Theory and Applications of Mathematics & Computer Science

(ISSN 2067-2764, EISSN 2247-6202)
http://www.uav.ro/applications/se/journal/index.php/tamcs

Theory and Applications of Mathematics & Computer Science 5 (1) (2015) 37–52

The Applicability of $-Calculus to Solve Some Turing Machine Undecidable Problems

Eugene Eberbach[a,*]

[a]*Department of Engineering and Science, Rensselaer Polytechnic Institute, Hartford, 275 Windsor Street, CT 06120, United States.*

Abstract

The $-calculus process algebra for problem solving applies the cost performance measures to converge in finite time or in the limit to optimal solutions with minimal problem solving costs. The $-calculus belongs to superTuring models of computation. Its main goal is to provide the support to solve hard computational problems. It allows also to solve in the limit some undecidable problems. In the paper we demonstrate how to solve in the limit Turing Machine Halting Problem, to approximate the universal search algorithm, to decide diagonalization language, nontrivial properties of recursively enumerable languages, and how to solve Post Correspondence Problem and Busy Beaver Problem.

Keywords: problem solving, hypercomputation, expressiveness, superTuring models of computation, resource bounded computation, process algebras, $-calculus.
2010 MSC: primary classification: 68Qxx, 68Txx.
2012 CCS: primary classification: Theory of computation, mathematics of computing. Secondary classifications: models of computation, Turing machines, concurrency, process calculi, formal languages and automata theory, formalisms, automata over infinite objects, mathematical optimization, discrete optimization.

1. Introduction

In this paper, the expressiveness of the $-calculus process algebra of bounded rational agents (Eberbach, 1997, 2005*a*, 2006, 2007) is investigated. The $-calculus, presented in this paper, belongs to superTuring models of computation and provides a support to handle intractability and undecidability in problem solving. In the paper, we present the applicability of $-calculus to solve (in hypercomputational sense) some undecidable problems.

The paper is organized as follows. In section 2, we briefly recall some basic notions related to Turing machine problem solving and hypercomputation. In section 3, we outline the $-calculus

*Corresponding author
Email address: eberbe@rpi.edu (Eugene Eberbach)

process algebra of bounded rational agents. In section 4, we present the solution of the halting problem of Universal Turing Machine, and approximate solution of the universal search algorithm. In section 5, other TM unsolvable problems are investigated, including the diagonalization language, nontrivial properties of recursively enumerable languages, Post Correspondence Problem and Busy Beaver Problem. Section 6 contains conclusions.

2. Problem Solving in Turing Machines and Hypercomputation

Turing Machines (TMs) (Turing, 1937, 1939) and algorithms are two fundamental concepts of computer science and problem solving. Turing Machines describe the limits of problem solving using conventional recursive algorithms, and laid the foundation of current computer science in the 1960s.

Note that there are several other models of algorithms, called super-recursive algorithms, that can compute more than Turing Machines, using hypercomputational/superTuring models of computation (Burgin, 2004; Syropoulos, 2007). The battle between reductionists (believing in strong Church-Turing Thesis and "unsinkability" of Turing machine model) and remodelers (hyper-computationalists trying to develop new super-Turing models of computation for solution of Turing Machine undecidable problems) is not over, however shifting gradually in favor of hyper-computationalists (Aho, 2011; Cooper, 2012; Wegner *et al.*, 2012).

It turns out that (TM) *undecidable problems* cannot be solved by TMs and *intractable problems* are solvable, but require too many resources (e.g., steps or memory). For undecidable problems effective recipes do not exist - problems are called nonalgorithmic or nonrecursive. On the other hand, for intractable problems algorithms exist, but running them on a deterministic Turing Machine, requires an exponential amount of time (the number of elementary moves of the TM) as a function of the TM input.

We use the simplicity of the TM model to prove formally that there are specific problems (languages) that the TM cannot solve (Hopcroft *et al.*, 2001). Solving the problem is equivalent to decide whether a string belongs to the language. A problem that cannot be solved by computer (Turing machine) is called *undecidable* (TM-undecidable). The class of languages accepted by Turing machines are called *recursively enumerable (RE-) languages*. For RE-languages, TM can accept the strings in the language but cannot tell for certain that a string is not in the language.

There are two classes of Turing machine unsolvable languages (problems):

recursively enumerable RE but not recursive - TM can accept the strings in the language but cannot tell for certain that a string is not in the language (e.g., the language of the universal Turing machine, or Post's Correspondence Problem language). A language is decidable but its complement is undecidable, or vice versa: a language is undecidable but its complement is decidable.

non-RE - no TM can even recognize the members of the language in the RE sense (e.g., the diagonalization language). Neither a language nor its complement is decidable.

Decidable problems have a (recursive) algorithm, i.e., TM halts whether or not it accepts its input. Decidable problems are described by *recursive languages*. Algorithms as we know are associated

with the class of recursive languages, a subset of recursively enumerable languages for which we can construct its accepting TM. For recursive languages, both a language and its complement are decidable.

Turing Machines are used as a formal model of classical (recursive) algorithms. An algorithm should consist of a finite number of steps, each having well defined and implementable meaning. We are convinced that computer computations are not restricted to such restrictive definition of algorithms only.

