDM.

First translate (non-)zero checking conditions Φ componentwise. The result we denote by Φ', that is a conjunction of equalities of terms (i.e. checking emptiness of all counters). We will use it to identify what is a current state of the counter automata. Then we translate the set of counters updates into a sequence of suitable pebble moves $move_1, \ldots, move_n$ for each counter. In order to overcome one structural restriction that is only one movement of a pebble can be done at a time we define set of n formulas $\Phi'(now)$, $\Phi'(last(now)), \ldots$, $\Phi'(last(\ldots last(now)\ldots))$. Here $\Phi'(t(now))$ denotes a result of replacement of all occurrences of now in Φ' with the term $t(now)$.

Thus a rule is translated into a set of formulae:

$$\begin{gathered} \Phi'(now) \rightarrow move_1 \\ \Phi'(last(now)) \rightarrow move_2 \\ \ldots \\ \Phi'(last(\ldots last(now)\ldots)) \rightarrow move_n \end{gathered}$$

implementing a sequence of counters updates. During all n moves of a pebble, Φ' is referring to the same moment of time and checking counter values that moment in order to simulate parallel update of counters by the sequential run. It is easy to see that the modelling is done in real time, that is every step of n-counter machine is modelled by exactly n steps of HDM.

Taking into account Proposition 3.1 and the above constructions the following holds:

Theorem 3.1. The *model of HDM* is a universal model of computation.

Corollary 3.1. Let M be a given HDM with a set of rules P. The liveness problem: "whether a rule $R \in P$ will be applicable during the run of machine M" is algorithmically undecidable.

Following the reduction of halting problem to the question about existence of periodic orbits in counter machines, presented in [3], we can formulate another corollary:

Corollary 3.2. Let M be a given HDM. The problems to decide whether a number of HDM states is finite and whether a sequence of visited states is periodic are both algorithmically undecidable.

4. Temporal logic with predicate λ-abstraction

When querying the history we are mainly interested in temporal properties. Because in HDM we assume potentially infinite set of states the propositional temporal logic is not enough to get an equivalent model - one needs some first-order extension.

It is fairly straightforward to devise an alternative (and even equivalent) model of HDM using a fragment of first-order temporal logic. So we leave it to an interested reader. Instead, we present in this section an extension of propositional temporal logic by predicate λ-abstraction suitable for this purpose. We show that in that extension one can specify faithfully the computations of HDM. Then as the application of the result on universality of HDMs we show that the fragment is not recursively axiomatizable.

In [7] M.Fitting has proposed the idea of extension of propositional modal logics by predicate (lambda-)abstraction. He suggested to extend a modal propositional logic L by adding relation symbols, flexible constants and the operator of predicate abstraction, but no quantifiers. For a propositional modal logic L such an extension $L_{\lambda(=)}$ (either with or without equality) can be alternatively seen as a very restricted fragment of corresponding first-order variant QL of L. Thus, this is a general technique for obtaining the logics, which are in a sense, intermediate between propositional and first-order. It is proved in [7] that $S5_{\lambda=}$ is undecidable, but for many other classical modal logics L their extensions $L_{\lambda(=)}$ are still decidable.

In [9, 10] we have started the study of such an extension applied to the classical propositional linear time logic (with past and future modalities). The full description of syntax and semantics of $LTL_{\lambda=}$ is given in [9, 10] (see also [7] for the general case of modal logics). Here we outline only some crucial clauses.

Given the alphabets of *variables*, *constant symbols* and *predicate symbols* the formulae and terms of $LTL_{\lambda=}$ are defined as in the case of first-order temporal logic, except:

• quantifiers are not allowed;

• If φ is a formula, x is a variable, and t is a term, that is a variable, or a constant, then $\langle \lambda x.\varphi \rangle(t)$ is a well-formed formula.

Formulae of $LTL_{\lambda=}$ are interpreted in first-order temporal models with constant domain and with the time flow isomorphic to ω.

Definition 4.1. A *model* is a structure $\mathfrak{M} = \langle D, \mathcal{I} \rangle$, where:

1. D is a non-empty set, the domain;
2. $\mathcal{I}$ is an interpretation mapping that assigns:
    - to each constant symbol some function from $\mathbb{N}$ to D;
    - to each n-ary relation symbol some function from $\mathbb{N}$ to 2^{D^n} (the power set of D^n)
3. $\mathcal{I}(=)$ is the constant function assigning the equality relation on D to every moment of time from $\mathbb{N}$.

