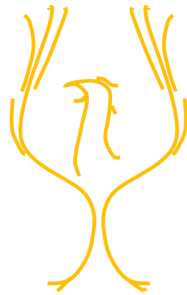
Future and Emerging Technologies
FET-Open



Phoenix

665347

Definition of Agent's Instinct and Related Algorithms

Deliverable D5.1

Contents

1 Introduction **4**
1.1 Phoenix . . . 4
1.2 About this Deliverable . . . 5
1.2.1 Scope . . . 5
1.2.2 Objectives . . . 5
1.3 Deliverable Structure . . . 6

2 Instinct Definition **7**
2.1 Problem Statement . . . 7
2.2 Biological Instincts . . . 8
2.3 Instinct Representation . . . 9
2.3.1 Behaviour Trees . . . 9

3 Instinct Evolution **12**
3.1 Evolutionary Algorithms . . . 12
3.2 Grammatical Evolution . . . 13
3.3 Instinct Evolution Scheme . . . 15
3.3.1 Agent Abstraction . . . 16
3.3.2 Action Generation . . . 17
3.3.3 Actions Pool . . . 17
3.3.4 Nodes Pool . . . 17
3.3.5 Conditions Pool . . . 17
3.3.6 Evolution of Instincts . . . 18
3.4 Case Study: Compression . . . 18

4 Conclusion **23**

5 Future Outlook **23**

This project is funded
by the European Union

1 Introduction

In this section we start with a short description of the Phoenix project, with an emphasis on the concepts relative to this deliverable. Secondly, we outline the deliverable scope then shed the light on its objectives. Finally, we lay down the deliverable overall structure. This deliverable summarises the activities related to the definition of agent's instinct within the Phoenix project and related algorithms. Firstly, a short overview of the Phoenix project is presented, highlighting the concepts relevant to this deliverable. This is followed by definition of the scope of the deliverable and its objectives. Finally, the overall structure of the document is outlined.

1.1 Phoenix

In Phoenix project the process starts with user's question about an unknown environment. The answer to this question is then obtained via two main loops as shown in Figure 1.

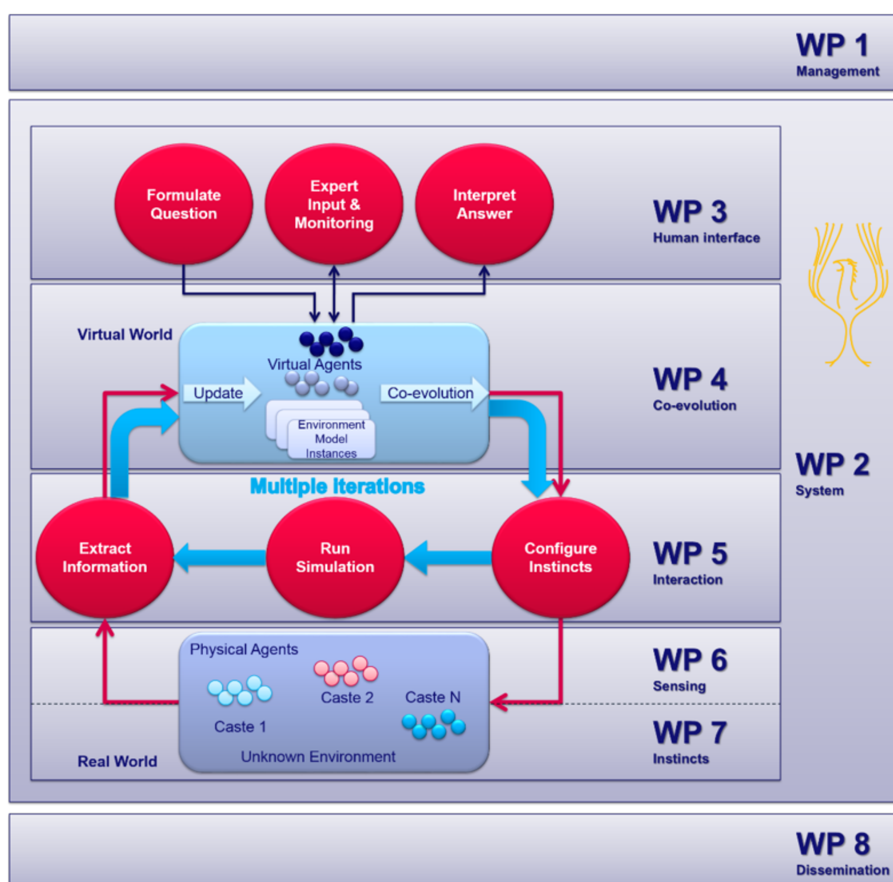


Figure 1 – Phoenix Scheme

In the first *outer* loop (red cycle), extremely small autonomous physical agents explore the unknown environment. The first iteration of the loop is initiated based on a *best guess* model for the environment and the agent configuration derived from — possibly uncertain — knowledge elicited from experts that is stored in a database.

Based on the information provided by the agents recovered from the environment, a virtual model of the environment is constructed, and further refined in the *inner* loop (blue cycle). During this nested loop, the virtual model (and possible multiple instances of it) is repeatedly explored by virtual representations of the physical agents using an evolutionary algorithm (EA), co-evolving both the virtual model and the virtual agents, helping to refine the next incarnation of the actual physical agents.

These reincarnated agents will again explore the unknown environment to gather more data. In this context, axiomatically, the adaptive behaviour of the agents is imperative. Moreover, this behaviour has to facilitate an adequate capture of different physical properties of the environment under investigation, in order to allow its reconstruction while optimising the agents' available resources to achieve that. Furthermore, this behaviour needs to function in a real-time system, thus it should guarantee a response within specified time constraints. In addition, it has to be implemented with limited hardware. Due to these challenges regarding the agent behaviour, we introduced the concept of *instinct*.

In Phoenix we target to mimic the biological instinctive behaviour, where we *evolve* agents reactive behaviour to different stimulations from its sensors via evolutionary algorithms (EAs) and embedding such evolved instincts on agents' hardware.

In that regard, the objective of the co-evolution module is to insure that the evolved behaviour optimises the agents' resources, while capturing needed environment properties that would empower the system to answer the user's inquiry on the environment under investigation. But, *why do we need to mimic instincts from biological systems*? *how to define and represent an instinct within Phoenix scheme*? and *how to evolve it within the EA framework*? In this deliverable, we will address these questions.

1.2 About this Deliverable

In this subsection, the scope of the deliverable followed by the main objectives is presented.

