Activity-based Credit Assignment (ACA) in Hierarchical Simulation

Alexandre Muzy, Bernard P. Zeigler

Activity-based Credit Assignment (ACA) in Hierarchical Simulation

Alexandre Muzy*, Bernard P. Zeigler**

* LISA UMR CNRS 6240, Università di Corsica – Pasquale Paoli, Campus Grossetti, BP 52, 20250 Corti - France, Email: a.muzy@univ-corse.fr.

** Chief Scientist, RTSync Corp, 530 Bartow Drive Suite A Sierra Vista, AZ 85635, United-States of America, Email: zeigler@rtsync.com.

The use of activity-based credit assignment (ACA) for the automatic evaluation and selection of candidate components of systems is considered here. The whole process consists of a precise automatic structured specification of systems. Mathematical definitions and algorithms are provided. ACA converges on good components/compositions faster than repository-based random search. As systems constitute a vast class of problems to be specified by a modeler, this automatic composition of systems opens new research perspectives. The paper also places ACA within the context of existing approaches to credit assignment in classifier systems.

1 Introduction

Among the plethora of learning methods in *machine Learning* [SB98],*supervised learning* focuses on the determination of optimal functions and *reinforcement learning* on the determination of optimal actions on world's state. In supervised learning, considering a set of inputs/outputs (I/O) pairs: $\{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, a learning algorithm will determine, in a set of functions G, a function $g : X \to Y$, with X the input set, and Y the output set. In reinforcement learning, a learning algorithm will determine an optimal action-value function: $Q^*(s, a) = \underset{\pi}{max} Q^\pi(s, a)$, where π is a policy[1], s is the state of the world, and a the action modifying the world's state s.

In *systems theory* [BPZ00] the hierarchical specification of the dynamic of a *system* is achieved from the *I/O frame*: $\mathcal{IOF} =< X, Y >$, where X and Y have been defined previously, passing through the *I/O function*: $\mathcal{IOFO} =< X, \Omega, Y, F >$, where X and Y have been defined previously, $F = \{f_1, f_2, f_3, ...\}$ is the set of I/O functions, and Ω is the input segment set, and ending to *structured systems*: $\mathcal{S} =< T, X, \Omega, Q, Y, \Delta, \Lambda >$, where X, Y, Ω have been defined previously, T is the time base, Q is the set of states, $\Delta : Q \times \Omega \to Q$ is the global state transition function, and $\Lambda : Q \times X \to Y$ is the output function. As pinpointed by George Klir in [KE03], systems constitute a large class of problems.

[1]"A policy, π, is a mapping from each state, $s \in S$, and action, $a \in A$, to the probability $\pi(s, a)$ of taking action a when [the world is] in state s"[SB98].

The *main idea* of this paper is, at a high level of control, to automatically and hierarchically specify the components of a structured system, using a discrete-event system specification (DEVS) and supervised learning. To achieve this goal, *activity* of components and compositions [MZ08, MTV$^+$10] is used to select target components. Activity consists of a quantitative measure of both external and internal event occurrences in components. This usage amount of components is correlated with the *evaluation* of compositions. The basic idea is to assign *credit* based on the correlation of activity of a component and the score of the composition it is in. This is referred to the *credit assignment problem.*

The *main advantages of activity-based credit assignment (ACA)* can be summarized as follows:

1. Applies to any DEVS hierarchical components and compositions within any DEVS Experimental Frame,

2. Evaluates components/compositions at each level of the hierarchy,

3. Converges on good components/compositions faster than repository-based random search,

4. Automatically synthesises systems from a model-base thus enabling reusability of highly rated components in compositions.

Section 2 describes the main definitions and the credit assignment problem and the underlying search algorithms. Section 3 validates the whole approach drawing the main simulation statistics. Section 4 compares ACA with other credit assignment approaches. Finally Section 5 concludes and presents some research perpectives.

2 Credit Assignment Problem

We define here the activity, credit and algorithm used for automatically selecting high performance components and compositions of systems.

2.1 Activity and ACA Definitions

Activity of a composition is defined recursively as the sum of activity of its lower level components. Running many trials allows finding progressively "good" components.

