A Uniform Problem Solving in the Cognitive Algebra of Bounded Rational Agents

Eugene Eberbach

Dept. of Engineering and Science, Rensselaer Polytechnic Institute
275 Windsor Street, Hartford, CT 06120, USA
eberbe@rpi.edu

Abstract. The $-calculus cognitive process algebra for problem solving provides the support for automatic problem solving and targets intractable and undecidable problems. Consistent with the ideas of anytime algorithms, $-calculus applies the cost performance measures to converge to optimal solutions with minimal problem solving costs. In the paper, we concentrate on a uniform problem solving and its implementation aspects illustrated on two benchmarks from concurrency and machine learning areas.

1 Introduction

The paper presents a theory of computation for automatic problem solving based on process algebras, utility theory and anytime algorithms. In particular, we try to formalize AI, based on classical (but unformalized) AI textbook by Russell and Norvig on meta-search algorithms and bounded rational agents [8]. Such unifying theory for AI does not exist so far yet. Resource-based reasoning [6, 8] called also anytime algorithms, trading off the quality of solutions for the amount of resources used, seems to be a particularly well suited for such a new AI framework. Process algebras [7], currently the most mature approach to concurrent and distributed systems, seem to be the appropriate way to formalize multiagent systems, and to span AI with the rest of computer science.

In the paper, we describe very briefly the $-calculus algebra of bounded rational agents as a proposal of the unifying framework for AI [3–5]. The $-calculus can be used in the same uniform way for search, planning, evolution, learning, and problem solving under bounded resources in dynamic and uncertain environments. In the paper, we concentrate on automatic problem solving and its implementation aspects illustrated by examples from two divergent areas.

2 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 main difference compared to other computational theories). The $-calculus is a cognitive process algebra of Bounded Rational

G. Jezic et al. (eds.), *Agent and Multi-Agent Systems: Technologies and Applications*,
Advances in Intelligent Systems and Computing 296,
DOI: 10.1007/978-3-319-07650-8_13,

Agents for interactive problem solving targeting intractable and undecidable problems [3–5]. 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 [6, 8]. 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 [7] is believed to be the most mature approach for concurrent systems. Although being a new approach, the $-calculus has found already several applications including DSL languages to control NAVY Autonomous Vehicles (AUVs) (for more deatils look at [3–5]).

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.

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

$$
\begin{array}{lll}
\alpha ::= & (\$_{i\in I}\ P_i) & \text{cost} \\
| & (\rightarrow_{i\in I}\ c\ P_i) & \text{send } P_i \text{ with evaluation through channel } c \\
| & (\leftarrow_{i\in I}\ c\ X_i) & \text{receive } X_i \text{ from channel } c \\
| & ('_{i\in I}\ P_i) & \text{suppress evaluation of } P_i \\
| & (a_{i\in I}\ P_i) & \text{defined call of simple \$-expression } a \text{ with parameters } P_i, \text{ and} \\
 & & \text{and its optional associated definition } (:=\ (a_{i\in I}\ X_i)\ <R>) \\
 & & \text{with body } R \text{ evaluated atomically} \\
| & (\bar{a}_{i\in I}\ P_i) & \text{negation of defined call of simple \$-expression } a
\end{array}
$$

$$
\begin{array}{lll}
P ::= & (\circ_{i\in I}\ \alpha\ P_i) & \text{sequential composition} \\
| & (\parallel_{i\in I}\ P_i) & \text{parallel composition} \\
| & (\uplus_{i\in I}\ P_i) & \text{cost choice} \\
| & (\uplus_{i\in I}\ P_i) & \text{adversary choice} \\
| & (\sqcup_{i\in I}\ P_i) & \text{general choice} \\
| & (f_{i\in I}\ P_i) & \text{defined process call } f \text{ with parameters } P_i, \text{ and its associated} \\
 & & \text{definition } (:=\ (f_{i\in I}\ X_i)\ R) \text{ with body } R \text{ (normally} \\
 & & \text{suppressed); } (^1\ R) \text{ will force evaluation of } R \text{ exactly once}
\end{array}
$$

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. 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* and λ-calculus around a *function*.

