Behavior Trees for Evolutionary Robotics

Scheper, Kirk; Tijmons, Sjoerd; de Visser, Coen; de Croon, Guido

DOI
10.1162/ARTL_a_00192

Publication date
2016

Document Version
Accepted author manuscript

Published in
Artificial Life

Citation (APA)
Scheper, K., Tijmons, S., de Visser, C., & de Croon, G. (2016). Behavior Trees for Evolutionary Robotics. *Artificial Life*, *22*(1), 23-48. https://doi.org/10.1162/ARTL_a_00192

Behaviour Trees for Evolutionary Robotics

Kirk Y.W. Scheper*, Sjoerd Tijmons, Coen C. de Visser, Guido C.H.E. de Croon

Faculty of Aerospace Engineering
Delft University of Technology
2629HS Delft, The Netherlands
Email: k.y.w.scheper@tudelft.nl

Telephone: +31 65 78 82127

***Abstract*—Evolutionary Robotics allows robots with limited sensors and processing to tackle complex tasks by means of sensory-motor coordination. In this paper we show the first application of the Behaviour Tree framework to a real robotic platform using the Evolutionary Robotics methodology. This framework is used to improve the intelligibility of the emergent robotic behaviour as compared to the traditional Neural Network formulation. As a result, the behaviour is easier to comprehend and manually adapt when crossing the reality gap from simulation to reality. This functionality is shown by performing real-world flight tests with the 20-gram DelFly Explorer flapping wing Micro Air Vehicle equipped with a 4-gram onboard stereo vision system. The experiments show that the DelFly can fully autonomously search for and fly through a window with only its onboard sensors and processing. The success rate of the optimised behaviour in simulation is 88% and the corresponding real-world performance is 54% after user adaptation. Although this leaves room for improvement, it is higher than the 46% success rate from a tuned user-defined controller.**

***Index Terms*—Behaviour Tree, Evolutionary Robotics, Reality Gap, Micro Air Vehicle**

I. Introduction

Small robots with limited computational and sensory capabilities are becoming more commonplace. Designing effective behaviour for these small robotic platforms to complete complex tasks is a major challenge. This design problem becomes even more complex when the robots are expected to collaboratively achieve a task as a swarm. A promising methodology to address this problem is found in Evolutionary Robotics (ER), in which a robots controller, and possibly its body, is optimised using Algorithms (EAs) [4, 41]. This approach satisfies given computational constraints, while often resulting in unexpected solutions which exploit sensory-motor coordination to achieve complex tasks [39].

Early investigations into ER used on-line EAs, in which evolution took place directly using the robotic hardware. However, this process is time consuming [17, 42]. With the ever improving computing technologies, off-line EAs based on simulation has become the predominant method to evaluate ER. However, this method has some drawbacks of its own. Simulated environments always differ to some degree from reality. The resultant artifacts from the simulation are sometimes exploited by the evolved solution strategy [17]. As a result the behaviour seen in simulation can often not be reproduced on a real robotic platform. This problem has been termed the reality gap [25, 42].

Many methods have been investigated to reduce this reality gap, which can be separated into three main approaches [4]. The first approach investigates the influence of simulation fidelity on the EA, with investigation focusing on the influence of adding differing levels of noise to the robotic agents inputs and outputs [25, 32, 34]. It was shown that sufficient noise can deter the EA from exploiting artifacts in the simulation but that this approach is generally not scalable as more simulation runs are needed to distinguish between noise and true features. A notable exception to this is the work of Jakobi who discusses the idea of combining limited but varying noise with differing levels of simulation fidelity in what he calls Minimal Simulations [24]. This approach requires the designer to make choices as to which aspects of the environment the robot will use before evolution even begins, limiting the solution space of the EA. Additionally, selecting the type and magnitude of the noise applied requires some foreknowledge of the environmental model mismatch which is not always the case.

The second approach focuses on co-evolution, this approach simultaneously develops a robotic controller which is evaluated in simulation while the simulation model is updated using the performance error with a real world robotic platform [5, 48]. Alternatively, the error between the simulation and real world environment can be used to estimate the suitability of a learnt behaviour on the real robot. A multi-objective function is used to trade off simulated robotic performance and the transferability of the behaviour [29].