Definition 2.1. *By* superTuring computation *(also called* hypercomputation*) we mean any computation that cannot be carried out by a Turing Machine as well as any (algorithmic) computation carried out by a Turing Machine.*

In (Eberbach & Wegner, 2003; Eberbach *et al.*, 2004), several superTuring models have been discussed and overviewed. The incomplete list includes Turing's o-machines, c-machines and u-machines, cellular automata, discrete and analog neural networks, Interaction Machines, Persistent Turing Machines, Site and Internet Machines, the π-calculus, the $-calculus, Inductive Turing Machines, Infinite Time Turing Machines, Accelerating Turing Machines and Evolutionary Turing Machines. In particular, the author proposed two superTuring models of computation: the $-Calculus (Eberbach, 2005*a*, 2007) and Evolutionary Turing Machine (Eberbach, 2005*b*; Eberbach & Burgin, 2009).

SuperTuring models derive their higher than the TM expressiveness using three principles: *interaction, evolution*, or *infinity*. In the *interaction principle* the model becomes open and the agent interacts with either a more expressive component or with an infinite many components. In the *evolution principle*, the model can evolve to a more expressive one using non-recursive variation operators. In the *infinity principle*, models can use unbounded resources: time, memory, the number of computational elements, an unbounded initial configuration, an infinite alphabet, etc. The details can be found in (Eberbach & Wegner, 2003; Eberbach *et al.*, 2004).

3. The $-Calculus Algebra of Bounded Rational Agents

The $-calculus is a mathematical model of processes capturing both the final outcome of problem solving as well as the interactive incremental way how the problems are solved. The $-calculus is a process algebra of Bounded Rational Agents for interactive problem solving targeting intractable and undecidable problems. It has been introduced in the late of 1990s (Eberbach, 1997, 2005*a*, 2007). The $-calculus (pronounced cost calculus) is a formalization of resource-bounded computation (also called anytime algorithms), proposed by Dean, Horvitz, Zilberstein and Russell in the late 1980s and early 1990s (Horvitz & Zilberstein, 2001; Russell & Norvig, 2002). Anytime algorithms are guaranteed to produce better results if more resources (e.g., time, memory) become available. The standard representative of process algebras, the π-calculus (Milner *et al.*, 1992; Milner, 1999) is believed to be the most mature approach for concurrent systems.

The $-calculus rests upon the primitive notion of *cost* in a similar way as the π-calculus was built around a central concept of *interaction*. Cost and interaction concepts are interrelated in the sense that cost captures the quality of an agent interaction with its environment. The unique feature of the $-calculus is that it provides a support for problem solving by incrementally searching for

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

3.1. The $-Calculus Syntax

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

Definition 3.1. The $-calculus The set $\mathcal{P}$ of $-calculus process expressions consists of simple $-expressions α and composite $-expressions P, and is defined by the following syntax:

α	::=	$(\$_{i\in I}\ P_i)$	cost
	|	$(\rightarrow_{i\in I}\ c\ P_i)$	send P_i with evaluation through channel c
	|	$(\leftarrow_{i\in I}\ c\ X_i)$	receive X_i from channel c
	|	$('_{i\in I}\ P_i)$	suppress evaluation of P_i
	|	$(a_{i\in I}\ P_i)$	defined call of simple $-expression a with parameters P_i, and and its optional associated definition $(:=\ (a_{i\in I}\ X_i)\ <R>)$ with body R evaluated atomically
	|	$(\bar{a}_{i\in I}\ P_i)$	negation of defined call of simple $-expression a
P	::=	$(\circ_{i\in I}\ \alpha\ P_i)$	sequential composition
	|	$(\parallel_{i\in I}\ P_i)$	parallel composition
	|	$(\cup_{i\in I}\ P_i)$	cost choice
	|	$(\uplus_{i\in I}\ P_i)$	adversary choice
	|	$(\sqcup_{i\in I}\ P_i)$	general choice
	|	$(f_{i\in I}\ P_i)$	defined process call f with parameters P_i, and its associated definition $(:=\ (f_{i\in I}\ X_i)\ R)$ with body R (normally suppressed); $(^1\ R)$ will force evaluation of R exactly once

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

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

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

3.2. The \$-Calculus Semantics: The kΩ-Search

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

The basic \$-calculus problem solving method, the $k\Omega$-optimization, is a very general search method providing meta-control, and allowing to simulate many other search algorithms, including A*, minimax, dynamic programming, tabu search, or evolutionary algorithms (Russell & Norvig, 2002). The problem solving works iteratively: through select, examine and execute phases. In the select phase the tree of possible solutions is generated up to k steps ahead, and agent identifies its alphabet of interest for optimization Ω. This means that the tree of solutions may be incomplete in width and depth (to deal with complexity). However, incomplete (missing) parts of the tree are modeled by silent \$-expressions ε, and their cost estimated (i.e., not all information is lost). The above means that $k\Omega$-optimization may be if some conditions are satisfied to be complete and optimal. In the examine phase the trees of possible solutions are pruned minimizing cost of solutions, and in the execute phase up to n instructions are executed. Moreover, because the \$ operator may capture not only the cost of solutions, but the cost of resources used to find a solution, we obtain a powerful tool to avoid methods that are too costly, i.e., the \$-calculus directly minimizes search cost. This basic feature, inherited from anytime algorithms, is needed to tackle directly hard optimization problems, and allows to solve total optimization problems (the best quality solutions with minimal search costs). The variable k refers to the limited horizon for optimization, necessary due to the unpredictable dynamic nature of the environment. The variable Ω refers to a reduced alphabet of information. No agent ever has reliable information about all factors that

influence all agents behavior. To compensate for this, we mask factors where information is not available from consideration; reducing the alphabet of variables used by the $-function. By using the $k\Omega$-optimization to find the strategy with the lowest $-function, meta-system finds a satisficing solution, and sometimes the optimal one. This avoids wasting time trying to optimize behavior beyond the foreseeable future. It also limits consideration to those issues where relevant information is available. Thus the $k\Omega$ optimization provides a flexible approach to local and/or global optimization in time or space. Technically this is done by replacing parts of $-expressions with invisible $-expressions ε, which remove part of the world from consideration (however, they are not ignored entirely - the cost of invisible actions is estimated).