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.

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.

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 [3–5].

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 is estimated (i.e., not all information is lost). The above means that $k\Omega$-optimization may be (if certain conditions are satisfied) complete and optimal (see [4]). The building trees (or DAGs, in a general case) is done either by using inference rules from LTS (in the style of AI planners, unification from Prolog, or matching from expert systems), or by using random number generators to generate random sequences of simple $-expressions (in the style of genetic programming), or the user is responsible to define the LTS tree. 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 also 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 can directly minimize search cost. This basic feature, inherited from anytime algorithms, is needed to directly tackle hard optimization problems, and allows solving 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. The b is the branching factor of the search tree, n - the number of steps selected for execution in the execute phase, and p - the number of agents. 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" (i.e., good enough - term coined by Simon [8]) solution, and sometimes (when appropriate conditions are satisified) - the optimal one. This avoids wasted 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. 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 which only sometimes will be the same for all agents (this is analogous to the subjective view of the "objective" world by individuals having possibly different goals and different perception of the universe).

More details on the $k\Omega$-optimization, including the inference rules of the Labeled Transition System, observation and strong bisimulations and congruences, and the standard cost function definition can be found in [3–5].

3 Illustration of Versality and Power of the $k\Omega$-meta Search

3.1 Dining Philosophers - Multi-agent Searching and Planning for a Deadlock-Free and Fair Solution

The Dining Philosophers Problem is a simple abstraction of a typical synchronization problem to allocate mulitple shared reusable resources among several processes in a deadlock and starvation-free manner (for example, an abstraction of the access to I/O devices). It was posed and solved by E. Dijkstra. Scenario: five philosophers are seated around a table. Each philosopher has a plate of spaghetti, which is so slippery that each philosopher needs two forks to eat (sometimes to be more realistic, spaghetti is replaced by rice, and forks by chopsticks). Between each plate is a fork, and if the fork is grabbed, it is not released

until a philosopher finishes to eat. Each philosopher does in cycle: taking forks, eating, releasing forks and thinking. The goal is to provide a solution allowing maximum parallelism where philosophers do not deadlock (otherwise all philosophers starve) and sometimes, additionally fairness is required, for a deadlock-free solution to provide a guarantee that no one will starve. Of course, we know how to solve the dining philosophers. We use this problem for illustration of how $k\Omega$-optimization will arrive at a solution that by minimizing costs will avoid deadlocks (having infinite costs) and starvation (by design - philosophers who ate above average will refrain nicely from competing for forks).

Let's consider problem solving (planning + execution) for dining philosophers providing deadlock-free and fair solution, and expressed as a special case of $k\Omega$-search. Note that the solution subsumes hierarchical (user-defined functions Phil, Fork and Count) and partial-order planning (due to concurrency from process algebra).

The system consists of 5 agents-philosophers, i.e., p=5, which are interested and can observe everything, i.e., Ω is an alphabet of all simple $-expressions, with a standard shared by all philosphers cost function $\$ = \$_1(\$_2(k\Omega[t], \$_3(x[t]))$, where $\$_1$ is an aggregating function in the form of addition, $\$_2(k\Omega[t])$ represents costs of the $k\Omega$-search, and $\$_3(x[t])$ represents the quality of solutions. A strong congruence is used. In other words, payoff is associated with empty/invisible actions ε for finding the plan (complete tree) and/or for executing it. The number of steps in the derivation tree selected for optimization in the examine phase $k = \infty$, the branching factor $b = \infty$, and the number of steps selected for execution in the examine phase $n = 0$, i.e., execution is postponed until the plan is found. Flags $gp = reinf = update = 0$ and $strongcong = 1$. The goal of plan is to find the cheapest deadlock-free and fair solution. The solution takes the form of the tree (truly DAGs) of $-expressions that are pruned and passed to execution phase.