1.2.1 Scope

This deliverable attempts to abstractly fill the design space between the co-evolution module, defined in WP4, and the hardware implementation laid down in WP6 and 7. In other words, the scope of this deliverable is the interlayer between the EA framework and the hardware design and functionality.

1.2.2 Objectives

The deliverable is interconnected with Task 5.1 ("Agent's instinct"). The main objectives are:

- Definition of agent's instincts
- Definition of related algorithms

1.3 Deliverable Structure

This deliverable is structured as follows: In the next section, the problem statement is first defined, then the concept of instinct is presented. This is followed by an illustration of the instinct representation scheme. In section, the main architecture of the co-evolution state-of-the-art framework, dedicatedly designed for Phoenix, is introduced. In Section 3 the evolution process of instincts is presented, highlighting the use of Grammatical evolution. Finally, Sections 4 and 5 outline, respectively, the conclusion and the future outlook of the deliverable.

2 Instinct Definition

In order to define instincts in the context of Phoenix, this section starts with setting the fundamentals of Phoenix problem regarding the agent's behaviour and its learning process via presenting different design objectives. This is followed by an overview on biological instincts. Finally, the instinct representation scheme is illustrated.

2.1 Problem Statement

One of the key aspects in Phoenix problem statement regarding agent's behaviour is the fact that the mathematical representation, associating different low level hardware design variables with high level defined objectives, is either not possible due to the huge design distance between such lower level variables and higher level objectives, or limited since it may exists only for some sub-objective(s) and independent from the other available sub-modules in the agent's hardware architecture. This leads to the first design objective:

Objective 1 *The learning process of the behaviour must rely primarily on the behaviour performance metrics, i.e. its fitness, and on the assumption that only limited, if any, access to the mathematical formulation relating available tunable variables with the defined objectives is available.*

Moreover, as stated in the introduction and from Phoenix scheme in Figure 1, EA is used in co-evolving both the virtual environment model and the virtual agents, to help refine the next incarnation of the actual physical agents. These reincarnated agents will again explore the unknown environment to gather more data and eventually answer the user's question. In that regards, deliverable 4.1 [1] introduced an EA framework to be used for wide range of optimisation problems on both agent and environment parameters. Consequently, the learning process of the agent's behaviour must not be separated from this co-evolution module. Therefore:

Objective 2 *The learning process of the agent's behaviour must be integrable with the evolutionary algorithm framework set in* $WP4$.

Furthermore, initial investigations that focused on the use of game theory to perform optimisation on different agents' parameters concluded that game-theory-based optimisation algorithms are only viable for few of the agent's parameters such as transmission power, due to the lack of corresponding agent interaction models for the other parameters. In addition, due to the profound limitations in the agent's resources, it was inferred that on-line game theoretic approaches can not be used, consequently:

Objective 3 *The learning process of the agent's behaviour must be conducted off-line, i.e. off-line learning for an on-line behaviour.*

Additionally, it is expected that the behaviour will cover different levels of abstraction, from low level behaviour, as on the agent's sensor level, to more abstract schemes such as swarm intelligence like collective task allocation. Moreover, this behaviour should facilitate the separation, re-combination and addition of different agent sub-modules, and adapt accordingly, i.e. should be modular, thus:

Objective 4 *The learned behaviour must facilitate hierarchical expansion and modularity.*

Empirically, it is assumed that it is infeasible to know a priori which agent will be where in the environment and when, at least with reliable probabilistic certainty. Therefore, the learned behaviour shall be used with all agents and must function effectively everywhere in the environment.

Objective 5 *The evolved behaviour must be universal, i.e. fit to be used by any agent at any point on the spatial and temporal dimensions.*

Moreover, this evolved behaviour will be implemented in hardware, consequently, it must be integrable with it. Hence, even if its actions optimise the use of the available hardware to achieve user's defined objectives, the behaviour's control flow itself triggering these actions must also be efficient and should minimise the use of hardware resources:

Objective 6 *The learning process of the agent's behaviour must attempt to find the control flow that achieves its functionality with least possible consumption of hardware resources.*

The rest of the deliverable is dedicated to present a scheme and its building concepts that fulfil these objectives. Next, biological instincts are introduced and the benefits from mimicking them in Phoenix is presented.

2.2 Biological Instincts

In biological systems, any behaviour is considered instinctive if it is performed without being based upon prior self experience but rather on hereditary bases. This means that the learning process is done on the genetic level, i.e. the learning experience is accumulated genetically. Such learned behaviours found in nature may include a simple, low level and fast or a complex and multi-system reaction(s) to a given stimulus.

This instinctive behaviour is of exceptional interest to Phoenix, as mimicking it can play an important role in solving the problem of behaviour learning. On one hand, it is genetic based, thus, learning is conducted on the genetic level, which gives the power to achieve solving a problem without — or with very limited — understanding to its mathematical formulation and only based on its performance on that environment (Objective 1). Moreover, having a learning model that is using genetic bases makes the integration process with the EA module easier, which puts us one step closer to achieving Objective 2. Furthermore, in many species this behaviour is fast, reflex-like, and doesn't require extensive computation to be conducted, which is particularly suited for Phoenix due to the agent's resource limitations (Objective 6). For all these reasons and more, biological instincts are adopted as our design methodology, and the presented design in this deliverable is an attempt to mimic it.

For the rest of the deliverable, it will be referred to the agent's behaviour by *instinct*. This is valid for both virtual and real agents. One of the first challenges in adopting the

instinct concept is how to represent this instinct in a way that fits Phoenix's problem statement and achieve its defined objectives. In the next section, a representation method called *Behaviour Trees* (BTs) is presented.

2.3 Instinct Representation

This subsection is dedicated to the representation scheme of instincts. In this regard, it is proposed to use *Behaviour Trees* (BTs) [2] [3]. A good starting point to emphasise BTs' strength would be to highlight the problems with the other commonly used schemes. A widely used representation scheme is for example based on *Finite State Machines* (FSM).

Firstly, with FSM it is hard to achieve both hierarchical expansion or modularity because the separation or re-combination of a state requires a total re-design of the whole FSM, thus contradicting with Objective 4. Furthermore, they are not flexible to user defined actions and conditions as it is a complex process to abstract these attributes from the FSM structure.

BTs share many of the *good* attributes with FSM, but it lacks a lot of its problems. Next, an overview on BTs is presented.