2.1.1 Definition of Activity

A component corresponds to a Discrete Event System Specification, which is a tuple, denoted as $c =< X, Y, S, \delta, \lambda, \tau >$, where X is the *set of input values*, Y is the *set of output values*, S is the *set of partial sequential states*, $\delta : Q \times (X \cup \{\emptyset\}) \rightarrow S$ is the *transition function*, where $Q = \{(s, e) \,|s \in S, 0 \leq e \leq \tau (s)\}$ is the *set* of *total states*, e is the *time elapsed* since the last transition, $\emptyset$ is the *null input value*, $\lambda : S \rightarrow Y$ is the *output function*, $\tau : S \rightarrow \mathbb{R}^{+}_{0,\infty}$ is the *time advance function*. If no event arrives at the system, it will remain in partial sequential state s for time $\tau (s)$. When $e = \tau (s)$, the system produces an output $\lambda (s)$, then it changes to state $(\delta(s, e, x), e) = \big(\delta(s, \tau (s) , \emptyset), 0\big)$, which is defined as an *internal transition.* If an external event, $x \in X$, arrives when the system is in state (s, e), it will change to state $(\delta(s, \tau (s) , x), 0)$, which is defined as an *external transition.*

For a simulation that starts at t and ends at t', activity is defined as:

- *External activity*, A_{ext}, in the time interval $[t, t']$, is defined as a natural number equal to the sum of *external transitions* $\delta_{ext}(s, x) = (\delta(s, \tau (s) , x), 0)$.
- *Internal activity*, A_{int}, in the time interval $[t, t']$, is defined as a natural number equal to the sum of *internal transitions* $\delta_{int}(s) = \big(\delta(s, \tau (s) , \emptyset), 0\big)$.
- Total activity is equal to: $A = A_{ext} + A_{int}$.

A hierarchical composition of components is defined as: $N =< X,\ Y,\ D,\ \{c\} , \{I_i\} , \{Z_{ij}\} >$, where X is the set of input values, Y is the set of output values, D is the set of references to lower level components, I_i is the set of influencees of the lower level components $i \in D$, and Z_{ij} is the i to j translation function inside the hierarchical composition.

As previously defined in [MTV$^+$10], the activity of a composite model depends on the activity of its components. The activity of a hierarchical composition N is the total activity of its components $\{c\}$: $A_N = \Sigma_{i \in D} A_i$

2.1.2 Definition of ACA

Our goal is to evaluate candidate components and compositions in a candidate composition N, at trial $r \in \mathbb{N}$, with $r \leq R$, where R is the total number of trials. To achieve this goal, a *score* is defined as a measure of performance of the top level hierarchical composition B_N, where N is the top level composition. The objective of ACA is to find compositions that achieve a high level of performance.

In one trial r, the *credit* $\varsigma_{i,r}$ of a *candidate component* $i \in D$ is defined as the product of its activity $A_{i,r}$ and of the score of the top level hierarchical composition $B_{N,r}$:

$$\varsigma_{i,r} = A_{i,r}.B_{N,r}$$

The *accumulated credit (*or *achievement)* $\overline{\varsigma_{i,r}}$ *of a component* $i \in D$ over a number of R trials, is defined as:

$$\overline{\varsigma_{i,R}} = \Sigma_{r=1}^{R} \varsigma_{i,r}$$

At trial R the *accumulated credit of a component* $\overline{\varsigma_{i,R}}$ is used to drive the selection of the components *or* compositions.

2.2 Simulation Algorithm

Algorithm 1 summarizes the main function runGeneralTrial() of the program. This loop consists in a composition-simulation-evaluation algorithm. At each iteration, various models are successively instantiated and tested:

- Line 1: The trial loop, with nbTrial the number of trials,
- Line 2: If nbTrial is smaller than startOfBias, the model is "pruned" (instantiated) randomly, *i.e.*, all components are chosen randomly. If nbTrial is greater than startOfBias, the newModel is pruned based on previous credit results of components through the pruneWBestEvaluatedCreditComponents(maxComposition,searchLevel) function, described in Algorithm 2.
- Line 3: The newModel is run,
- Line 4: Credit-based evaluations are saved to repository.
- Line 5: The loop ends when the maximum score (known in advance) is found.