The third approach performs adaptation of the real robot behaviour after first being optimised by the EA. This can be achieved using many methods which are differentiated by their level of supervision and how the fitness of the behaviour is determined. One approach involves the use of unsupervised learning where Lamarckian Evolutionary theory is used to evolve the neural structure and ontogenetic learning rules are used to generate a population of adaptive individuals

* Corresponding author

Accompanying video showing some flight test results: https://www.youtube.com/watch?v=CBJOJO2tHf4&feature=youtu.be

[18, 38, 40]. Alternatively, semi-supervised methods such as Reinforcement Learning can be used to retrain the neural nets after evolution [16]. This work shows that systems which adapt to their environments are typically more robust to the reality gap. A typical downside of this approach, however, is that the time needed for the on-line learning to converge may be significant. This is especially problematic for robotic platforms performing complex tasks and operating in an unforgiving environment.

One factor adding to the reality gap problem is that typically Artificial Neural Networks (ANNs) are used to encode the robot behaviour [41]. Although analysis of the evolved ANNs is possible, they do not lend themselves well to manual adaptation hence requiring retraining algorithms to bridge the gap. Encoding the optimised behaviour in a more intelligible framework would aid a user in understanding the solution strategy. It would also help to reduce the reality gap by facilitating manual parameter adaptation when moving to the real robotic platform.

Traditionally, user-defined autonomous behaviours are described using Finite State Machine (FSM) which has also been successfully used within ER [19, 28, 44, 45]. FSMs are very useful for simple action sequences but quickly become illegible as the tasks get more complex due to state explosion [36, 47] This complexity makes it difficult for developers to modify and maintain the behaviour of the autonomous agents.

A more recently developed method to describe behaviour is the Behaviour Tree (BT). Initially developed as a method to formally define system design requirements, the BT framework was adapted by the computer gaming industry to control non-player characters [7, 14]. BTs do not consider states and transitions the way FSMs do, but rather they consider self contained behaviour made up of a hierarchical network of actions and conditions [7, 22]. The rooted tree structure of the BT makes the encapsulated behaviour readily intelligible for users.

Previous work on evolving BTs has been applied to computer game environments where the state is fully known to the BT and actions have deterministic outcomes [31, 43]. The evolution of BTs has not yet been applied to a real world robotic task. Operating in the real world introduces complicating factors such as state and action uncertainty, delays, and other properties of a non-deterministic and not fully known environment.

In this paper, we perform the first investigation into the use of Behaviour Trees in Evolutionary Robotics. Section II will describe the DelFly Explorer [11], the flapping wing robotic platform selected to demonstrate our approach as well as the fly-through-window task it had to perform. This is followed by a detailed description of the BT framework used in Section III. Section IV goes on to describe how offline EAs techniques are used to automatically develop BTs. The results of the optimisation are presented in Section V.

Additionally, the performance of the best individual from the EA is compared to a human user designed BT to show the efficacy of this automatically generated behaviour. Finally, the implementation of both behaviours on the real world DelFly Explorer is described in Section VI to investigate if the reality gap can indeed be actively reduced by a user as a result of the legible behaviour expressed using the proposed method. This is followed by a discussion of how this technique can be scaled to more complex systems and applied to other applications in Section IX.

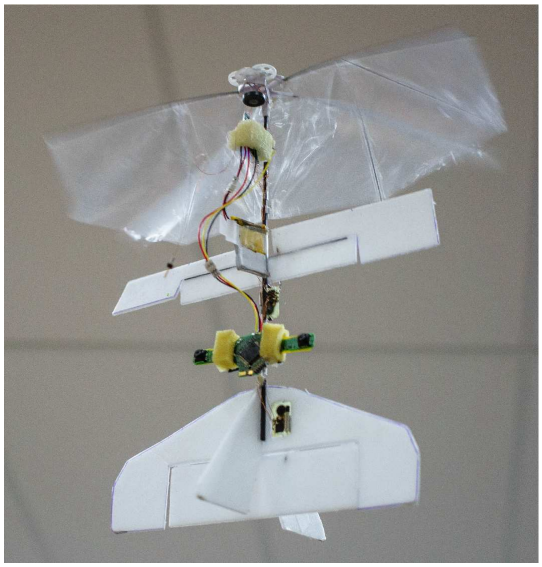

Fig. 1. DelFly Explorer 20-gram flapping wing MAV in flight with 4-gram dual camera payload. An onboard stereo vision algorithm generates a depth map of the environment which is used for autonomous navigation.