First-order (non-temporal) structures corresponding to each point of time will be denoted $\mathfrak{M}_n = \langle D, \mathcal{I}(n) \rangle$. Intuitively, first-order temporal model $\mathfrak{M}$ is a sequence of first-order structures, or *states*, such as $\mathfrak{M}_0, \mathfrak{M}_1, \ldots, \mathfrak{M}_n \ldots$. Predicates and constants are assumed to be *flexible*, meaning their interpretations depend on moments of time. In contrast, variables are considered as *rigid*, that is assignments do not depend on time in which variables are evaluated.

An *assignment* in D is a function $\mathfrak{a}$ from the set of individual variables to D.

Definition 4.2. The *truth-relation* $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ (or simply $n \models^{\mathfrak{a}} \varphi$, if $\mathfrak{M}$ is understood) in the structure $\mathfrak{M}$ for the assignment $\mathfrak{a}$ is defined inductively in the usual way under the following semantics of temporal operators:

$n \models^{\mathfrak{a}} \bigcirc\varphi$ iff $n+1 \models^{\mathfrak{a}} \varphi$;

$n \models^{\mathfrak{a}} \Diamond\, \varphi$ iff there is $m \geq n$ such that $m \models^{\mathfrak{a}} \varphi$;

$n \models^{\mathfrak{a}} \Box\varphi$ iff $m \models^{\mathfrak{a}} \varphi$ for all $m \geq n$;

$n \models^{\mathfrak{a}} \odot\varphi$ iff $n-1 \models^{\mathfrak{a}} \varphi$;

$n \models^{\mathfrak{a}} \blacklozenge\, \varphi$ iff there is $0 \leq m \leq n$ such that $m \models^{\mathfrak{a}} \varphi$;

$n \models^{\mathfrak{a}} \blacksquare\, \varphi$ iff $m \models^{\mathfrak{a}} \varphi$ for all $0 \leq m \leq n$.

For the case of abstraction we have:

$n \models^{\mathfrak{a}} \langle \lambda x.\varphi \rangle(t)$ iff $n \models^{\mathfrak{a}'} \varphi$, where $\mathfrak{a}'$ coincides with $\mathfrak{a}$ on all variables except x and $\mathfrak{a}'(x)$ equals to the interpretation of t at the moment n.

A formula φ is said to be *valid* iff $\mathfrak{M}_0 \models^{\mathfrak{a}} \varphi$ for every structure $\mathfrak{M}$ and for all assignments $\mathfrak{a}$.

Simple example illustrating the above semantics of λ-abstraction is given by the following two formulae: $\Box\langle \lambda x.P(x)\rangle(c)$ and $\langle \lambda x.\Box P(x)\rangle(c)$. The first says that at any moment in the future P holds on the element of the domain which c *then* denotes, while the second one says that always in the future P holds on the element of the domain which c denotes *now*.

Despite being very restricted fragment of the first-order temporal logic the logic $LTL_{\lambda=}$ can express many non-trivial properties of its models. Because of the interpretation given to flexible constants they can be though of as the *pebbles* moving over elements of the domain. Thus, $LTL_{\lambda=}$ can be used to specify dynamic properties of pebble systems and, in particular, history dependent machines [10]. For example, the formula $\Box\langle \lambda x. \bigcirc \Box\langle \lambda y.y \neq x\rangle(d)\rangle(d)$ specifies that the constant d denotes different elements of the domain at different moments of time (the pebble d occupies different positions in different moments of time).

We have shown in [9] that $LTL_{\lambda=}$ is not recursively axiomatizable:

Theorem 4.1. [9] The set of valid formulas of $LTL_{\lambda=}$ is not recursively enumerable.

In fact, very restricted fragment of $LTL_{\lambda=}$ has been used to model faithfully the universal model of computation, that is two counter Minsky machine. So, we have proved the above theorem for $LTL_{\lambda=}$ in the vocabulary consisting of *three* flexible constants and no predicate symbols, except equality and 0-ary predicate symbols (propositional letters). We denote such a fragment by $LTL^3_{\lambda=}$.