The $k\Omega$-optimization meta-search procedure can be used both for single and multiple cooperative or competitive agents working online ($n \neq 0$) or offline ($n = 0$). The $-calculus programs consist of multiple $-expressions for several agents.

Let's define several auxiliary notions used in the $k\Omega$-optimization meta-search. Let:

- $\mathcal{A}$ - be an alphabet of $-expression names for an enumerable *universe of agent population* (including an environment, i.e., one agent may represent an environment). Let $\mathcal{A} = \bigcup_i A_i$, where A_i is the alphabet of $-expression names (simple or complex) used by the i-th agent, $i = 1, 2, ..., \infty$. We will assume that the names of $-expressions are unique, i.e., $A_i \cap A_j = \emptyset, i \neq j$ (this always can be satisfied by indexing $-expression name by a unique agent index. This is needed for an agent to execute only own actions). The agent population size will be denoted by $p = 1, 2, ..., \infty$.

- $x_i[0] \in \mathcal{P}$ - be an initial $-expression for the i-th agent, and its initial search procedure $k\Omega_i[0]$.

- $min(\$_i\ (k\Omega_i[t]\ \ x_i[t]))$ - be an implicit default goal and $Q_i \subseteq \mathcal{P}$ be an optional (explicit) goal. The default goal is to find a pair of $-expressions, i.e., any pair $(k\Omega_i[t], x_i[t])$ being

$$min\{(\$_i\ (k\Omega_i[t], x_i[t])) = \$_{1i}(\$_{2i}(k\Omega_i[t]), \$_{3i}(x_i[t]))\},$$

  where $\$_{3i}$ is a problem-specific cost function, $\$_{2i}$ is a search algorithm cost function, and $\$_{1i}$ is an aggregating function combining $\$_{2i}$ and $\$_{3i}$. This is the default goal for total optimization looking for the best solutions $x_i[t]$ with minimal search costs $k\Omega_i[t]$. It is also possible to look for the optimal solution only, i.e., the best $x_i[t]$ with minimal value of $\$_{i3}$, or the best search algorithm $k\Omega_i[t]$ with minimal costs of $\$_{i2}$. The default goal can be overwritten or supplemented by any other termination condition (in the form of an arbitrary $-expression Q) like the maximum number of iterations, the lack of progress, etc.

- $\$_i$ - a cost function performance measure (selected from the library or user defined). It consists of the problem specific cost function $\$_{3i}$, a search algorithm cost function $\$_{2i}$, and an aggregating function $\$_{1i}$. Typically, a user provides cost of simple $-expressions or an agent can learn such costs (e.g., by reinforcement learning). The user selects or defines also how the costs of composite $-expressions will be computed. The cost of the solution tree is the function of its components: costs of nodes (states) and edges (actions). This allows to express both the quality of solutions and search cost.

- $\Omega_i \subseteq \mathcal{A}$ - a *scope of deliberation/interests of the i-th agent*, i.e., a subset of the universe's of \$-expressions chosen for optimization. All elements of $\mathcal{A} - \Omega_i$ represent irrelevant or unreachable parts of an environment, of a given agent or other agents, and will become invisible (replaced by ε), thus either ignored or unreachable for a given agent (makes optimization local spatially). Expressions over $A_i - \Omega_i$ will be treated as observationally congruent (cost of ε will be neutral in optimization, e.g., typically set to 0). All expressions over $\Omega_i - A_i$ will be treated as strongly congruent - they will be replaced by ε and although invisible, their cost will be estimated using the best available knowledge of an agent (may take arbitrary values from the cost function domain).

- $b_i = 0, 1, 2, ..., \infty$ - a branching factor of the search tree (LTS), i.e., the maximum number of generated children for a parent node. For example, hill climbing has $b_i = 1$, for binary tree $b_i = 2$, and $b_i = \infty$ is a shorthand to mean to generate all children (possibly infinite many).

- $k_i = 0, 1, 2, ..., \infty$ - represents *the depth of deliberation*, i.e., the number of steps in the derivation tree selected for optimization in the examine phase (decreasing k_i prevents combinatorial explosion, but can make optimization local in time). $k_i = \infty$ is a shorthand to mean to the end to reach a goal (may not require infinite number of steps). $k_i = 0$ means omitting optimization (i.e., the empty deliberation) leading to reactive behaviors. Similarly, a branching factor $b_i = 0$ will lead to an empty deliberation too. Steps consist of multisets of simple \$-expressions, i.e., a parallel execution of one or more simple \$-expressions constitutes one elementary step.

- $n_i = 0, 1, 2, ..., \infty$ - the number of steps selected for execution in the execute phase. For $n_i > k_i$ steps larger than k_i will be executed without optimization in reactive manner. For $n_i = 0$ execution will be postponed until the goal will be reached.

  For the depth of deliberation $k_i = 0$, the $k\Omega$-search will work in the style of imperative programs (reactive agents), executing up to n_i consecutive steps in each loop iteration. For $n_i = 0$ search will be offline, otherwise for $n_i \neq 0$ - online.

- gp, $reinf$, $strongcon$, $update$ - auxiliary flags used in the $k\Omega$-optimization meta-search procedure.