II. DelFly Fly-Through-Window

The limited computational and sensory capabilities of the DelFly Explorer make it a challenge to design even the most simple behaviour. As a result, the DelFly Explorer is an ideal candidate for the implementation of ER. We will give a brief description of this platform and its capabilities.

A. DelFly Explorer

The DelFly is a bio-inspired flapping-wing Micro Air Vehicle (MAV) developed at the Delft University of Technology (TU Delft). The main feature of its design is its biplane-wing configuration which flap in anti-phase [10]. The DelFly Explorer is a recent iteration of this micro ornithopter design [11]. In its typical configuration, the DelFly Explorer is $20g$ and has a wing span of $28cm$. In addition to its 9 minute flight time, the DelFly Explorer has a large flight envelope ranging from maximum forward flight speed of $7m/s$, hover, and a maximum backward flight speed of $1m/s$. A photo of the DelFly Explorer can be seen in Figure 1.

The main payload of the DelFly Explorer is a pair of light weight cameras used to perform onboard vision based navigation as shown in Figure 1. Each camera is set to a resolution of 128×96 pixels with a field of view of $60^\circ \times 45^\circ$ respectively. The cameras are spaced $6cm$ apart facilitating stereo-optic vision. Using computer vision techniques these images can be used to generate depth perception with a method

called Stereo Vision [**?**]. This makes the DelFly Explorer the first flapping wing MAV that can perform active obstacle avoidance using onboard sensors facilitating fully autonomous flight in unknown environments [11].

B. Fly-Through-Window Task

In this paper, the DelFly Explorer is tasked to navigate a square room in search for an open window which it must fly through using onboard systems only. This is the most complex autonomous task yet attempted with such a light-weight flapping wing platform. Due to the complexity of finding and flying through a window, we currently limit the task to directional control: height control can be added in future work.

Other work on the fly-through-window task include the H^{2}Bird 13g flapping wing MAV [27]. Unlike the DelFly Explorer, the H^{2}Bird used a ground based camera and off-board image processing to generate heading set-points. In this work the DelFly must perform all tasks using only onboard computation and sensing making the task much more complex than that of the H^{2}Bird.

C. Vision Systems

In the light of the task, the following vision algorithms will be running onboard the DelFly Explorer:

1) LongSeq Stereo Vision: The DelFly Explorer uses a Stereo Vision algorithm called *LongSeq* to extract depth information of the environment from its two onboard optical cameras [11]. The main principle in artificial stereo vision is to determine which pixel corresponds to the same physical object in two or more images. The apparent shift in location of the pixels is referred to as the disparity. This can be applied to entire features, groups of pixels or to individual pixels. The stereo vision algorithm produces a disparity map of all pixels in the images [**?**].

LongSeq is a localised line based search stereo vision algorithm. This is one candidate resulting from the trade-off between computational complexity and image performance made by all image processing algorithms. The relatively low computational and memory requirements of LongSeq makes it a good candidate for application on the limited computational hardware onboard the DelFly Explorer.

2) Window Detection: An Integral Image window detection algorithm is used to aid the MAV in the fly-through-window task. Integral image detection is a high speed pattern recognition algorithm which can be used to identify features in a pixel intensity map [8, 26]. The integral image ($II(x,y)$) is computed as

$$II(x,y) = \sum_{x' \leq x, y' \leq y} I(x',y') \tag{1}$$

where x and y are pixel locations in the image I. As each point of the integral image is a summation of all pixels above and to the left of it, the sum of any rectangular subsection is simplified to the following computation

$$\begin{aligned} rect(x,y,w,h) =& II(x+w,y+h) + II(x,y) \\ & - II(x+w,h) - II(x,y+h) \end{aligned} \tag{2}$$

This method has been previously used to identify a dark window in a light environment by using cascaded classifiers [12]. That algorithm was designed specifically to operate when approaching a building in the daytime on a light day. Naturally, a more generalised method is to apply the same technique described above to the disparity map rather than the original camera images. The disparity map would show a window as an area of low disparity (dark) in an environment of higher disparity (light).