Algorithm 1 Main Activity-based Composition-simulation-evaluation Algorithm.

```
runGeneralTrial():
1. for(nbTrial = 0; nbTrial < maxNbTrial; nbTrial++)
2.   newModel=nbTrial > startOfBias ? pruneWBestEvaluatedCreditCompo-
       nents(maxComposition,level): pruneRandomlyModel()
3.   run(newModel)
4.   creditRepository.add(newModel)
5.   If (terminate(bestScore)) Then
6.       break
```

Algorithm 2 summarizes the pruneWBestEvaluatedCreditComponents(maxComposition,level) function:

- Line 1: For levels lower than the top level, the credit-based probability relation, probaRn, is built as:

$$\text{probaRn} = \{(p(i, R), M) \,|\, i \in D,\, p(i, R) = \frac{\overline{\varsigma_{i,R}}}{\Sigma_{i=1}^{n}\overline{\varsigma_{i,R}}},\ M \subseteq D\}$$

where n is the number of lower level components.

- Line 2: The bestComponents are selected from the repository (more details in Algorithm 3),
- Line 3: bestModel is pruned using bestComponents.

Algorithm 2 pruneWBestEvaluatedCreditComponents(maxComposition,level) Algorithm.

```
pruneWBestEvaluatedCreditComponents(maxComposition,level):
1. Relation probaRn=repHierarchy.buildCreditProbaFunctionFrom(level)
2. bestComponents
      =getBestComponentsFromRep(repHierarchy,maxComposition,probaRn)
3. bestModel = pruneWith(bestComponents)
4. return bestModel
```

Algorithm 3 summarizes the getBestComponentsFromRep(repHierarchy, maxComposition, probaRn) function:

- Line 1: The number of best components nbOfBestComponents is randomly drawn according to maxComposition, which is equal to the maximum number of components at this level.
- Line 2: The corresponding bestComponents (whose number is equal to nbOfBestComponents) are obtained.
- Line 3-4: Remaining components are randomly selected (randomComponents).
- Line 4: The selectedComponents consist of both bestComponents and randomComponents. As discussed in sections 4 and 3.1, this allows to do not be trapped in local optima.

Algorithm 3 getBestComponentsFromRep(repHierarchy, maxComposition, probaRn) Algorithm.

```
getBestComponentsFromRep(repHierarchy,maxComposition,probaRn):
1. int nbOfBestComponents=rand.nextInt(maxComposition)
2. bestComponents=getBestModels(probaRn,nbOfBestComponents)
3. If(nbOfBestComponents < maxComposition) Then
     randomComponents
        =getBestModels(probaRn,maxComposition,nbOfBestComponents)
4. selectedComponents=bestComponents + randomComponents
5. return selectedComponents
```

3 Simulation Results

An archetype model (hockey team) is presented here. Efficency of ACA is shown in section 3.2 .

3.1 Model and Simulation

3.1.1 Description of the Hockey Team Model

Simulation results concern a generic example implementation: The hockey team. A picture of the whole composition is provided in Figure 1.

This model has the following characteristics:

- Only two kinds of position are considered: Defense and attack,
- Only four players are selected at a time, two in defense and two in attack,
- *Abilities* can be only *good* or *bad*. Therefore, there are $2^3 = 8$ possible players at each position (see Figure 2 for player compositions),
- Defenders consist of *abilities* to: pass to defense, pass to forward, and stop. There are $8^2 = 64$ possible compositions in defense.
- Attackers consist of *abilities* to: (good or bad) to: pass to forward, get puck, shoot. There are $8^2 = 64$ possible compositions in attack.
- Number of possible team selections: $64^2 = 4096$.

Figure 1 shows the coupling of the experimental frame to the model. During a trial, the experimental frame sends one attack (puck external event) to each defender. When a defender receives the puck he must stop it, pass it to the other defender who must forward it to one attacker. The latter must pass it to the other attacker who must shoot, in oder to score a goal. Any break in this chain results in no score. To ensure a score, the abilities at every point of the sequence are required to be “good”. Therefore, since there are two such attacks, the maximum score is two.

Finding the best hockey team (i.e., the global optimum) consists in finding the best players (the local optima). The class of problems corresponding to this hockey model is equivalent to the ones resolved by the *greedy algorithm* [CLRS09], which achieves/assumes local optimal choices to find a global optimum. Also, as described in both sections 4 and 2.2, for ACA, in order to do not be trapped in local optima, some components are selected randomly.