2.3.1 Behaviour Trees

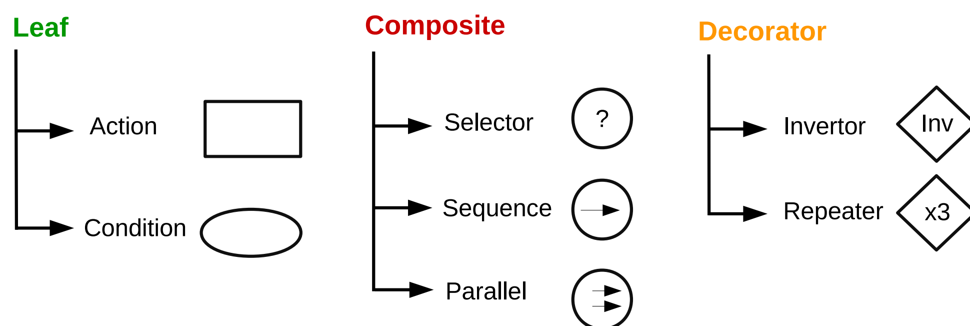


Figure 2 – Behaviour Tree Nodes: Examples

As an alternative to FSM, BTs were introduced in the gaming industry mainly to offer a more modular, flexible and readable representation scheme. And because of these reasons, many famous games now use it for developing their AI. In addition, they have become widely popular in robotics and unmanned air vehicles.

Formally, a BT is depth-first acyclic directed graph. Any node in this tree is either a parent, i.e. connected to lower level nodes dubbed as a *child* nodes, or a *leaf* node which has no child nodes. Moreover, each BT has one unique parent node named *root* which can not have a parent and has only a single child.

Each node can have one of three possible states: Success, failure or running. Furthermore, all nodes, except the root node, fall under two main categories: control flow nodes or

execution (leaf) nodes. Control flow nodes are further sub-categorised to: composites nodes and decorative nodes, each include wide range of possibilities such as selector and parallel nodes. On the other hand, the execution (leaf) nodes have only two possibilities: action or condition nodes. Figure 2 highlights the most commonly used nodes in each category. Their functionality can be described as follows:

- **Leaf Nodes**
  - **Action node**: Returns *running* when the given action is still running, *success* when it is done and *failure* otherwise.
  - **Condition node**: Returns *success* when the condition is fulfilled and *failure* otherwise.
- **Composite Nodes**
  - **Selector node**: Sequentially ticks its children nodes, starting from the outmost left node, and returns the state of the first non-failing child, i.e. either *success* or *running*. Otherwise, it returns *failure*.
  - **Sequence node**: Sequentially ticks its children nodes, starting from the outmost left node, and returns the state of the first non-succeeding child, i.e. either *failure* or *running*. Otherwise, it returns *success*.
  - **Parallel node**: Sequentially ticks its children nodes, starting from the outmost left node, if the number of succeeding children exceed a defined node parameter limit S, it returns *success* and if the number of failing children exceed a defined node parameter limit F, it returns *failure*, otherwise it returns *running*.
- **Decorator Nodes**
  - **Inverter node**: Ticks its only child, and returns *success* if the child returns *failure*, and *failure* when the child returns *success*.
  - **Repeater node**: Ticks its only child, and returns the state of its child when it is sent n number of times, where n is an internal variable.

As for the execution process of BTs, it starts from the root *ticking* its only child node, which consequently start ticking its children signalling the permission to start their functional properties, and, consequently, return their respective state to the parent that ticked them. Figure 3 shows a BT example without the root node.

Given this description, it important to highlight why BT is an appropriate scheme for representing instincts in Phoenix. Firstly, adding and removing nodes can be easily done without any changes in either their parent nodes or child nodes. Furthermore, as it is an acyclic graph with a root node, thus, it facilitates hierarchical expansion, which is in compliance with Objective 4. Moreover, The availability of dedicated nodes for actions and conditions gives the user design flexibly. And since it is a tree by structure, its complexity in representation can be a quantitatively measured, which is in agreement with Objective 6. One important aspect is handling parallelism, with the ticking method,

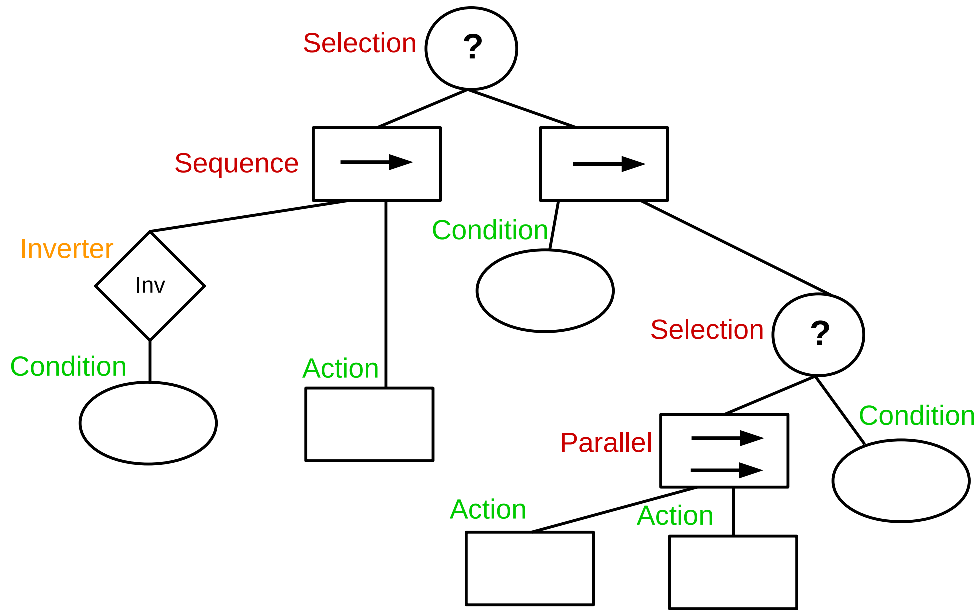


Figure 3 – BT: an example

all deadlock avoidance algorithms are not needed, thus minimising complexity which is reflected positively on hardware resources.

Finally, having a dedicated nodes for actions and conditions gives the user significant flexibility and gives any learning module a wide design space. The challenge now is how to integrate a BT in the learning mechanism of an EA module? This is a critical task that needs to satisfy Objective 1 and 2.