Each agent has its own $k\Omega$-search procedure $k\Omega_i[t]$ used to build the solution $x_i[t]$ that takes into account other agent actions (by selecting its alphabet of interests Ω_i that takes actions of other agents into account). Thus each agent will construct its own view of the whole universe that only sometimes will be the same for all agents (this is an analogy to the subjective view of the "objective" world by individuals having possibly different goals and different perception of the universe).

Definition 3.2. The $k\Omega$-Optimization Meta-Search Procedure The $k\Omega$- *optimization meta-search procedure* $k\Omega_i[t]$ for the i-th agent, $i = 0, 1, 2, ...$, *from an enumerable* universe of agent population *and working in time generations* $t = 0, 1, 2, ...$ *is a complex \$-expression (meta-procedure) consisting of simple \$-expressions* $init_i[t]$, $sel_i[t]$, $exam_i[t]$, $goal_i[t]$, $\$_i[t]$, *complex \$-expression* $loop_i[t]$ *and* $exec_i[t]$, *and constructing solutions, its input* $x_i[t]$, *from predefined and user defined simple*

and complex $-expressions. For simplicity, we will skip time and agent indices in most cases if it does not cause confusion, and we will write init, loop, sel, exam, goal$_i$ and $\$_i$. Each i-th *agent performs the following kΩ-search procedure k$\Omega_i[t]$ in the time generations $t = 0, 1, 2, ...$:*

```
(:= (kΩ_i[t]  x_i[t]) (∘ (init  (kΩ_i[0] x_i[0]))   // initialize kΩ_i[0] and x_i[0]
    (loop  x_i[t + 1]))                             // basic cycle: select, examine,
)                                                   // execute
```

where loop meta-$-expression takes the form of the select-examine-execute cycle performing the kΩ-optimization until the goal is satisfied. At that point, the agent re-initializes and works on a new goal in the style of the never ending reactive program:

```
(:= (loop  x_i[t])                          // loop recursive definition
   (⊔ (∘ (goal_i[t]  (kΩ_i[t] x_i[t]))      // goal not satisfied, default goal
                                            // min($_i (kΩ_i[t] x_i[t]))
         (sel  x_i[t])                      // select: build problem solution tree k step
                                            // deep, b wide
         (exam  x_i[t])                     // examine: prune problem solution tree in
                                            // cost ∪ and in adversary ⊎ choices
         (exec  (kΩ_i[t] x_i[t]))           // execute: run optimal x_i n steps and
                                            // update kΩ_i parameters
         (loop  x_i[t + 1]))                // return back to loop
      (∘ (goal_i[t]  (kΩ_i[t] x_i[t]))      // goal satisfied - re-initialize search
         (kΩ_i[t] x_i[t])))
)
```

Simple $-expressions init, sel, exam, goal with their atomicly executed bodies are defined below. On the other hand, exec can be interrupted after each action, thus it is not atomic.

1. **Initialization** (:= (*init* (kΩ_i[0] x_i[0])) < *init_body* >): *where init_body* = (∘ ($\leftarrow_{i\in I}$ *user_channel* X_i) (⊔ *cond_init* (∘ *cond_init* (*init_body*))), *and cond_init* = (⊔ ($x_i[0] = \perp$) ($k_i = n_i = 0$)), *and successive* X_i, $i = 1, 2, ...$ *will be the following:* kΩ_i[0] *an initial meta-search procedure (default: as provided in this definition),* $k_i, b_i, n_i, \Omega_i, A_i$ *(defaults:* $k_i = b_i = n_i = \infty$, $\Omega_i = A_i = \mathcal{A}$*); simple and complex $-expressions definitions over* $A_i \cup \Omega_i$ *(default: no definitions);*
   initialize costs of simple $-expressions randomly and set reinforcement learning flag $reinf = 1$ *(default: get costs of simple $-expressions from the user,* $reinf = 0$*);* $\$_{i1}$ *an aggregating cost function (default: addition),* $\$_{i2}$ *and* $\$_{i3}$ *search and solution specific cost functions (default: a standard cost function as defined in the next section);*
   Q_i *optional goal of computation (default: min*($\$_i$ (k$\Omega_i[t], x_i[t]$)));
   $x_i[0]$ *an initial $-expression solution (an initial state of LTS for the i-th agent) over alphabet* $A_i \cup \Omega_i$*. This resets* gp_i *flag to 0 (default: generate* $x_i[0]$ *randomly in the style of genetic programming and* $gp_i = 1$*);*
   / receive from the user several values for initialization overwriting possibly the defaults. If atomic initialization fails re-initialize init. */*
2. **Goal** (:= (*goal*$_i$[t] (kΩ_i[t] x_i[t])) < *goal_body* >): *where goal_body checks for the maximum predefined quantum of time (to avoid undecidability or too long verification) whether*

goal state defined in the init phase has been reached. If the quantum of time expires, it returns false $\perp$.