3.1.2 Parallel Pseudorandom Simulation

A simulation is determined by: a random number seed, a hierarchical level, and a bias. The random number seed is used to randomly generate a team, using Algorithm 1. There are four hierarchical levels: Level 0 is the top team level, level 1 is the lines level, level 2 is the players level, and finally, level 3 is the abilities level (atomic level). Refering to Algorithm 1, first the bias is set to infinity and the simulation is run (without any connection to the activity credit assignment), until the maximum score is reached. The resulting number of trials is called *nbEvalWithoutHistory*. This variable represents the number of trials required for that seed at that level to find purely randomly the team that can score two goals. After, for each bias from 0 to *nbEvalWithoutHistory*, the simulation is run using the activity credit assignment function and the number of trials required at that level (to find the team that can score two goals), is obtained and called *nbEvalWithHistory*. For each bias value, the speed is defined as: $speedUp = \frac{nbEvalWithoutHistory}{nbEvalWithHistory}$. Note that $speedUp \geqslant 1$ for any algorithm that is not worse than random search. Also $speedUp < 1$ is possible.

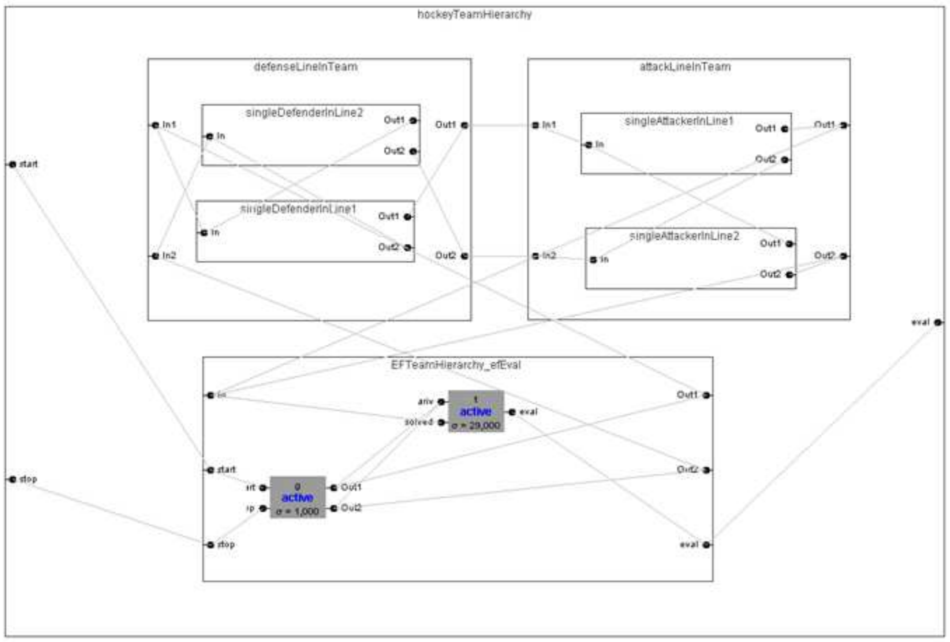

Figure 1: Hockey Team Components. On the bottom is the experimental frame. On the top are the coupled models: attack and defense lines.

As described in Algorithm 1, at each level, different from level 0, atomic components and components of compositions are evaluated. Above the value *startOfBias*, the algorithm takes advantage of the credits of components and compositions. At level 3, there are 12 possible abilities (atomic models). A threshold is randomly selected between 0 and 12, such that below and equal to the threshold, components c are generated using a credit-based probability discrete density function: $p(i, R) = \frac{\overline{\varsigma_{i,R}}}{\Sigma_{i=1}^{n}\overline{\varsigma_{i,R}}}$. Above this threshold, components are randomly generated using a uniform distribution function for each choice. This is the same approach as the one used to obtain the *nbEvalWithoutHistory*. For level 2, there are 4 possible players (compositions). For level 1, there are 2 possible lines (compositions). For levels 2 and 1, the same mechanism is used except than the threshold choice is based on the number of players and lines respectively. Level 0 is different because there is only one composition.