D. SmartUAV Simulation Platform

SmartUAV is a Flight Control Software (FCS) and simulation platform developed in-house at the TU Delft [1]. It is used primarily with small and micro sized aerial vehicles and it notably includes a detailed 3D representation of the simulation environment which is used to test vision based algorithms. It can be used as a ground station to control and monitor a single MAV or swarms of many MAVs. As SmartUAV is developed in-house, designers have freedom to adapt or change the operating computer code at will, making it very suitable for use in research projects.

SmartUAV contains a large visual simulation suite which actively renders the 3D environment around the vehicle. OpenGL libraries are used to generate images on the PC's GPU increasing SmartUAV's simulation fidelity without significant computational complexity. In this paper we will only utilise the simulation capabilities. The BT will be placed in series following the LongSec disparity map generation and the window detection algorithm.

In terms of the larger SmartUAV simulation, the vision based calculations are the most computationally intensive portion making it the limiting factor for the speed of operation of the wider decision process. The higher the decision loop frequency relative to the flight dynamics the longer a single simulation will take. This must be balanced by the frequency at which the DelFly is given control instructions, where generally higher is better. Considering this trade-off, the decision loop was set to run at 10Hz. This is a conservative estimate of the actual performance of the vision systems onboard the real DelFly Explorer.

E. Simplified DelFly Model

The modelling of flapping wing MAV dynamics is an active research area driven by the largely unknown micro scale aerodynamic effects [3, 6, 10]. Due to the lack of accurate models, an existing model of the DelFly II previously implemented based on the intuition of the DelFly designers will be used in this work. This model is not an accurate representation of the true DelFly II dynamics but was sufficient for most vision based simulations previously carried out.

The DelFly II has three control inputs, namely: Elevator (δ_e), Rudder (δ_r) and Thrust (δ_t). The elevator and rudder simply set the control surface deflection and the thrust sets the flapping speed. The actuator dynamics of the DelFly rudder actuator is implemented using a low pass filter with a rise time of 2.2s and a settling time of 3.9s. The elevator deflection and

flapping speed have no simulated dynamics and are directly set to the set-point.

For the simulated flights in this paper, the throttle setting and elevator deflection were held constant at a trim position resulting in a flight speed of $0.5m/s$ and no vertical speed. Additionally, the rudder deflection was limited to a resultant maximum turn rate of $0.4rad/s$ resulting in a minimum turn radius of $1.25m$. The simulated dynamics had no coupling in the flight modes of the simulated DelFly which is a significant simplification of real world flight.

Now, there are some notable differences between the DelFly II and DelFly Explorer. Firstly the Explorer replaces the rudder with a pair of ailerons which allows the DelFly Explorer to turn without the camera rotating around the view axis. Additionally, the DelFly Explorer is $4g$ heavier and has a slightly higher wing flapping frequency. It is expected that the DelFly model mismatch will exaggerate the resultant reality gap.

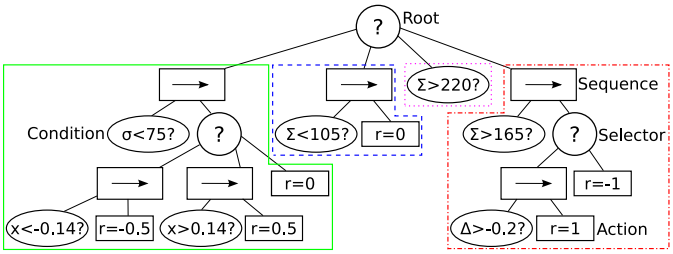

Fig. 2. Typical representation of the Behaviour Tree showing the basic node types and execution flow. The leaf nodes of the tree are composed of Action and Condition nodes whilst the branches are referred to as Composites. All nodes return either Success or Failure. There are two types of Composite nodes used: Selectors and Sequences. Selectors return Success if one of their children is successful and Failure if they all fail. Conversely, Sequences return Failure if one of their children fail and Success is they all succeed. In this example, Condition nodes 3, 13, 15, 17 and 20 return Failure in the given time step or tick. The lightly shaded nodes return Success and the dark nodes evaluate Failure. The nodes with no shading are not evaluated in this tick. The arrows show the propagation of evaluations in the tree.