Let's assume that user defined functions $Phil_i$, $Fork_i$, $Count$ are given and they are partially designed only, i.e., $\uplus$ represents unsolved alternatives in the design:

$(:= \ grab2_i \ \langle(\parallel \ (\leftarrow \ p_{i,i} \ f_i) \ (\leftarrow \ p_{(i+4)mod\,5,i} \ f_{(i+4)mod\,5})\rangle)$ -atomic $grab2_i$ def.

$(:= \ Fork_i \ (\circ$ // allow to grab fork f_i by right/left neighbor and receive it back

$(\sqcup \ (\circ \ (\rightarrow \ p_{i,(i+1)mod\,5} \ f_i) \ (\leftarrow \ p_{i,(i+1)mod\,5} \ f_i))$

$(\circ \ (\rightarrow \ p_{i,i} \ f_i) \ (\leftarrow \ p_{i,i} \ f_i))$

)

$Fork_i$ // call recursively $Fork_i$, $i = 0, 1, ..., 4$ process again

))

$(:= \ Count \ (\circ \ (\rightarrow \ ch \ c)$ // send and receive global count c of eatings

$(\leftarrow \ ch \ c)$

$Count$ // call recursively $Count$

))

$(:= \ Phil_i \ (\circ (\leftarrow \ ch \ c)$ //grab global count c of eatings; each i-th phil., $i = 0, ..., 4$
// grabs fork $f_{(i+4)mod\,5}$ through channel $p_{(i+4)mod\,5,i}$ and fork f_i through channel $p_{i,i}$
$(\sqcup \ (\circ \ (\leq \ 5c_i \ c)$ // for fairness: grab forks if you did not eat above average
$(\uplus \ (\circ \ (\leftarrow \ p_{i,i} \ f_i) \ (\leftarrow \ p_{(i+4)mod\,5,i} \ f_{(i+4)mod\,5}))$ // grab right next left fork
$(\circ \ (\leftarrow \ p_{(i+4)mod\,5,i} \ f_{(i+4)mod\,5}) \ (\leftarrow \ p_{i,i} \ f_i))$ // grab left next right fork
$grab2_i$ // grab both forks atomically in parallel, def. call of simple $-expr.
) // in atomic $grab2_i$ all components should not block, else $grab2_i$ will block
eat
$c_i + +$ // increase your private count of eatings; initially all counts 0
$c + +$ // increase global count of eatings
$(\rightarrow \ p_{i,i} \ f_i) \ (\rightarrow \ p_{(i+4)mod\,5,i} \ f_{(i+4)mod\,5})$ //return both forks
$think$) // do the job that philosophers supposed to do
$(> \ 5c_i \ c)$ // for fairness: be nice - do nothing if you ate above average
)
$(\rightarrow \ ch \ c)$ // return global count of eatings through channel ch to $Count$
$Phil_i$ // call recursively $Phil_i$, $i = 0, 1, ..., 4$ process again
))

Total Optimization: The goal will be to minimize costs for $\$ = \$_2 + \$_3$. The empty actions (representing actions not executed yet) have cost being the sum of payoffs for finding the plan/solutions and for execution of plan (for not starving philosophers). Each action has negative payoffs for action planned and executed (represented by function $\$_3$, and costs for searching for the plan and executing it (costs of running $-Ruby interpreter - function $\$_2$). Assume that payoff for finding the plan is 500 (negative cost -500) and payoff for executing plan is 1000 (because it is a reactive never terminating program, it will never be reached and we can ignore it). Each action during planning (select and examine phase) and execution has cost 1 and no payoff (payoff will be paid after end of planning). During execution each action has cost 1 and payoff $2 - 0.02m$, where $m = 0, 1, 2, ...$ represents successive uses of the action. This means that payoffs initially will dominate, but after 100 uses actions will incur only costs.