For each bias, 30 replications (random seeds) have been run in parallel on a Symmetric Multiprocessing (SMP) with 8 quadcore-processors (32 cores). Each level runs roughly 40 biases for 30 replications each. It takes approximatively one hour to run one level. Approximately, it would take a day, on a sequential machine, to do the same thing.

3.2 Efficiency of Activity-based Hierarchical Composition Algorithm

The behavior of speed up is a function of bias shown in Figure 3 for a *repository-based random strategy* at level 1 (line level). This strategy is an intermediate one between a purely random strategy and ACA. This strategy intends to evaluate the impact of the repository use on ACA gains. Here, at each bias, the algorithm scans the repository and builds randomly new compositions reusing components from the repository. As for algorithms in Section 2.2, some components are randomly chosen to avoid being trapped in local optima. Above bias=100 and until bias=400, some speed up seems to arise. At level 1, a maximum of two best components can be selected. During the start up phase (before the bias), it seems that there is a high probability for best lines to have been selected randomly and stored in the repository. Then, the repository-based random strategy gets higher probability to select the good lines in a restricted candidate set (the repository). Above bias=400, the average speed up oscillates around 1.0. At bias=0, a very large confidence interval occurs (with a lower bound negative). This can be explained because at this bias, the minimum speed up is only 0.29, which is three times slower than no speed up.

The behavior of speed up is a function of bias shown in Figure 4 for atomic level 3. Note that the curve shows consistency. The average speed up is increasing until 100 and then decreasing. The curve converges to one, without going below. Even the lower bound of the confidence interval never goes below one (showing that bias algorithm is never worse than only randomly selecting components without any use of repository).

Table 1 summarizes the data from the bias perspective for the repository-based random strategy. The table shows that minimum and maximum speed ups are close to the average speed up, which is equal (or very close to one). Also, the bias for largest minimum is very large. It can be concluded that there is no impact of the bias on speed up.

Table 2 summarizes the data from the bias perspective for ACA. For level 3, the table shows the bias (300) for

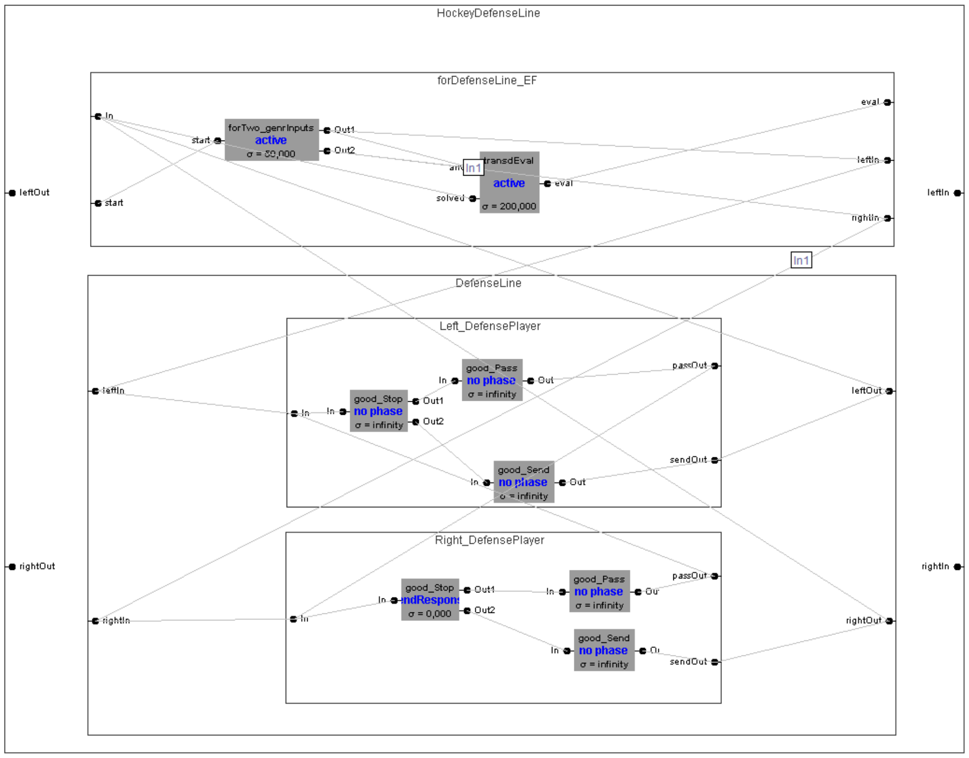

Figure 2: Defense player components.