Here we show that one can model faithfully the computations of HDMs in $LTL_{\lambda=}$ with just one flexible constant.

Theorem 4.2. $LTL^1_{\lambda=}$ is not recursively axiomatizable.

Proof:
We demonstrate how any history-dependent machine M used in the modelling multiple-counter automata can be translated into a formula Φ_M of $LTL^1_{\lambda=}$ whose models are precisely computations of M. First, we introduce the formula $first^\tau(x)$ of $LTL^1_{\lambda=}$ capturing the semantics of $first$ construct:

$$first^\tau(x): \quad \langle \lambda y.(x = y)\rangle(a) \wedge \bigodot \blacksquare \langle \lambda y.(x \neq y)\rangle(a)$$

Further, define translations:

$$(move(new))^\tau : \qquad \bigcirc \langle \lambda x.(\odot \blacksquare \langle \lambda y.(x \neq y)\rangle(a))\rangle(a)$$

$$(move(s(last(now))))^\tau : \qquad \bigcirc \langle \lambda x.(\odot \odot \langle \lambda y.(x = y)\rangle(a))\rangle(a)$$

$$(move(s(last(first(last(now))))))^\tau : \qquad \bigcirc \langle \lambda v. \odot \odot \langle \lambda x.\blacklozenge(first^\tau(x) \wedge \odot \langle \lambda z.z = v\rangle(a))\rangle(a)\rangle(a)$$

$$(s(now) = s(first(now)))^\tau : \qquad \langle \lambda x.\blacklozenge first^\tau(x)\rangle(a)$$

$$(s(last(now)) = s(first(last(now))))^\tau : \ \odot \langle \lambda x.\blacklozenge first^\tau(x)\rangle(a)$$

Given a rule R of the form $\wedge_i \phi_i \rightarrow move(\ldots)$, define its translation R^τ as a $LTL^1_{\lambda=}$ formula $\Box(\wedge_i \phi_i^\tau \rightarrow (move(\ldots))^\tau)$. Now, given a program $\Pi = \{R_1, \ldots, R_k\}$ of an HDM define its translation $\Pi^\tau = \{R_1^\tau, \ldots, R_k^\tau\}$

Straightforward, but tedious checking shows that we get a faithful translation, i.e. any model of Π^τ is a computation of Π and a computation of Π is a model of Π^τ. Further, consider the formula $\Psi \Leftrightarrow \Box\Diamond(s(now) = s(first(now)))^\tau$ which says that the specified (first) counter gets the value 0 infinitely often. The set of multi-counter automata with such a property is not recursively enumerable. It follows that a set of valid formulae of the form $\Pi^\tau \rightarrow \Psi$ is not r.e. which, in turn, implies the statement of the theorem. $\Box$

In [6] a constraint linear temporal logic Contraint LTL with the freeze quantifier is studied. As it has turned out, this logic and our $LTL_{\lambda=}$ are syntactical variants of each other. The recursive non-axiomatizability of Constraint LTL with the freeze quantifier and one flexible variable ($\equiv$ flexible constant in our terms) is proven in [6] also using a modelling of (2-)counter machine computations. The modelling in [6], unlike ours, is not a real time: one needs unbounded numbers of time slices to model one step of a 2-counter machine.

4.1. Executable semantics of $LTL^1_{\lambda=}$

The above modelling suggests that one can reverse argument and assign an executable semantics to a subset of $LTL^1_{\lambda=}$ according to the principle " declarative past implies imperative future" [2]. Here we assume that the language of $LTL^1_{\lambda=}$ contains no predicates except equality. Consider a formula Φ of the form $P \rightarrow \bigcirc\langle\lambda x.(\bigodot\varphi)\rangle(a))$, where P and φ are formulae without future time operators. Let $\mathfrak{M}_0, \mathfrak{M}_1, \ldots, \mathfrak{M}_n$ be an initial fragment of a first-order temporal model. Then one-step execution of Φ does the following: if $\mathfrak{M}_n \models P$ then the initial fragment of the temporal model is extended with the structure $\mathfrak{M}_{n+1}$ such that $\mathfrak{M}_n \models \bigcirc\langle\lambda x.(\bigodot\varphi)\rangle(a))$. Notice that this extension amounts to choosing (in general, non-deterministically) an interpretation of a at the moment $n+1$. It is straightforward to extend the executable semantics to the arbitrary conjunctions of $\square$-closed formulae of the above form. Then we get computational universality of the very restricted fragment of executable first-order temporal logic.