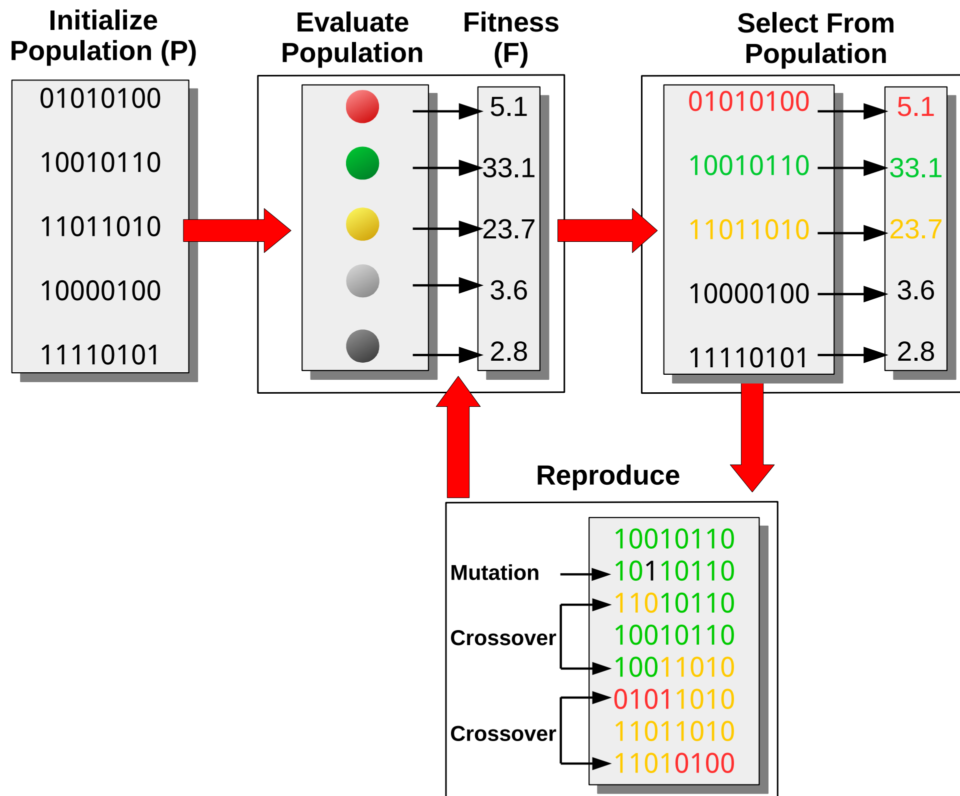


Figure 4 – EA: An example

3 Instinct Evolution

In this section, a description of the evolutionary process of agent's instinct is presented. The next two subsection give an overview on evolutionary algorithms and grammatical evolution, to set the fundamentals needed to explain in details the evolution of instincts scheme.

3.1 Evolutionary Algorithms

Survival of the fittest is the fundamental principle of the Darwinian natural selection process. This principle remains to ensure the survival of living organisms. In fact, the biological evolutionary process has a unique ability to adapt and optimise its parameters without a full understanding of either the environment or the relationships between these parameters.

Evolutionary Algorithms (EAs), which mimic this biological process computationally, are a valuable search and optimisation tool suitable for many real-world problems characterised by complex multidimensional search spaces. As a result, EAs are widely used e.g. in designing and reinventing electronic circuits [4], developing software [5] and even designing antennae for satellites orbiting outer space [6, 7].

To illustrate how EA functions, Figure 4 presents an example. Let's assume it is re-

quired to find the best agent design that is fit for a certain environment. The first step is *initialise population*, where a population of possible designs are — usually randomly — generated in the form of genes. These genes could take wide range of forms, e.g. binary or integers. Then, using a genotype-phenotype map, these numbers are realised as agents (Red, green, yellow, light grey and dark grey agents). Afterwards, each agent is sent to the environment for *evaluation*. In that regard, the user must define a criteria how to judge which one is more fit to the environment, i.e. a *fitness function*. Next, depending on their performance in the evaluation, some of those designs are *selected* to *reproduce*. Many schemes for selection and reproduction exist in literature. Generally, reproduction means that the selected agents will conduct some exchange of their genes, or at least experience some alterations on them, based on probability among other possible criterion. Crossover and mutation are two widely used operations in the reproduction process.

The newly generated population is then mapped again to real agents and evaluated. This loop continues with each generation the solution is getting closer to the optimum, until a certain user-defined criteria is fulfilled, e.g. the exhaustion of pre-defined computation resources. From this example it is clear that genotype-phenotype mapping is a critical process to perform optimisation via EA .

Since it is already established that agent's instincts shall be evolved via EA mimicking biological instinctive behaviour, and since BT is the scheme picked to represent an instinct, a genotype-phenotype map for BT is needed.

3.2 Grammatical Evolution

One of the most common concepts used in Computer Science is the concept of *grammars* [8, 9]. Like in non-programming languages, their main objective is to restrict a certain domain via the definition of possible legal expressions.

In grammar terminology, a non-terminal character is a character that can be replaced with other characters to eventually turn into a terminal one using the production rules, on the other hand, a terminal character is an elementary character that can not be further replaced. A typical context-free Grammar is defined by a 4-tuple [10]:

$$G = \langle N, \Sigma, P, S \rangle \tag{1}$$

Where N is a finite set of *non-terminal characters*, Σ is a finite set of *terminals*, P is a finite set of *productions* in the form, $S \in N$ is the *start symbol* and

$$X \rightarrow \alpha, X \in N \wedge \alpha \in (N \cup \Sigma)^*, \tag{2}$$

where * is a Kleene star.

In EA context, grammar plays an important role in the genotype-phenotype mapping process [11] [12]. Conceptually, the genotype-phenotype mapping is the interpretation of a genotype of an individual in the population to an actual solution. In that regard, if the solution is a structured entity that can be defined with a set of rules, grammar then can