0. t=0, initialization phase $init$:
   $x[0] = (\circ \ (\| \ Phil_0 \ Phil_1 \ Phil_2 \ Phil_3 \ Phil_4 \ Fork_0 \ Fork_1 \ Fork_2 \ Fork_3 \ Fork_4 \ Count) \ \varepsilon)$
   The initial tree consists of the root state $x[0]$ calling in parallel processes $Phil_i$, $Fork_i$,$i = 0, 1, ..., 4$ and $Count$, and an empty action ε which cost is equal to the estimated payoffs for finding plan and execution 500. Because $x[0]$ is not the goal state (plan and its execution has not been done yet), interpreter goes to the first loop iteration consisting of select, examine, and execute phases.
1. t =1, first loop iteration:
   select phase sel : because $k = \infty$ only one loop iteration will be needed and a complete potential tree of solutions is expanded, i.e., user defined functions $Phil_i$, $Fork_i$ and $Count$ are replaced by their body definitions.
   examine phase $exam$:

Execution is postponed ($n = 0$) until pruning is done. Assume that cost of multiset of parallel actions is equal to maximum of its component costs, thus grabbing in parallel two forks will have cost 1, and grabing sequentially 1+1=2. Some sequential orders like grabbing first left and next right fork by all philosophers will result in deadlock (and infinite costs), thus they will be eliminated by the $-calculus optimization mechanism. Parallel grabbing of both forks will be cheaper than sequential (and deadlock-free), thus design will leave only parallel grabbing in $\uplus$ definition of $Phil_i$.
execute phase *exec* :
Actions are executed, but with gradually decreasing payoffs (corresponding to the span of life of philosophers). Initially it will be cheaper to execute plan, but after 100 cycles for philosphers, costs will decrease payoff for finding plan, and execution should be stopped, because optimum will be lost and it will lead to infinite cost for immortal philosophers. After that the $k\Omega$-search re-initializes for a new problem to solve (e.g., machine learning ID3).

3.2 Learning the Best Decision Tree for a Single Agent

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees, i.e., the ID3 algorithm by Quinlan and its successor C4.5 [8]. ID3 performs a simple hill-climbing search through the hypothesis space of possible decision trees using as an evaluation function the Shannon-based information gain measure. ID3 maintains only a single current hypothesis as it searches through the space of decision trees. The search space is exponential, i.e., the problem is intractable. To alleviate that, ID3 picks up the attribute with the maximum information gain which leads to the shortest tree (i.e., it uses the Occam razor principle). The information gain $Gain(S, A)$ of an attribute A relative to a collection of examples S is defined as $Gain(S, A) = Entropy(S) - \Sigma_{v \in Values(A)} Entropy(S_v)|S_v|/|S|$, where $Entropy(S) = \Sigma_i - p_i log_2 p_i$.

Let's consider problem solving (learning + classification) for ID3 expressed as a special case of $k\Omega$-search finding the shortest classification tree by minimizing the sum of negative gains, i.e., maximizing the sum of positive gains.

The system consists again of one agent only, i.e., p=1, which is interested only in information gain for alphabet $A = \{a_i, a_{ij}\}, i, j = 1, 2$, i.e., $\Omega = A$, i.e., costs of other actions are ignored (being neutral - in this case having zero cost), and it uses a standard cost function \$ = \$$_3$, where \$$_3$ represents the quality of solutions in the form of cummulative negative information gains - payoff in $-calculus. In other words, total optimization is not performed, only regular optimization like in the original ID3 to illustrate a regular optimization, not that the total optimization is not desirable. A weak congruence is used. In other words, empty actions have zero cost. The number of steps in the derivation tree selected for optimization in the examine phase $k = 2$, the branching factor $b = \infty$, and the number of steps selected for execution in the examine phase $n = 0$, i.e., execution is postponed until learning is over. Flags $gp = reinf = update = strongcong = 0$. The goal of learning/classification is to minimize the

sum of negative information gains. The machine learning takes the form of the tree of $-expressions that are built in the select phase, pruned in the examine phase and passed to execution phase for classification work. Data are split into training and test data as usual. Let's assume for simplicity that we have only one decision attribute and two input attributes a_1 and a_2 with data taking two possible values on them denoted by a_{11},a_{12},a_{21},a_{22}. Let assume that cost of actions is equal to entropy of data associated with this action, i.e., $\$(a_i) = -Entropy(a_i)$,$\$(a_{ij}) = Entropy(a_{ij})$,$i,j = 1,2$.