5. Concluding remarks

We have introduced a simple abstract computation model of history-dependent machines and have shown that the model is computationally universal. It is shown by modelling muliticounter machine computations in real time.This reveals the computational power of querying the history as a programming primitive. We have pointed out two relevant lines of research. One is related to the development of Elephant (and the like) programming language(s), another to the investigation of a temporal logic as a programming language itself (METATEM approach). Making a link with the second line we have outlined how to assign an executable semantics to a very restricted subset of the first-order temporal logic to get an universal model of computation. More work on clarification of relationships with these two lines is required.

Finally, we would like to point out that in the model of history-dependent machines we have two natural "dimensions": one is a query language (in our case it is fragments of first-order (temporal) logics), and another is a family of allowed sets of possible states (we did not impose any restrictions here). Varying both "dimensions" we can get either universal models, or weaker but still interesting models with many decidable properties. For example, if one restricts a set of possible states to be a fixed finite set, then the computations of a history-dependent machine over this set can be specified faithfully in the propositional temporal logic, which leads to decidability of all natural properties of the model. On the other hand if we enrich a query language with aggregation facilities, for example including a variant of counting quantifiers, than we may get a universal model even for a fixed finite set. One of the future directions is to understand better the border between universality (undecidability) and decidability in this setting.

References

[1] Jiannong Cao, Xinyu, Feng, Jian Lu, Sajal K. Das, Mailbox-Based Scheme for Mobile Agent Communications, *IEEE Computer*, Vol. 35, No. 9, September 2002, 54–60.

[2] H. Barringer, M.Fisher, D. Gabbay, R.Owens, M.Reynolds (eds.), *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press Ltd, 1996.

[3] V. Blondel, J. Cassaigne, C. Nichitiu, *On the Presence of Periodic Configurations in Turing Machines and in Counter Machines*, Theoretical Computer Science, 289, (2002), 573-590.

[4] Jan Chomicki, Damian Niwiński, On the Feasibility Checking Temporal Integrity Constraints, *Journal of Computer and System Sciences*, 51(3), pp 523–535, 1995.

[5] H. Comon and Y.Jurski, Multiple counters automata, safety analysis and Presburger arithmetic, Research Report LSV-98-1, Mar.1998, Laboratoire Specification at Verification.

[6] Stéphane Demri, Ranco Lazić, David Nowak, On freeze quantifier in Constraint LTL: decidablity and complexity, *12th International Symposium on Temporal Representation and Reasoning, TIME'05*, 113–121, IEEE Computer Society, 2005.

[7] M. Fitting, Modal Logic Between Propositional and First Order, *Journal of Logic and Computation*, 12, 1017-1026, 2002.

[8] Dov M.Gabbay, Mark A. Reynolds, Marcelo Finger, Temporal Logic. Mathematical Foundations and Computational Aspects, Volume 2, Oxford Science Publications, Clarendon Press, 2000.

[9] Alexei Lisitsa and Igor Potapov, Temporal logic with predicate abstraction, CoRR cs.LO/0410072, The Computing Research Repository, `http://xxx.lanl.gov/archive/cs/`, 14pp, 2004.

[10] Alexei Lisitsa and Igor Potapov. Temporal logic with predicate lambda-abstraction, *12th International Symposium on Temporal Representation and Reasoning, TIME'05*, 147–155, IEEE Computer Society, 2005.

[11] John McCarthy. Elephant 2000. Technical report, Stanford Formal Reasoning Group, 1996. Available only as http://www-formal.stanford.edu/jmc/elephant.html.

[12] M. Minsky, Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines, *Annals of Mathematics*, 74, pp 437–455, 1961.

[13] M. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall International, 1967.

[14] A. Sistla, O. Wolfson, Temporal Triggers in Active Databases, *IEEE Transactions on Knowledge and Data Engineering*, 7 (3), pp 471-486, 1995.