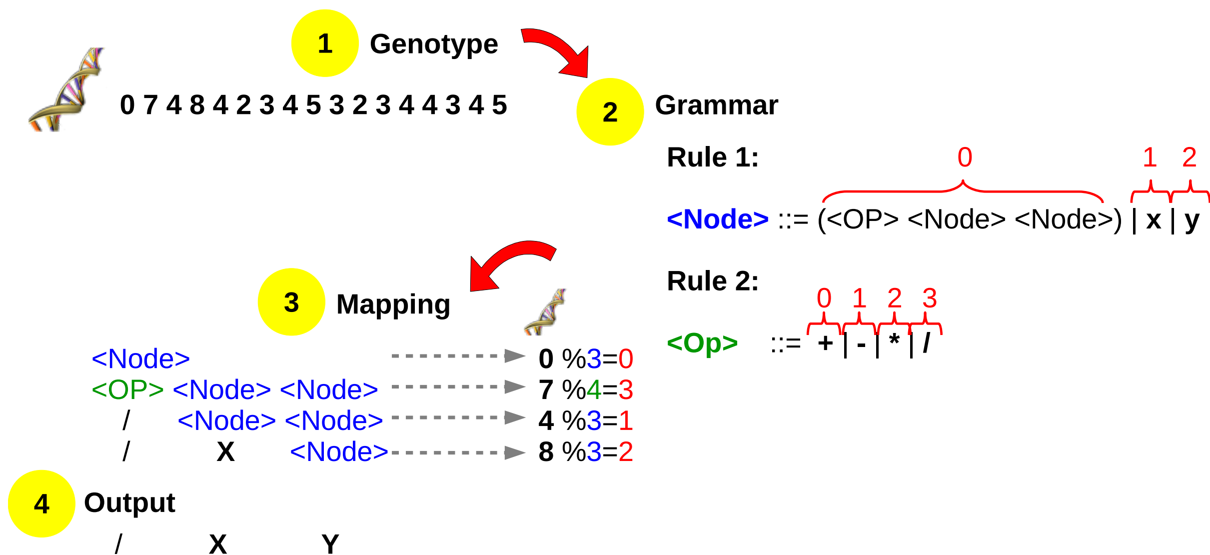


Figure 5 – Grammatical Evolution: Example

be used for mapping and to guarantee that the generated interpretation is syntactically correct.

Figure 5 illustrates an example of using a grammar in genotype-phenotype mapping, the idea is to map the genotype given in step 1 to an output math operation in the prefix format, e.g + y x, from a starting non terminal given character <Node>. To do this mapping, a grammar (rules are written in Backus Naur Form (BNF)) is given with two rules. Rule 1 states that the non-terminal character <Node> can be replaced with one of three possibilities. Similarly, rule 2 states that the non-terminal character <Op> can be replaced with one of 4 possible terminal characters: +, -, *, and /. Now, let us use the grammar to know which operation the genotype will replace <Node> with:

1. The first genotype value is 0, and since the code started a with the non-terminal character <Node>, therefore rule 1 is applied and since rule 1 has 3 possibilities, mod 3 will be used on the genotype value, leading to 0 mod 3 = 0, therefore, <Node> becomes <OP><Node><Node>.

2. The next genotype value is 7, and the first non-terminal character now is <Op>, therefore rule 2 is applied and since rule 2 has 4 possibilities, mod 4 will be used on the genotype value; 7 mod 4 = 3, therefore, <OP><Node><Node> becomes / <Node><Node>.

3. The next genotype value is 4, and the first non-terminal character is <Node> (/ is a terminal character), therefore rule 1 is applied again and mod 3 will be used on the genotype value; 4 mod 3 = 1, therefore, / <Node><Node> becomes / x <Node>.

4. The next genotype value is 8, and the first non-terminal character is <Node> (/ and x are a terminal characters), therefore rule 1 is applied again with mod 3 used on the genotype value; 8 mod 3 = 2, therefore, / x <Node> becomes / **x y**.

This methodology can be used in the genotype-phenotype mapping of BTs, facilitating its evolution. Next, the full instinct evolution scheme is laid out.

3.3 Instinct Evolution Scheme

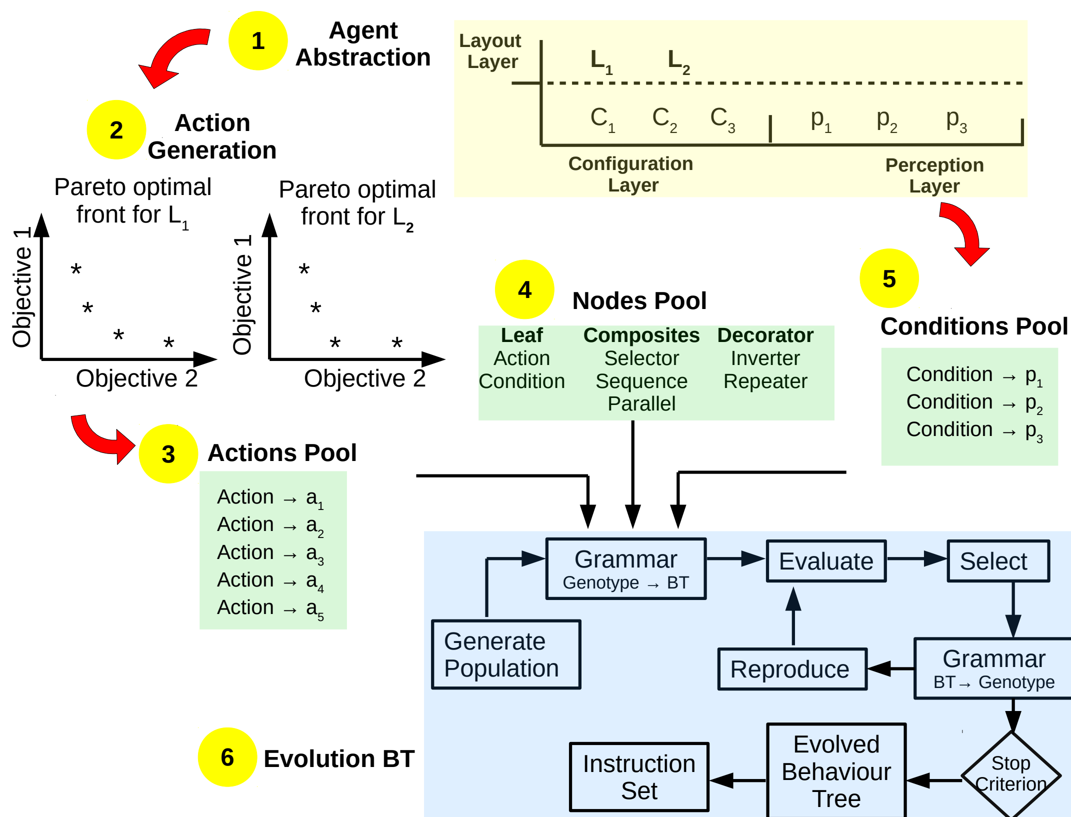


Figure 6 – Instinct evolution scheme

So far the objectives of Phoenix regarding agent's behaviour and its learning process is identified (Section 2.1), the general methodology is chosen: mimicking biological instinctive behaviour (Section 2.2), the representation scheme of an instinct is picked: behaviour trees (section 2.3.1) and its genotype-phenotype mapping method is chosen: grammatical evolution (section 3.2). Now a full description to the instinct evolution scheme is presented. The scheme has six main steps as shown in Figure 6.