3. **Select Phase**
   (:= $(sel\ x_i[t])$ < ($\sqcup$ *cond_sel_exam* ($\circ$ $\overline{cond\_sel\_exam}$ *sel_body*)) >)*: where cond_sel_exam =* ($\sqcup$ ($k_i = 0$) ($b_i = 0$)) *and sel_body builds the search tree with the branching factor* b_i *and depth* k_i *over alphabet* $A_i \cup \Omega_i$ *starting from the current state* $x_i[t]$*. For each state s derive actions a being mulitsets of simple $-expressions, and arriving in a new state* s'*. Actions and new states are found in two ways:*
   (a) *if* gp_i *flag is set to 1 - by applying crossover/mutation (in the form of send and receive operating on LTS trees) to obtain a new state* s'*. A corresponding action between s and* s' *will be labeled as observationally congruent* ε *with neutral cost 0.*
   (b) *if* gp_i *flag is set to 0 - by applying inference rules of LTS to a state.*
   Each simple $-expression in actions is labeled
      - *by its name if simple $-expression belongs to* $A_i \cup \Omega_i$ *and width and depth* b_i,k_i *are not exceeded,*
      - *is renamed by strongly congruent* ε *with estimated cost if flag strongcong* $= 1$ *(default: renamed by weakly congruent* ε *with a neutral (zero) cost, strongcong =* 0*) if width* b_i *or depth* k_i *are exceeded /* hiding actions outside of the agent's width or depth search horizon, however not ignoring, but estimating their costs */.*

   For each new state s' *check whether width/depth of the tree is exceeded, and whether it is a goal state. If so,* s' *becomes the leaf of the tree (for the current loop cycle), and no new actions are generated, otherwise continue to build the tree. If* s' *is a goal state, label it as a goal state.*

4. **Examine Phase**
   (:= $(exam\ x_i[t])$ < ($\sqcup$ *cond_sel_exam* ($\circ$ $\overline{cond\_sel\_exam}$ *exam_body*)) >)*: where exam_body prunes the search tree by selecting paths with minimal cost in cost choices and with maximal cost in adversary choices. Ties are broken randomly. In optimization, simple $-expressions belonging to* $A_i - \Omega_i$ *treat as observationally congruent* ε *with neutral cost (typically, equal to 0 like e.g., for a standard cost function) /* hiding agent's actions outside of its interests by ignoring their cost */.*

5. **Execute Phase**
   (:= $(exec\ (k\Omega_i[t]\ x_i[t]))$ *exec_body): where exec_body =*
   ($\circ$ ($\sqcup$ ($\circ$ ($n_i = 0$)($\overline{goal\_reached}$)($current\_node = leaf\_node\_with\_min\_costs$)) ($\circ$ ($n_i = 0$)($goal\_reached$)($execute(x_i[t])$)($current\_node = leaf\_node$))
   ($\circ$ ($\overline{n_i = 0}$)($execute\_n_i\_steps(x_i[t])$)
   ($current\_node = node\_after\_n_i\_actions$))) $update\_loop$)
   /* If $n_i = 0$ (offline search) and no goal state has been reached in the Select/Examine phase there will be no execution in this cycle. Pick up the most promising leaf node of the tree (with minimal cost) for expansion, i.e., make it a current node (root of the subtree expanded in the next cycle of the loop appended to an existing tree from the select phase, i.e., pruning will be invalidated to accommodate eventual corrections after cost updates). If $n_i = 0$ (offline search) and a goal state has been reached in the Select/Examine phase, execute optimal

$x_i[t]$ up to the leaf node using a tree constructed and pruned in the Select/Examine phase, or use LTS inference rules otherwise (for $gp = 1$). Make the leaf node a current node for a possible expansion (if it is not a goal state - it will be a root of a new tree) in the next cycle of the loop. If $n_i \neq 0$ (online search), execute optimal $x_i[t]$ up to at most n_i steps. Make the last state a current state - the root of the tree expanded in the next cycle of the loop. In execution simple $-expressions belonging to $\Omega_i - A_i$ will be executed by other agents.*/
The update_loop by default does nothing (executes silently ε with no cost) if update flag is reset. Otherwise if update = 1, then it gets from the user potentially all possible updates, e.g., new values of b_i, k_i, n_i and other parameters of $k\Omega[t]$, including costs of simple $-expressions, Ω_i, $goal_i$. If update = 1 and the user does not provide own modifications (including possible overwriting the $k\Omega[t]$), then self-modification will be performed in the following way. If execution was interrupted (by receiving message from the user or environment invalidating solution found in the Select/Examine phase), then $n_i = 10$ if $n_i = \infty$, or $n_i = n_i - 1$ if $n_i \neq 0$, or $k_i = 10$ if $n_i = 0, k_i = \infty$, or $k_i = k_i - 1$ if $n_i = 0, k_i \neq \infty$. If execution was not interrupted increase $n_i = n_i + 1$ pending $0 < n_i \leq k_i$. If $n_i = k_i$ increase $k_i = k_i + 1, b_i = b_i + 1$. If cost of search ($\$_{2i}(k\Omega[t])$) larger than a predefined threshold decrease k_i and/or b_i, otherwise increase it. If reinforcement learning was set up $reinf = 1$ in the init phase, then cost of simple $-expressions will be modified by reinforcement learning.

The building of the LTS tree in the select phase for $gp_i = 0$ combines imperative and rule-based/logic styles of programming (we treat clause/production as a user-defined $-expression definition and call it by its name. This is similar to Robert Kowalski's dynamic interpretation of the left side of a clause as the name of procedure and the right side as the body of procedure.).

In the *init* and *exec*/*update* phase, in fact, a new search algorithm can be created (i.e., the old $k\Omega$ can be overwritten), and being completely different from the original $k\Omega$-search. The original $k\Omega$-search in self-modication changes the values of its control parameters mostly, i.e., k, n, b, but it could modify also *goal*, *sel*, *exam*, *exec* and $.