Level	*Largest minimum*	*Bias for largest minimum*	*Average at that bias*	*Maximum at that bias*
3 - atomic abilities	0.84	2 700	1.04	1.27
2 - players	1.18	2 900	1.2	1.23
1 - lines	1.17	2 900	1.2	1.23

Table 1: Repository-based random strategy: Largest Minimum and Bias where the largest minimum occurs.

which the largest value of minimum speed up is obtained (2.6), the average at that bias (6.5), and the maximum at that bias (11). This can be interpreted as providing guidance for the use of a riskadverse activity credit assignment approach, where a bias should be selected, which guarantees a minimum speed up without sacrifycing possibility for significantly larger speed up. On the other hand the risky approach would be to use a smaller bias with a lower minimum and higher maximum. Levels 1 and 2 show the same kind of behavior. The riskadverse bias choice decreases to a value of 100 where a minimum of 1.76, average of 7 and maximum of 24 occur. At level 2, except for largest minimum, which is slightly lower than for level 1 and level 3, average and maximum values are better. Therefore, we can consider that it should be more valuable to use ACA at this level, for bias=100.

Table 3 shows the size of each components and the number of components for each level. Increasing in levels, the number of components increases. This explains the ACA results at level 2. At this level the size of each component is not too small or too large.

4 Comparison with other Credit Assignment Methods

The *credit assignment problem* [Min61] consists in assigning partial credit to sub-decisions leading to a complete task. This allows investigating new rule paths even if the first steps do not provide immediate reward (*e.g.*, at chess, you can choose to loose a piece to win the game...)

A first solution to credit assignment problem consists in the *bucket brigade algorithm* [Hol92]. In the bucket brigade algorithm, the basic entities are *classifier systems.* A *Learning Classifier system* interacts with the environment through an I/O interface using condition-action rules. The rule-base consists of a population of many condition-action rules (*classifiers.*) The rule conditions and actions are strings of characters from the *ternary alphabet:* $\{0, 1, \#\}$, with $\#$ being as "don't care" when appearing in the condition part. According to the reward of an action (changing the state of the environment) a rules reinforcement is performed. Rules are generated by a *genetic algorithm.* In the bucket brigade algorithm, "the highest bidding classifiers may place their message on the message list of the next cycle, but they have to pay with their bid which is distributed among the classifiers active during the last time step which set up the triggering

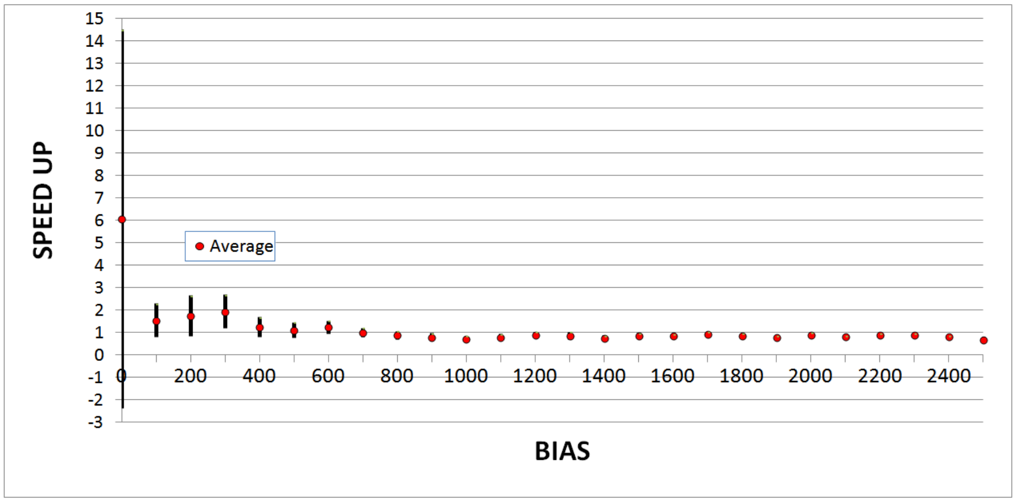

Figure 3: Repository-based random strategy: Average speed up and confidence interval results for each bias (accross 30 replications) – at level 1.