To give an overview on the instinct evolution scheme, a bottom-up approach will be used starting from step 6: In step 6, the evolution of the BT using grammatical evolution is conducted, the objective here is to allow the EA process to explore the search space efficiently and generate the optimum behaviour in the form of instruction set. As described earlier, BTs are flexible and facilitate user defined actions and conditions. Additionally, there is a wide range of possible nodes, each facilitate different functionality. Consequently, defining pools of possible *interesting* actions, conditions and node types before the commencement of the evolution process will achieve multiple objectives:

- It will minimize the solution space and, consequently, facilitate faster convergence in producing the optimum behaviour. And although faster convergence is figure of merit in any optimization scheme, it is of an exceptional importance in Phoenix as the simulation process of fluid environment is computationally exhaustive.
- It will facilitate the accumulation of knowledge by simply adding and/or removing actions or conditions.
- It will allow to use the scheme on different levels on the hardware architecture without changing its structure.

As a result, steps 3, 4 and 5 are designed for that purpose. However, in order to define *interesting* actions, an optimisation on the possible tunable variables relative to user defined objective must be conducted first, which is done in step 2. Finally, to pin point all possible tunable variables for each sub-module in the hardware architecture to conduct step 2 and to identify possible perception points in the hardware needed in step 5, an agent abstraction is needed to highlight all these elements, this is executed in step 1. For further analysis to each introduced step, a detailed description is given in the following:

3.3.1 Agent Abstraction

Motivated by the definition of action and condition pools, the agent is abstracted into three main layers:

- Layout layer
- Configuration layer
- Perception layer

These layers are designed in order to be able to explicitly identify all possible configurations and their respective hierarchical position. Identifying the relationship between possible configurations and hierarchy will allow the EA process to control effectively the evolution process with the fitness function, e.g. setting a higher cost in the configuration change that is done on a higher hierarchical level relative to a lower one.

In the layout layer the each configurable module in the hardware architecture identifies all possible sub-modules it can activate. e.g. the compression module might have three possible sub-modules to run: a Zero-order hold (ZOH), a wavelet and a Fast Fourier Transform (FFT). In the layout layer these sub-modules are identified.

One the other hand, the configuration layer contains the set of configurable parameters for each of these sub-modules.

Both the layout layer and the configuration layer are linked to the actions pool. But, in order to identify the points where the conditions can be applied on, a perception layer is used, where all sensing points interacting with the environment are identified, e.g. temperature and pressure sensors.

3.3.2 Action Generation

In this step a multi-objective algorithm is used define the Pareto optimal front of the variables in the configuration layer for each sub-system in the layout layer relative to user defined objectives. It is suggested to use non-dominated sorting genetic algorithm. In this process the user is given a chance to identify the objectives relevant to sub-module under investigation.

The outcome of this process is a set of *interesting* configurations for each element in the layout layer, which will help in the generation of actions pool. It is worth mentioning that setting the maximum genotype integers will lead to setting the max number an action ID can have, which will finally set the max number of actions a pool can take.

3.3.3 Actions Pool

The set of configurations extracted from the Pareto-optimal front constitute a pool of potential actions an agent can conduct, setting up the actions pool. Each action is identified by:

1. A unique ID to the sub-module(s) the action is associated to (layout layer)
2. A configuration ID with a set of the the size of configurable variables with the respective configurations. (configuration layer)

A pre-set number of integers will be allocated to be used in identify the action from the genotype value as explained in details in section 3.3.6, this number will set the maximum possible actions available in the action pool.

3.3.4 Nodes Pool

In this step the pool of all possible control flow nodes of interest are defined. Each type of node has a different complexity in their implementation in the instruction set, and like all other pools, the trade-off between extending the solution space to include more possible nodes versus flexibility by having a wider range of nodes, each adding a new dimension of functionality, must be analysed and controlled in the fitness function.

Furthermore, although BTs come with a huge list of possible control flow nodes and possible extensions, e.g the probability extension where the parent node chooses with a certain probability which child node to tick, it is also possible to define new nodes to fit Phoenix design needs.

3.3.5 Conditions Pool

All entities of the perception layer represent a potential condition point to be monitored for further triggering of an action, and as a result all these points are fed to the condition pool.

Each entity in the condition pool has three basic dimensions:

1. A unique ID to its real perception point, for example an accelerometer can read three axis, therefore, the possible IDs are three.
2. Number of threshold points to be applied on sensor readings, e.g. one threshold means classify sensor readings into two segments.
3. The respective position of the threshold point(s) within the sensor range, i.e. where is the threshold(s) identified in the previous step is placed within the sensor range.

It is important to highlight here Objective 5, consequently, all definition of threshold points and their positions must be conducted in probabilistic manner, i.e multiple runs and from multiple agents. And with each conducted experiment, this probabilistic knowledge is accumulated converging to the best threshold definitions and for all agents as a whole, not for a specific agent in a specific run.

3.3.6 Evolution of Instincts

At this point we have three different pools as follows:

$$A_{pool} = \{a_1 \dots a_i\} \tag{3}$$
$$N_{pool} = \{n_1 \dots n_j\} \tag{4}$$
$$C_{pool} = \{p_1 \dots p_k\} \tag{5}$$

The first step is the generation of a population, then a grammatical evolutionary process is executed with the following BNR rules

$$BT ::= \langle T, T \rangle \tag{6}$$
$$T ::= T, T | N\langle T \rangle | A | C \tag{7}$$
$$N ::= n_1 | \dots | n_{i-1} | n_i \tag{8}$$
$$A ::= a_1 | \dots | a_{j-1} | a_j \tag{9}$$
$$C ::= p_1 | \dots | p_{k-1} | p_k \tag{10}$$

Where BT is the start symbol, T sets a non-terminal character, N sets the grammar rule for choosing a node n_i from the nodes pool, A sets the grammar rule for choosing an action a_i from the actions pool, C sets the grammar rule for choosing a condition p_i from the conditions pool. It is important to highlight that this used grammar can be easily extended.

All evolutionary settings for step 6 can be generated using KIEA framework, which will lead to optimization of these EA parameters, in addition to accumulation of knowledge. Additionally, the cost function can include all needed properties such as instinct complexity.