Note that all parameters k_i, n_i, Ω_i, $\$_i$, A_i, and $\mathcal{A}$ can evolve in successive loop iterations. They are defined as the part of the *init* phase, and modified/updated at the end of the Execute phase *exec*. Note that they are associated with a specific choice of the meta-system: a $k\Omega$-optimization search. For another meta-system, different control parameters are possible.

More details on the $k\Omega$-search, including inference rules of the Labeled Transition System, observation and strong bisimulations and congruences, necessary and sufficient conditions to solve optimization, search optimization and total optimization problems, illustrating examples (including simulation by $k\Omega$-optimization of A*, Minimax, Traveling Salesman Problem, and other typical search algorithms), the details of implementations and applications, can be found in (Eberbach, 2005*a*, 2007).

4. The $-Calculus Expressiveness and its Support to Solve TM Undecidable Problems

To deal with undecidability, the $-calculus uses all three principles from introductory section: the infinity, interaction, and evolution principles:

- *infinity* - because of the infinity of the indexing set I in the $-calculus operators, it is clear that the $-calculus derives its expressiveness mostly from the *infinity* principle.
- *interaction* - if to assume that simple $-expressions may represent oracles, then the $-calculus can represent the interaction principle. Then we define an equivalent of the oracle as a user defined simple $-expression, that somehow in the manner of the "black-box" solves unsolvable problems (however, we do not know how).
- *evolution* - the $k\Omega$-optimization may be evolved to a new (and hopefully) more powerful problem solving method

It is easier and "cleaner" to think about implementation of unbounded (infinitary) concepts, than about implementation of oracles. The implementation of scalable computers (e.g., scalable massively parallel computers or unbounded growth of Internet) allows to think about a reasonable approximation of the implementation of *infinity* (and, in particular, the π-calculus, or the $-calculus). At this point, it is not clear how to implement oracles (as Turing stated *an oracle cannot be a machine*, i.e., implementable by mechanical means), and as the result, the models based on them. One of potential implementation of oracles could be an infinite lookup table with stored all results of the decision for TM. The quite different story is how to initialize such infinite lookup table and how to search it effectively using for instance hash indices or B+ trees.

The expressiveness of the $-calculus is not worse than the expressiveness of Turing Machines, i.e., it is straightforward to show how to encode λ-calculus (Church, 1936, 1941) in $-calculus (Eberbach, 2006, 2007). In (Eberbach, 2005*a*) it has been demonstrated, how some other models of computation, more expressive than Turing Machines, can be simulated in the $-calculus. This includes the π-calculus, Interaction Machines, cellular automata, neural networks, and random automata networks.

It is interesting that the $-calculus can solve in the limit the halting problem of the Universal Turing Machine, and approximate the solution of the halting/optimization problem of the $-calculus. This is a very interesting result, because, if correct, besides the $-calculus, it may suggest that a self-modifying program using infinitary means may approximate the solution of its own decision (halting or optimization) problem.

4.1. Solving the Turing Machine Halting Problem and Approximating the Universal Search Algorithm

The results from Eberbach (2006, 2007) justify that the $-calculus is more expressive than the TM, and may represent non-algorithmic computation. In Eberbach (2006, 2007) three ways how the $-calculus solves the halting problem of the Universal Turing Machine using either an *infinity, evolution, or interaction principle*.

Theorem 4.1 (On solution of the halting problem of UTM by $-calculus (Eberbach, 2006, 2007))**.** *The halting problem for the Universal Turing Machine is solvable by the $-calculus.*

Proof. (Outline): In the infinity principle $-calculus taking an instance of TM code and its input runs an infinite number of steps of TM (same idea like for example in Infinite Time TM). Of course, in infinity the answer for halting will be yes or no. It is a matter of philosophical discussion whether

getting a definitive answer in infinity constitutes an answer at all. In the interaction principle, $-calculus $k\Omega$-search gets and answer from oracle, and in the evolution principle, the TM is evolved to TM with oracle. Assuming that a TM with an oracle can be encoded as a nonrecursive function ($-expresion) using a binary alphabet and an ordinary TM can be (of course) encoded as a binary string, thus a simple binary mutation changing one binary input to another can convert by chance an ordinary Turing Machine to the Turing Machine with an oracle (Turing, 1939). The probability of such event is very small, and because nobody has implemented the Turing Machine with an oracle (although we know what its transition function may look like - see, e.g. (Kozen, 1997)). We may have the big problem with the recognition of this encoding, thus even if theoretically possible, so far nobody has been able to detect the potential existence of the Turing Machine with an oracle. □

We know from the theory of computation that the search for the universal algorithm is a futile effort. If it was not so, such algorithm could be used immediately to solve the halting problem of TM.

We will show how the $-calculus can help potentially for the solution of the best search algorithm by approximating it. The best search algorithm will be in the sense of the optimum: finding the best-quality solutions/search algorithms for the all possible problems. The trouble and undecidability is caused by the requirement to cover exactly *all* possible problems. The number of possible algorithms is enumerable, however, the number of possible problems is infinite, but not enumerable (problems/languages are all possible subsets of all algorithms), see, e.g., (Hopcroft *et al.*, 2001).