III. Behaviour Tree Implementation

BTs are depth-first, ordered Directed Acyclic Graphs (DAGs) used to represent a decision process [14]. DAGs are composed of a number of nodes with directed edges. Each edge connects one node to another such that starting at the root there is no way to follow a sequence of edges to return to the root. Unlike FSMs, BTs consider achieving a goal by recursively simplifying the goal into subtasks similar to that seen in the Hierarchical Task Network (HTN) [15]. This hierarchy and recursive action make the BT a powerful way to describe complex behaviour.

A. Syntax and Semantics

A BT is syntactically represented as a rooted tree structure, constructed from a variety of nodes. Each node has its individual internal function whilst all nodes have the same external interface making the structure very modular. When evaluated, each node in a BT has a return status which dictates how the tree will be traversed. In its simplest form, the return statuses are either *Success* or *Failure*. As the terms suggest, Success is returned on the successful evaluation of the node and Failure when unsuccessful. As this does not provide much information as to the condition under which the node failed, some implementations have augmented states such as *Exception* or *Error* to provide this information.

Figure 2 shows a typical BT and node types used in this paper. Basic BTs are made up of three kinds of nodes: *Conditions*, *Actions* and *Composites* [7]. Conditions test some property of the environment whilst Actions allow the agent to act on its environment. Conditions and Actions make up the leaf nodes of the BT whilst the branches consist of Composite nodes. Naturally, leaf nodes are developed for specific robotic platforms dependent on the available sensors and actuators.

Composite nodes however are not platform dependent and can be reused in any BT. Each node requires no information about its location in the tree. Only Composite nodes need to know who its children are in order to direct the flow of execution down the tree. This structure makes BTs inherently modular and reusable.

The tree execution can also be seen in Figure 2. This demonstrates how the Composite nodes determine the execution path of the tree dependant on the return value of their children. To understand this flow structure we must first describe the Composite node in more detail. Although many different types of Composite nodes exist, we will only consider the most basic nodes in this paper: *Selectors* and *Sequences*.

Composites evaluate their children in a fixed order, graphically represented from left to right. Selectors will break execution and return Success when one of its children return Success, or Failure when all of its children return Failure. Conversely, Sequences will break execution and return Failure when one of its children fails, or Success if all of its children return Success. The first node in the tree is called the Root node, which is typically a Selector with no parent. The execution of the behaviour tree is referred to as a *tick*.

This execution framework means that not all nodes are evaluated in every tick. The left most nodes are evaluated first and determine the flow through the tree implementing a sort of prioritised execution.

B. DelFly Implementation

Aside from the generic Sequence and Selector Composite nodes, two condition nodes and one action node were developed for the DelFly, namely: *greater_than*, *less_than* and *set_rudder*. These behaviour nodes are accompanied by a *Blackboard* which was developed to share information with the BT.

The Blackboard architecture implemented for the DelFly contains five entries: *window x location* (x), *window response* (σ), *sum of disparity* (Σ), *horizontal disparity difference* (Δ) and *rudder deflection* (r). The first four are condition variables and the last item is used to set the BT action output. The condition variables are set before the BT is ticked and the outputs are passed to the DelFly FCS after the tick is complete.

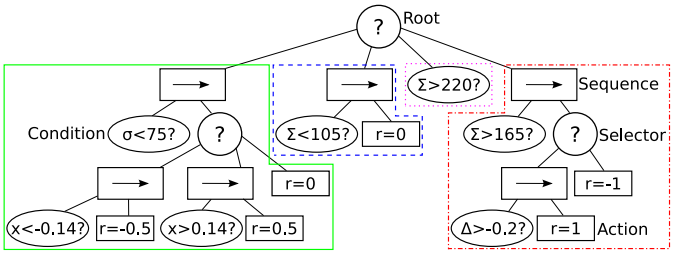

Fig. 3. Graphical depiction of user-defined BT for the fly-through-window task. Different sub-behaviours of the flight are encapsulated in boxes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity, Δ is the horizontal difference in disparity and r is the rudder deflection setting for the simulated DelFly II.