3.4 Case Study: Compression

In order to project the introduced instinct evolution scheme on a Phoenix related case, and only as a preliminary proof-of-concept, a scenario is introduced Figure 7. In this

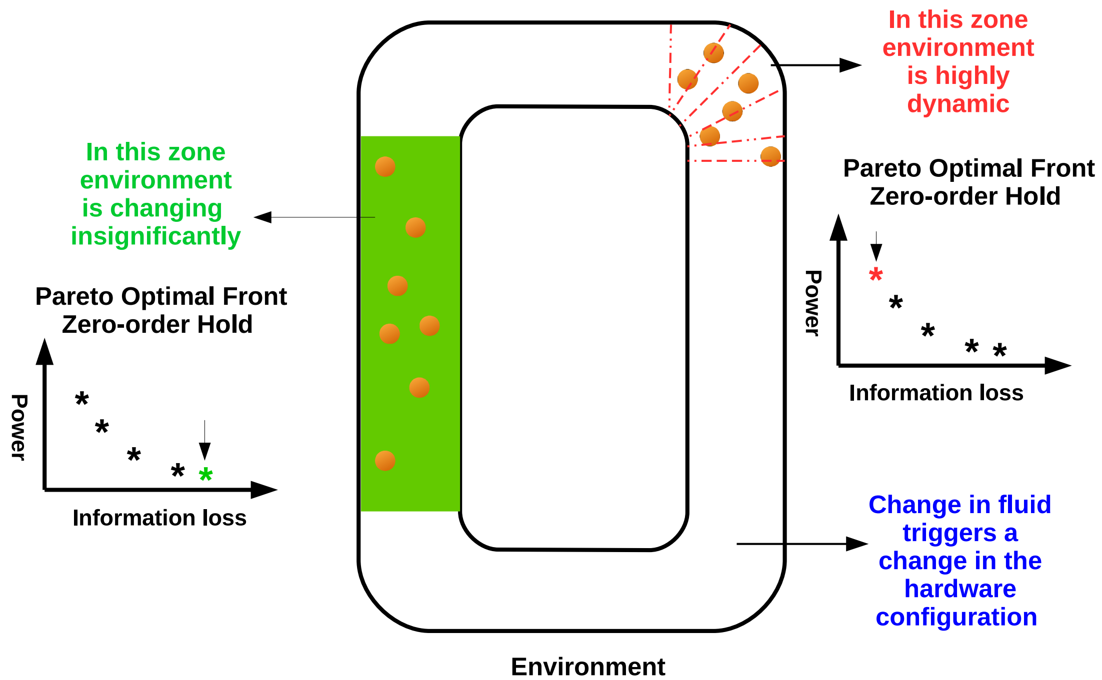


Figure 7 – Environment Scenario

scenario, agents are injecting in a water loop. In this context, compression is critical due to the limited energy available on the agents, this brings emphasis the role optimisation of agents resources through dynamic hardware reconfiguration (online behaviour) [13]. Ideally, the agent should use a compression technique with low power and high information loss in the environment zones where the changes are not significant (green zone) and use another configuration in the (red) zone, where it invests more energy to capture this highly dynamic zone properties.

In this example, the hardware architecture offers two compression techniques: Zero-order hold and wavelet. Furthermore the agents have two sensors: accelerometer and magnetometer each reads the three axis. The objective is develop an instinct off-line using EA that will make the agent capable of capturing the environment properties in different environment zones with the least possible power consumption by reconfiguring its available tunable parameters. Furthermore, agents were extracted after conducting a real experiment, and the sensor readings from this experiment is given.

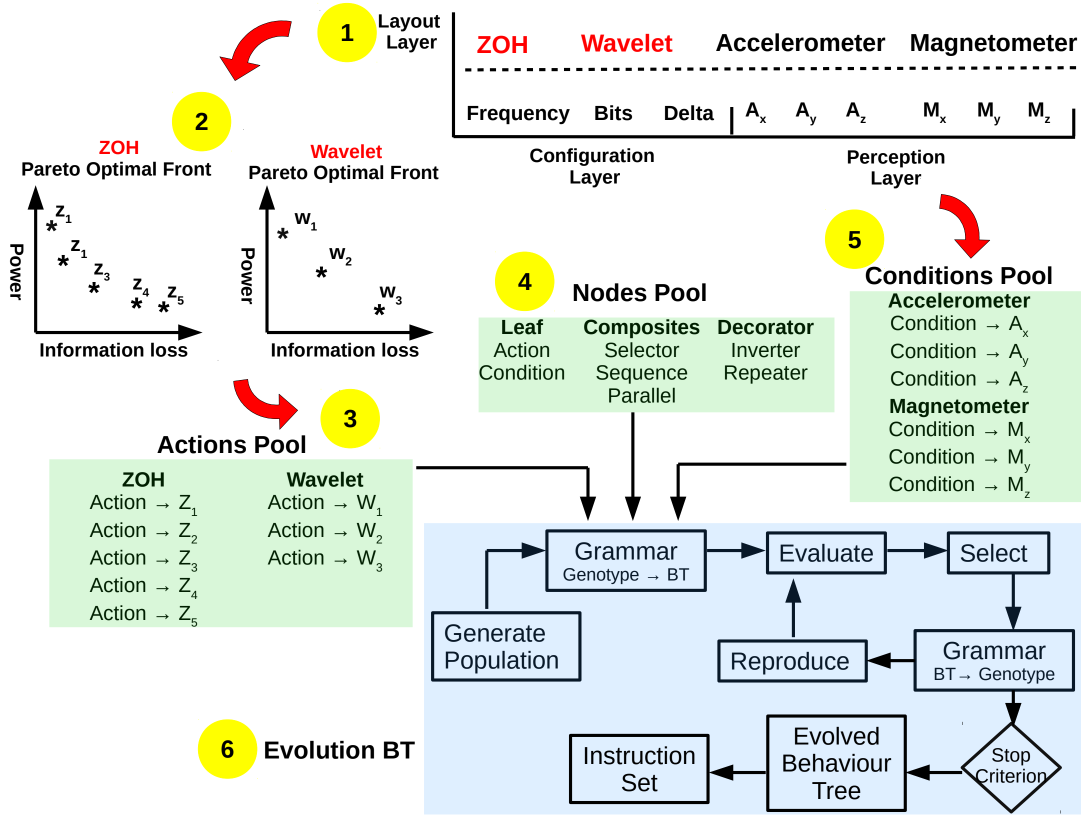


Figure 8 – Case Study: System Overview

Projecting instinct evolution scheme on this given scenario is illustrated in Figure 8. In the agent abstraction step (step 1), the layout layer identifies two compression sub-modules: ZOH and wavelet. Each has three configurable variables setting the configuration layer. In step 2, using non-dominated sorting genetic algorithm, the Pareto optimal front of the recognised variables in the configuration layer is generated with the two objectives: information loss and agent power, this is done separately for the two techniques identified in the layout layer. Tables 1 and 2 highlight selected solutions on the Pareto-optimal front with their relative solution configuration, where Δ is tolerable error of compression in bits, F_s is the ADC sampling frequency in Hz and *Bits* are its resolution. Details are found in deliverable 6.2 [13]