Theorem 4.2 (On approximating the universal search algorithm (Eberbach, 2006, 2007))**.** *The $k\Omega$-optimization taking itself as its input will converge with an arbitrarily small error in finite time to the universal search algorithm if search is complete and elitist selection strategy is used.*

Proof. (Outline): In the above approach completeness guarantees that no solution will be missed and elitist selection allows to preserve the best solution found so far. In other words, the $k\Omega$-search taking as its input itself, produces in the finite time better versions of itself and in infinity reaches the optimum (pending that it exists). This allows to approximate the best search algorithm in the finite time. In particular, this can be used for approximated solution of the UTM halting problem. □

In such way progress in mathematics or computer science (both being undecidable) is done. Proving all theorems (existing and not derived yet) in mathematics or computer science is impossible (the Entscheidungsproblem - Hilbert's decision problem is undecidable (Whitehead & Russell, 1910, 1912, 1913; Turing, 1937; Hopcroft *et al.*, 2001)). However, in spite of that, new generations of mathematicians and computer scientists work and generate new theorems and improve indefinitely the current state of art of mathematics and computer science. The famous unsolvable Entscheidungsproblem does not prevent scientists from discovering new theorems, improving our knowledge of mathematics, but we will never be able to write all theorems (unless to wait for eternity).

4.2. Deciding the Diagonalization Language, Nontrivial Properties, Solving Post Correspondence Problem and Busy Beaver Problem

In (Eberbach, 2003) several open problems have been posed. We will present solutions to some of them.

The diagonalization language L_d is an example of the language that is believed even tougher than L_u, i.e., language of UTM accepting words w for arbitrary TM M. L_d is non-RE, i.e., it does not exist any TM accepting it.

Definition 4.1. The diagonalization language L_d consists of all strings w such that TM M whose code is w does not accept when given w as input.

The existence of diagonalization language that cannot be accepted by any TM is proven by diagonalization table with "dummy", i.e., not real/true values. Of course, there are many diagonalization language encodings possible that depend how transitions of TMs are encoded. This means that there are infinitely many different L_d language instances (but nobody wrote a specific example of L_d). Solving the halting problem of UTM, can be used for a "constructive" proof of the diagonalization language decidability demonstrating all strings belonging to the language.

Theorem 4.3 (Deciding asymptotically the diagonalization language)**.** *The diagonalization language L_d is \$-calculus asymptotically decidable pending that language of halting UTM L_u is decidable.*

Proof. By Theorem 4.1 we fill a characteristic vector for each TM in diagonalization table, i.e., for each TM M_i and for each its input string w_j, $i, j = 0, 1, 2, ...$, we write 1 if w_j is accepted by M_i and 0 otherwise. This is always possible pending that halting of UTM is solvable. For each M_i we look at w_i (the diagonal value) and flip its bit to opposite value. Now for each w_i, $i = 0, 1, 2, ...$, we construct a characteristic vector for L_d. For each accepted string w_i we construct a finite automaton FA_i accepting exactly one word w_i (always trivially possible in finite time). We construct an infinite parallel composition - \$-calculus \$-expression $(\|_i\ FA_i)$ and put to it an arbitrary input string w. If $(\|_i\ FA_i)$ accepts w (i.e., one of FA accepts) then L_d accepts. If no FA accepts then L_d does not accept either. □

In such a way we can decide in \$-calculus a language L_d that is not possible to decide in the TM model.

In an analogous way we can decide other TM unsolvable languages/problems.

The language L_{ne} consisting of all binary encoded TMs whose language is not empty, i.e., $L_{ne} = \{M \mid L(M) \neq \emptyset\}$ is known to be recursively enumerable but not recursive, and its complement language $L_e = \{M \mid L(M) = \emptyset\}$ consisting of all binary encoded TMs whose language is empty is known to be non recursively enumerable.

Theorem 4.4 (Deciding asymptotically L_{ne} and L_e)**.** *The languages L_{ne} and L_e are \$-calculus asymptotically decidable pending that language of halting UTM L_u is decidable.*

Proof. It is enough to look at the diagonalization table found after solving UTM L_u halting problem. The rows containing at least one 1 allow to decide L_{ne}, and complementary rows consisting of all 0s allow to decide L_e. □

We can solve nontrivial properties (nonempty proper subsets of all RE languages), i.e., to prove solvability of Rice theorem (Rice, 1953) using hypercomputation.

Theorem 4.5 (Deciding asymptotically nontrivial properties)**.** *Every nontrivial property is $-calculus asymptotically decidable pending that language of halting UTM L_u is decidable.*

Proof. We identify the proper subset of the rows from the diagonalization table satisfying a specific nonempty property. It is necessary to translate specific property in terms of corresponding 1s and 0s in characteristic vectors for each TM M_i (e.g., to translate which rows correspond to the empty language, a finite language, a regular language, context-free language, context-sensitive language). □

It is also possible to decide PCP and Busy Beaver Problem.

Definition 4.2. The TM undecidable Post Correspondence Problem (PCP) (Post, 1946) asks, given two lists of the same number of strings over the same alphabet, whether we can pick a sequence of corresponding strings from the two lists and form the same string by concatenation.

Theorem 4.6 (Deciding asymptotically PCP)**.** *The Post Correspondence Problem is asymptotically decidable pending that language of halting UTM L_u is decidable.*

Proof. As the halting or terminal state of the TM solving PCP we put the condition whether both lists produce the same string. □

Definition 4.3. The TM undecidable Busy Beaver Problem (BBP) (Rado, 1962) considers a deterministic 1-tape Turing machine with unary alphabet $\{1\}$ and tape alphabet $\{1, B\}$, where B represents the tape blank symbol. TM starts with an initial empty tape and accepts by halting. For the arbitrary number of states $n = 0, 1, 2, ...$ TM tries to compute two functions: the maximum number of 1s written on tape before halting (known as the busy beaver function $\Sigma(n)$ and the maximum number of steps before halting (known as the maximum shift function $S(n)$.