Note that this implementation of a BT has no explicit concept of memory or time.

The Condition nodes check if some environmental variable is greater than or less than a given threshold. This means that each Condition node has two internal settings: the environmental parameter to be checked and the threshold. The Action node *set_rudder* sets the DelFly rudder input and therefore only has one internal setting. Actions were defined to always return Success.

C. User Designed Behaviour Tree

A human designed behaviour was used as a baseline to judge the performance of the genetically optimised solution. The designed tree has 22 nodes and the structure of the BT as shown in Figure 3. The behaviour is made up of four main sub-behaviours:

— window tracking based on window response and location in frame - try to keep the window in the centre of the frame

- - go straight when disparity very low - default action, also helps when looking directly through window into next room

-.- wall avoidance when high disparity - bidirectional turns to avoid collisions with walls, also helps to search for window

... action hold when disparity very high - ensures the chosen action is not changed when already evading a wall

After validation of this BT, it was observed that for 250 random initialisations in the simulated environment, 82% of flights where successful. This behaviour is good but suffers from one main flaw which was observed during the validation. Unwittingly, the bidirectional wall avoidance in a square room can result in the DelFly getting caught in and crashing into corners. There are available methods to correct for this behaviour [46, 49] but as this is a conceptual error typical for human designed systems, we will keep this behaviour as is. Figure 4 shows the path of successful and failed flight realisations of DelFly with the user-defined behaviour.

IV. Evolutionary Algorithm

Evolutionary Algorithms are a population based metaheuristic global optimisation method inspired by Darwinian evolution [20, 23, 30]. A population of feasible solutions for a particular problem are made up of a number of individuals. The fitness of each individual is measured by some user-defined, problem specific, objective function. The fitness of the individuals is evaluated each generation. Successful individuals are selected to generate the next generation using the genetic recombination method crossover. Each generated child may also be subject to mutation where individual parts of their genes are altered. These operations allow the EA to effectively explore and exploit the available search space [33].

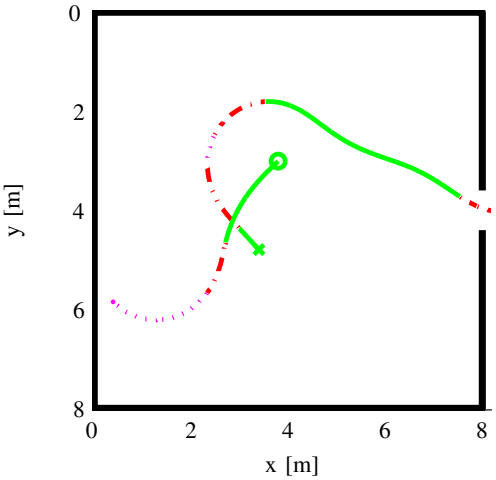

Fig. 4. Path of successful (x) and unsuccessful flight (o) initialisations of DelFly with the user-defined behaviour (top-down view). Line types denote different decision modes: Solid - window tracking; Dash - default action in low disparity; Dot Dash - wall avoidance; Dot - action hold

There are many implementations of EAs, each with a different method to encode the genetic material in the individuals [16, 20, 30**?**]. In this paper we will use an EA to optimise the behaviour for a task using the BT framework. The custom EA for BTs used in this work is described in the following sections.

A. Genetic Operators

a) Initialisation: The initial population of M individuals is generated using the *grow* method [**?**]. Nodes are selected at random to fill the tree with Composite, Action and Condition nodes with equal probability. Once a Composite node is selected, there is equal probability for a Sequence or Selector. This was done as more leaf nodes are typically needed in trees than branch nodes.