Assuming a given preset maximum size of the genotype of the instinct BT (50 integers), the maximum allocated number of bits representing actions, conditions and nodes can be set. Consequently, solutions are selected from the Pareto-optimal front. For the nodes pool a simple *selection* and *sequence* node types are used. In step 5, since the perception layer identifies 6 points and the number of threshold possibilities is set as 3 and assigned to 1, 2 and 3 the total number of conditions is 18. Statistical analysis done on the data available from real world experiment identifies the upper and lower limit for each condition and the position of thresholds (only single value for each threshold was used). Furthermore, the cost function is set as follows:

$$\Sigma = A + \eta C \tag{11}$$

Δ [#]	F_s **[Hz]**	**Bits** [#]	**Power** **[W]**	**Information Loss** [%]
49	487.2376	10	7.82E-06	0.9321E-03
161	159.9791	9	2.64E-06	2.7078E-03
197	151.0169	9	2.07E-06	3.4735E-03
569	100.0152	8	9.58E-07	8.1536E-03
1879	100	8	5.67E-07	21.2695E-03
2973	100.0279	8	5.08E-07	30.5925E-03
7218	100.3022	8	4.74E-07	49.9052E-03
16916	100.0042	8	4.67E-07	63.2073E-03

Table 1 – Selected Pareto optimal solutions for ZOH

Δ [#]	F_s **[Hz]**	**Bits** [#]	**Power** **[W]**	**Information Loss** [%]
270	396.4042	10	3.33E-06	1.4683E-03
377	671	10	4.21E-06	1.3702E-03
210	671	10	5.21E-06	1.2950E-06
587	150.7309	10	1.65E-06	2.4909E-03
2725	100	10	7.79E-07	7.2779E-03
63177	100.0125	10	4.03E-07	47.9297E-03
91020	100.0308	10	3.99E-07	55.9476E-03
91029	100.0339	10	3.99E-07	56.5052E-03

Table 2 – Selected Pareto optimal solutions for Wavelet

Where η is a weighting coefficient, A is the environment property identification probability and C is the relative complexity defined as follows:

$$A = Pr(\text{Agents identifying correctly environment zones}) \tag{12}$$

$$C = 1 - \frac{\text{number of BT nodes}}{\text{max number of BT nodes}} \tag{13}$$

Consequently, it holds $A \in [0, 1]$ and $C \in [0, 1]$.

Using a population of 50, mutation rate = 0.03, crossover rate = 0.5 and for 20 iterations, the final generated instinct in the instruction set form is:

```
state = action(215);
if state ~= 'FAILURE'
    state = action(347);
    if state == 'FAILURE'
        state = condition(138);
    end
end
if state ~= 'FAILURE'
    state = condition(343);
end
if state ~= 'FAILURE'
    state = action(109);
end
if state == 'FAILURE'
    state = condition(335);
end
if state == 'FAILURE'
    state = action(134);
    if state == 'FAILURE'
        state = condition(222);
    end
end
if state == 'FAILURE'
    state = condition(201);
end
if state == 'FAILURE'
    state = action(138);
end
```

4 Conclusion

In this deliverable the different reasons for mimicking the biological instinctive behaviour in Phoenix were presented. This was done by defining the main design objectives of agent's behaviour and its learning process. A proposed scheme for representing instincts was laid out and a genotype to phenotype mapping methodology was illustrated. All this led to the introduction of an instinct evolution scheme.

The presented scheme does not require any mathematical formalisation between tunable variables and user defined objectives, cf. Objective 1. And although achieving such objective usually requires significant computation resources since the search space tends to expand, the scheme structure offers a solution to that by the minimisation of the solution space via introducing actions pool, conditions pool and nodes pool. Additionally, this allows user defined actions, - and node types to be introduced in the solution space, cf. Objective 4. Furthermore, conditions define in a probabilistic manner threshold points and its position based on multiple experiment runs and from numerous agents, cf. Objective 5.

The output of the scheme is an instruction set that uses only conditional constructs to represent the evolved BT and due to the BT structure, embedding a complexity penalty in the fitness function is straightforward, cf. Objective 6. Furthermore, the scheme is integrable with the co-evolution module framework introduced in deliverable 4.1 dubbed as knowledge integrated evolutionary framework (KIEA) [14], cf. Objective 2 and Objective 3.

5 Future Outlook

This deliverable is a critical document for Phoenix since the concept of instinct is linked to many other deliverables in WP4, 5 and 6. Furthermore, what is presented in this deliverable leads the way for further instinct design and implementation on hardware, which is planned in deliverable 7.1. In addition to the software/hardware demonstrator planned in deliverable 7.3.

References

[1] Phoenix Project, Deliverable D4.1. [Online]. Available: https://phoenix-project.eu

[2] K. Y. Scheper, S. Tijmons, C. C. de Visser, and G. C. de Croon, "Behavior trees for evolutionary robotics," *Artificial life*, 2016.

[3] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 5420–5427.

[4] J. R. Koza, M. A. Keane, and M. J. Streeter, "What's ai done for me lately? genetic programming's human-competitive results," *IEEE Intelligent Systems*, no. 3, pp. 25–31, 2003.

[5] A. Arcuri and X. Yao, "Co-evolutionary automatic programming for software development," *Information Sciences*, vol. 259, pp. 412–432, 2014.

[6] G. S. Hornby, A. Globus, D. S. Linden, and J. D. Lohn, "Automated antenna design with evolutionary algorithms," in *AIAA Space*, 2006, pp. 19–21.

[7] J. D. Lohn, D. S. Linden, G. S. Hornby, W. F. Kraus, and A. Rodriguez-Arroyo, "Evolutionary design of an x-band antenna for nasa's space technology 5 mission," in *null*. IEEE, 2003, p. 155.

[8] A. Bundy and L. Wallen, *Context-Free Grammar*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 22–23.

[9] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010.

[10] J. J. Freeman, "A linear representation for gp using context free grammars," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 1998, pp. 72–77.

[11] M. Keijzer, M. O'Neill, C. Ryan, and M. Cattolico, "Grammatical evolution rules: The mod and the bucket rule," in *European Conference on Genetic Programming*. Springer, 2002, pp. 123–130.

[12] C. Ryan, J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *European Conference on Genetic Programming*. Springer, 1998, pp. 83–96.

[13] Phoenix Project, Deliverable D6.2. [Online]. Available: https://phoenix-project.eu/tiki-download_file.php?fileId=245

[14] A. Hallawa, A. Yaman, G. Iacca, and G. Ascheid, "A Framework for Knowledge Integrated Evolutionary Algorithms," in *Lecture Notes on Computer Science*. Springer, 2017, (to appear).