Theorem 4.7 (Deciding asymptotically BBP)**.** *The Busy Beaver Problem is asymptotically decidable pending that language of halting UTM L_u is decidable.*

Proof. As the halting state of the TM solving BBP we put the condition whether $\Sigma(n)$ and $S(n)$ have been computed. □

5. Conclusions

In the paper some hypercomputation solutions of Turing Machine unsolvable problems have been presented. We demonstrate that the solution of the halting problem is like to solve polynomially one NP-complete problem to resolve famous dilemma $\mathcal{P} ? = \mathcal{NP}$, i.e., it breaks the whole hierarchical puzzle of unsolvability. Namely, solving the halting problem of UTM is pivotal to solve many other Turing Machine unsolvable problems, including to decide the diagonalization language, nontrivial properties, PCP and BBP.

However, hypercomputational models are still not well researched and many scientists vigorously oppose the idea that computations going beyond Turing Machine are possible at all. Very

little is known about the hierarchy (or at least relations) between hypercomputional models. We do not know enough about the implementability of hypercomputation at least at the level similar to quantum computing or biomolecular computing implementations. We do not know the limits of computability in hypercomputational models. We do not know whether such limits exist at all. In other words, hypercomputation is in the situation similar like Turing, Church and Gödel were in 1930s before the whole digital computers explosive growth started. Whether these hopes and fears about hypercomputation will be materialized is a quite different story.

References

Aho, Alfred V. (2011). Ubiquity symposium: Computation and computational thinking. *Ubiquity*.

Burgin, M. S. (2004). *Super-Recursive Algorithms (Monographs in Computer Science)*. SpringerVerlag.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics* **58**(2), 345–363.

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton, N.J., Princeton University Press. New York, NY, USA.

Cooper, B. (2012). Turing's titanic machine?. *Commun. ACM* **55**(3), 74–83.

Eberbach, E. (1997). A generic tool for distributed AI with matching as message passing. In: *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*. pp. 11–18.

Eberbach, E. (2003). Is entscheidungsproblem solvable? beyond undecidability of Turing machines and its consequence for computer science and mathematics. In: *(ed. J. C. Misra) Computational Mathematics, Modelling and Algorithms*. pp. 1–32. Narosa Publishing House, New Delhi.

Eberbach, E. (2005*a*). $-Calculus of bounded rational agents: Flexible optimization as search under bounded resources in interactive systems. *Fundam. Inf.* **68**(1-2), 47–102.

Eberbach, E. (2005*b*). Toward a theory of evolutionary computation. *Biosystems* **82**(1), 1 – 19.

Eberbach, E. (2006). Expressiveness of the π-Calculus and the $-Calculus. In: *Proc. 2006 World Congress in Comp. Sci., Comp. Eng., & Applied Computing, The 2006 Intern. Conf. on Foundations of Computer Science FCS'06, Las Vegas, Nevada*. pp. 24–30.

Eberbach, E. (2007). The $-calculus process algebra for problem solving: A paradigmatic shift in handling hard computational problems. *Theoretical Computer Science* **383**(23), 200 – 243. Complexity of Algorithms and Computations.

Eberbach, E. and M. Burgin (2009). On foundations of evolutionary computation: An evolutionary automata approach. In: *(ed.Hongwei Mo): Handbook of Research on Artificial Immune Systems and Natural Computing: Applying Complex Adaptive Technologies, Section II: Natural Computing, Section II.1: Evolutionary Computing, Chapter XVI, Medical Information Science Reference/IGI Global, Hershey*. pp. 342–360. New York.

Eberbach, E., D. Goldin and P. Wegner (2004). Turings ideas and models of computation. In: *Alan Turing: Life and Legacy of a Great Thinker* (Christof Teuscher, Ed.). pp. 159–194. Springer Berlin Heidelberg.

Eberbach, Eugene and Peter Wegner (2003). Beyond Turing machines. *Bulletin of the EATCS* pp. 279–304.

Hopcroft, J. E., R. Motwani and J. D. Ullman (2001). Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News* **32**(1), 60–65.

Horvitz, E. and S. Zilberstein (2001). Computational tradeoffs under bounded resources. *Artificial Intelligence* **126**(12), 1 – 4. Tradeoffs under Bounded Resources.

Kozen, D. C. (1997). *Automata and Computability*. Springer-Verlag.

Milner, R. (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press. New York, NY, USA.

Milner, R., J. Parrow and D. Walker (1992). A calculus of mobile processes, i. *Information and Computation* **100**(1), 1 – 40.

Post, E. (1946). A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* **52**(4), 264–268.

Rado, T. (1962). On non-computable functions. *Bell System Technical Journal* **41**(3), 877–884.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* **74**, 358–366.

Russell, S. and P. Norvig (2002). *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Syropoulos, A. (2007). *Hypercomputation: Computing Beyond the Church-Turing Barrier (Monographs in Computer Science)*. Springer-Verlag New York, Inc.. Secaucus, NJ, USA.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society* **s2-42**(1), 230–265.

Turing, A. M. (1939). Systems of logic based on ordinals. *Proceedings of the London Mathematical Society* **s2-45**(1), 161–228.

Wegner, P., E. Eberbach and M. Burgin (2012). Computational completeness of interaction machines and Turing machines. In: *Turing-100* (Andrei Voronkov, Ed.). Vol. 10 of *EPiC Series*. EasyChair. pp. 405–414.

Whitehead, A. N. and B. Russell (1910, 1912, 1913). *Principia mathematica, vol.1, 1910, vol.2, 1912, vol.3, 1913*. Cambridge Univ. Press. Cambridge.