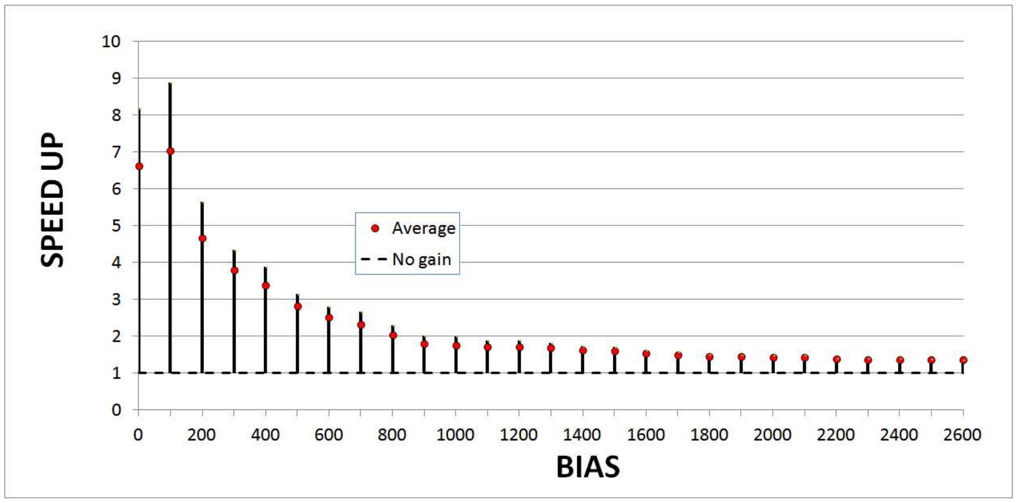

Figure 4: ACA: Average, maximum, and minimum speed up results for each bias (accross 30 replications) – at atomic level 3.

conditions. (...) The central idea is that classifiers which are not active when the environment gives payoff but which had an important role for setting the stage for directly rewarded classifiers can earn credit by participating in 'bucket brigade chains'. " [Sch89]. The first classifiers of a classifier chain gets a partial reward even if the action performed on the environment does not provide the maximum reward expected. Another solution to credit assignment problem consists in the *profit-sharing plan* [Gre88]. Bucket brigade algorithm focuses on incremental schemes. Profit-sharing plan focuses on reward schemes waiting for external rewards. In this approach, problem solving is divided into episodes delimited by the receipt of external reward. A rule is active during an episode if it wins a bidding competition. Whereas bucket brigade is better adapted to rules firing in parallel, profit-sharing is more adapted for single active chains [GBV].

We describe now the main differences/similarities between both activity-based and genetic algorithms(GA)-based credit assignments, starting from usual GA vocabulary:

- *Rules*: GA produces rules, *i.e.*, pairs of *(conditions,actions)* of a classifier system, expressed in bits (or #), GA selects the best actions to make new rules. In contrast, ACA applies to any DEVS hierarchical components and compositions, where the inputs of the component/composition correspond to the conditions from the environment, and outputs of the component/composition correspond to actions.

- *Selection*:

  - GA, at every generation: 1. Rank individuals according to their fitness value, and 2. Keep a percentage of best individuals. On the other hand, ACA: 1. Ranks components and compositions according to their performances in one or more viroment, 2. Maintains a model-base of components and compositions according to their ranking.

Level	*Largest minimum*	*Bias for largest minimum*	*Average at that bias*	*Maximum at that bias*
3 - atomic abilities	2.6	300	6.5	11
2 - players	1.76	100	7	24
1 - lines	1.98	300	4.2	6.5

Table 2: ACA strategy: Largest Minimum and Bias where the largest minimum occurs.

Level	*Number of components*	*Size of each component*
3 - atomic abilities	0-12	2
2 - players	0-4	8
1 - lines	0-2	64

Table 3: Number of components and size of each component, for each level.