OPTIMIZATION: The goal will be to minimize the sum of costs (negative gains).

0. t=0, initialization phase $init:\ \ S_0 = \varepsilon_0$
   The initial tree consists of an empty action ε_0 representing a missing classification tree of which cost is ignored (a weak congruence). Because S_0 is not the goal state, the first loop iteration consisting of select, examine, and execute phases replaces an invisible ε_0 two steps deep ($k = 2$) by all offsprings $b = \infty$.
1. t=1, first loop iteration:
   select phase sel :
   $\varepsilon_0 = (\uplus\ (\circ\ a_1(\sqcup\ (\circ\ a_{11}\ \varepsilon_{11})(\circ\ a_{12}\ \varepsilon_{12})))\ (\circ\ a_2(\sqcup\ (\circ\ a_{21}\ \varepsilon_{21})(\circ\ a_{22}\ \varepsilon_{22}))))$
   examine phase $exam:\ \$(S_0) = \$(\varepsilon_0) =$
   $= min(\$(a_1) + p_{11}\$(a_{11}) + p_{12}\$(a_{12}), \$(a_2) + p_{21}\$(a_{21}) + p_{22}\$(a_{22}))$
   Let's assume that attribute a_1 was selected, i.e., $-expresion starting from a_1 is cheaper. Note that due to appropriate definition of the standard cost function (for complete definition see [37,38]) this is a negative gain from ID3. This confirms that $-calculus cost function is defined in a reasonable way (at least from the point of ID3). Note that no estimates of future solutions are used (weak congruence - greedy hill climbing search). Execution is postponed ($n = 0$), and follow-up ε_{11} and ε_{12} will be selected for expansion in the next loop iteration.
2. t = 2, second loop iteration:
   select phase $sel:\ \ \varepsilon_{11} = (\circ\ a_2\ (\sqcup\ (\circ\ a_{21}\ \varepsilon_{21})(\circ\ a_{22}\ \varepsilon_{22})))$
   Let's assume that ε_{12} has data from one class only, thus this is the leaf node - no further splitting of training data is required.
   examine phase $exam:$ nothing to optimize/prune - all attributes were used in the path or the leaf node contained sample data from one class of the decision attribute. Thus the end of the learning phase and the shortest decision tree is designated for the execution:
   execute phase $exec:$
   Test data are classified by the decision tree left from the select/examine phases. After that the $k\Omega$-search re-initializes for the new problem to solve.

Note that we can change for example values of k (considering a few attributes in parallel), b, n and optimization to total optimization, then this will be related, but not ID3 algorithm any more. This is the biggest advantage and flexibility of $-calculus problem solving. It can modify "on fly" existing algorithms and design new algorithms, and not simulation of ID3 alone (as obviously ID3 can be expressed in any powerful enough programming language).

Other examples of problem solving by search simulated by $k\Omega$-optimization, including A*, minimax, dynamic programming, TSP, the halting problem of UTM, neural networks, cellular automata, simulation of λ-calculus or π-calculus can be found in [3–5]. This illustrates the power and versatility of the $k\Omega$-search meta-algorithm.

4 Implementation in $-cala and $-calculisp

We started to implement $-calculus framework in $-cala [1] and $-Calculisp [9] to gain necessary experience, to solve some theoretical problems, and as a prelimnary step in the main implementation in Ruby. Ruby is a powerful and versatile object-oriented language that has at the same time a flexibility to that of LISP in dealing with code and meta-code management. We selected Scala, because of very concise implementation of related π-calculus process algebra in Scala [2]. On the other hand, Common Lisp seems be a natural contrcandidate to Ruby. Initially, we implemented in $-cala and $-Calculisp the same examples as $k\Omega$-search: n-puzzle, A* and TSP, and next added dining philosphers and ID3. It looks that both Scala and Lisp have similar metaprogramming capabilities as Ruby, however they are less popular at this moment (and, most likely, in the future) and seem to have less powerful programming environment.