– GA, *e.g.* in bucket brigade, select a *rule* i to fire using a bid-based probability distribution:

$$p(i) = \frac{bid_i}{\Sigma_{i=1}^{n} bid_i}$$

where n is the number of rules. On the other hand, ACA select components and compositions using a credit-based probability distribution:

$$p(i, R) = \frac{\overline{\varsigma_{i,R}}}{\Sigma_{i=1}^{n} \overline{\varsigma_{i,R}}}$$

where n is the number of lower level components.

- *Avoiding traps in local optima*: GA use mutation, *i.e.*, a bit can be randomly changed in a rule, likewise ACA, in the model-base, uses a combination of highly ranked or randomly selected components/compositions.
- *Combination of sub-solutions*: GA use crossover genetic operator to combine parts of two parent chromosomes to make a new child chromosome. ACA, using hierarchical composition and at any level, combines (according to the above policies) components from the next lower level. Similarly to crossover both highly ranked and randomly selected lower level components are used.

5 Conclusion and Future Work

Using a hockey team archetype, we illustrate a proof of concept in this paper that ACA:

- applies to any DEVS hierarchical components and compositions within any DEVS Experimental Frame,
- evaluates components/compositions at each level of the hierarchy,
- converges on good components/compositions faster than purely random search,
- is built-in to the DEVS simulators with little overhead and a minable to high performance parallel, and
- automatically synthesises systems from a model-base thus enabling reusability of highly rated components in compositions.

Furthermore, as described in sub-section 4, *bucket brigade* algorithm is better adapted to rules firing in parallel, *profit-sharing* is more adapted for single active chains. ACA can be applied to models embedding both sequential and parallel chains of components.

In further research, the ACA needs now to be applied to more challenging (realistic/stochastic) model composition problems where the best performance is not known a-priori. Finally, more theoretical work can be done to justify the fact that there is an underlying credit ranking that any simulation using ACA would converge to. However, the results provided here still indicate that significant speed ups were achieved for ACA at each level of the hierarchy (e.g., between 2.6 and 85 for bias=0, at atomic level) and executed within feasible time frame on relatively unexpensive multi-processor.

Acknowledgements

This research is supported by the CNRS.

References

[BPZ00] H. Praehofer B. P. Zeigler, T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, 2000.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition edition, July 2009.

[GBV] A. H. Gilbert, Frances Bell, and Christine L. Valenzuela. Adaptive learning of process control and profit optimisation using a classifier system. *Evolutionary Computation.*

[Gre88] John J. Grefenstette. Credit assignment in rule discovery systems based on genetic algorithms. *Mach. Learn.*, 3:225–245, October 1988.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[KE03] George J. Klir and Doug Elias. *Architecture of Systems Problem Solving.* Springer, 2003.

[Min61] Marvin Minsky. Steps toward artificial intelligence. In *Computers and Thought*, pages 406–450. McGraw-Hill, 1961.

[MTV+10] Alexandre Muzy, Luc Touraille, Hans Vangheluwe, Olivier Michel, Mamadou Kaba Traoré, and David R. C. Hill. Activity regions for the specication of discrete event systems. In *Spring Simulation Multi-Conference Symposium On Theory of Modeling and Simulation (DEVS)*, pages 176–182, 2010.

[MZ08] A. Muzy and B. P. Zeigler. Introduction to the activity tracking paradigm in Component-Based simulation. *The Open Cybernetics and Systemics Journal*, 2:48–56, 2008.

[SB98] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning).* MIT Press, 1998.

[Sch89] J. Schmidhuber. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412, 1989.

Author Biographies

ALEXANDRE MUZY is research fellow CNRS at the Università di Corsica Pasquale Paoli. He received his Ph. D. from the same university, in 2004. He is currently Associate Editor of the SIMULATION Journal (Transactions of The Society for Modeling and Simulation International) and member of many international program committees. His main research domain concerns the use of activity for modeling and simulating complex systems.

BERNARD P. ZEIGLER is RTSync Chief Scientist, Emeritus Professor of Electrical and Computer Engineering at the University of Arizona (UA) and Research Professor in the C4I Center at George Mason University, is internationally known for his seminal contributions in modeling and simulation theory. He has published several books. He is well known for the Discrete Event System Specification (DEVS) formalism that he invented in 1976 and was named Fellow of the IEEE and SCS.