$-Calculisp is implemented in Common Lisp using a combination of functional programming, the powerful Common Lisp macro meta-programming system, and the Common Lisp Object System. It implements an object tree of $-nodes, and $-calculus operators for manipulating the $-node object tree. $-nodes are sub-classed and their methods extended to match a given problem domain or algorithm. Once the problem domain is defined, $-operators and the $k\Omega$ construct are used to implement the bulk of the algorithm.

$-cala is implemented in Scala - an object functional language with an extensible syntax that makes it an ideal host language [2]. $-cala attempts to provide all the primitives from $-calculus as objects. The programmer uses syntax similar to $-calculus to create an abstract syntax tree (AST) and then the AST is interpreted as $k\Omega$-search algorithm. As a consequence a $-cala program corresponds closely to a $-calculus program.

5 Conclusions and Further Work

In the paper, we presented a uniform problem solving by the same $k\Omega$-meta-search algorithm, and its preliminary implementation in $-cala and $-Calculisp. Using the same $k\Omega$-search we can investigate completeness, optimality, search optimality and total optimality of uninformed and informed search methods for adversary and cooperating agents (see [3–5]). Both sequential and concurrent (partial-order) planning, conditional planning, hierarchical planning can be expressed and investigated (their completeness and optimality) in the same uniform way as $-calculus search. We can quite easily express Dynamic Bayesian Networks - DBNs (using sequential composition for conditional probabilities,

and general choice operator combining varius choices weighted by probabilities). Bayesian learning (e.g., MAP/ML [8]) has a natural mapping by $k\Omega$-search tree.

We want to use $-calculus to experiment and combine various approaches. Even simple modifications with simulation of evolutionary algorithms, like changing for example values of n, k, b will lead to algorithms that are not *sensu stricto* evolutionary algorithms any more. Similarly, we would be able to experiment with algorithms based on existing algorithms, but they will not be classical A* or minimax any more. For example, A* with n!=0 leads to "on-line A*", or A* using general choice (besides cost choice) will lead to "probabilistic, fuzzy set or rough set A*" [5]; fixing b leads to SMA*. Additionally, using total optimization leads to algorithms that stop to be classical A*, minimax or evolutionary algorithms (they even do not have corresponding names in the literature). Of course, you can try to do that in any language, but it will be much easier to do in a language designed for it.

The author, besides earlier work on GPS by Simon and Newell and Koza's GPPS is not aware about any related approach so far. Thus the approach is totally novel and if successful can be compared only in its significance with the role of the Turing Machine model in computer science.

References

1. Ansari, B.: $-cala: An Embedded Programming Language Based on $-Calculus and Scala, Master Thesis, Rensselaer Polytechnic Institute at Hartford (2013)
2. Cremet, V., Odersky, M.: PiLib: A Hosted Language for Pi-Calculus Style Concurrency. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 180–195. Springer, Heidelberg (2004)
3. Eberbach, E.: $-Calculus of Bounded Rational Agents: Flexible Optimization as Search under Bounded Resources in Interactive Systems. Fundamenta Informaticae 68(1-2), 47–102 (2005)
4. Eberbach, E.: The $-Calculus Process Algebra for Problem Solving: A Paradigmatic Shift in Handling Hard Computational Problems. Theoretical Computer Science 383(2-3), 200–243 (2007), doi:dx.doi.org/10.1016/j.tcs.2007.04.012
5. Eberbach, E.: Approximate Reasoning in the Algebra of Bounded Rational Agents. Intern. Journal of Approximate Reasoning 49(2), 316–330 (2006), doi:dx.doi.org/10.1016/j.ijar.2006.09.014
6. Horvitz, E., Zilberstein, S.: Computational Tradeoffs under Bounded Resources. Artificial Intelligence 126, 1–196 (2001)
7. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I & II. Information and Computation 100, 1–77 (1992)
8. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall (1995) (2010)
9. Smith, J.: $-Calculisp: an Implementation of $-Calculus Process Algebra in Common Lisp. Master Project, Rensselaer Polytechnic Institute at Hartford (2013)