The grow method results in variable length trees where every Composite node is initialised with its maximum number of children and the tree is limited by some maximum tree depth. This provides an initial population of very different tree shapes with diverse genetic material to improve the chance of a good EA search.

b) Selection: A custom implementation of Tournament Selection is used in this paper [35]. This is implemented by first randomly selecting a subgroup of s individuals from the population. This subgroup is then sorted in order of their

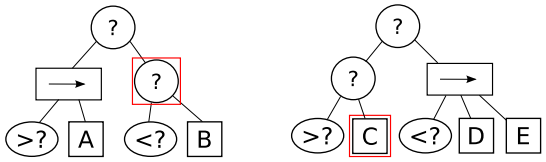

Fig. 5. Sample parent trees with selected nodes for crossover highlighted. Two-parent, single point Crossover is used for evolution.

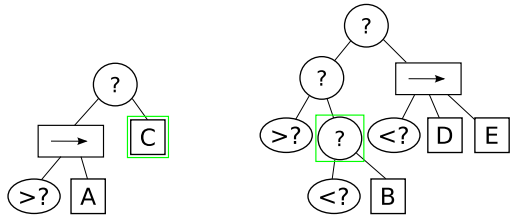

Fig. 6. Children of crossover of parents in Figure 5.

fitness. If two individuals have the same fitness they are then ranked based on tree size, where smaller is better. The best individual is typically returned unless the second individual is smaller, in which case the second individual is returned. This was done to introduce a constant pressure on reducing the size of the BTs.

c) Crossover: Crossover is an operation where the composition of two or more parents is recombined to produce offspring. In this paper we use two-parent crossover to produce two children. Each parent is selected from a different tournament selection. The percentage of the new population formed by Crossover is defined by the Crossover Rate P_c. The point in the BT used to recombine the parents is selected at random.

This selection is independent of its type or its location in the tree. Crossover can be applied to any node location till the maximum tree depth after which nodes are ignored. Figure 5 and Figure 6 graphically show this process.

d) Mutation: Mutation is implemented using two methods, namely: micro-mutation and macro-mutation (also referred to as Headless Chicken Crossover [2]). Micro-mutation only affects leaf nodes and is implemented as a reinitialisation of the node with new operating parameters. Macro-mutation is implemented by replacing a selected node by a randomly generated tree which is limited in depth by the maximum tree depth. This is functionally identical to crossover with a randomly generated BT. The probability that mutation is applied to a node is given by the mutation rate P_m. Once a node has been selected for mutation the probability that macro-mutation will be applied rather than micro-mutation is given by the Headless-Chicken Crossover Rate P_{hcc}.

e) Stopping Rule: Like many optimisation methods, EAs can be affected by overfitting. As a result an important parameter in EA is when to stop the evolutionary process. Additionally, due to the large number of simulations required to evaluate the performance of the population of individuals, placing a limit on the maximum number of generations can help avoid unnecessarily long computational time.

For these reasons, the genetic optimisation has a maximum number of generations (G) at which the optimisation will be stopped. Additionally, when the trees are sufficiently small to be intelligible, the process can be stopped by the user.

B. Fitness Function

The two main performance metrics used to evaluate the DelFly in the fly-through-window task are: Success Rate and Tree Size. The fitness function was chosen to encourage the EA to converge on a population that flies through the window as often as possible. After trying several different forms of fitness functions a discontinuous function was chosen such that a maximum score is received if the MAV flies through the window and a score inversely proportional to its distance to the window if not successful. The fitness F is defined as:

$$F = \begin{cases} 1 & if\ success \\ \frac{1}{1+3|\mathbf{e}|} & else \end{cases} \quad (3)$$

where success is defined as flying through the window and $\mathbf{e}$ is the vector from the centre of the window to the location of the MAV at the end of the simulation. This particular form of fitness function was selected to encourage the DelFly to try to get close to the window with a maximum score if it flies through. The values selected are not very sensitive and were chosen at the discretion of the designer. Changing the gain of the error term effects the selection pressure of the EA.

Although not incorporated in the fitness function, we will also analyse some secondary parameters that are not vital to the performance of the DelFly. These define the suitability of its behaviour from a user point of view and define the characteristics of a given fly-through-window behaviour. These parameters are defined as: Angle of Window Entry, Time to Success and Distance from Centre of Window at Fly-Through.

V. DelFly Task Optimisation

A. Simulated 3D Environment

The environment chosen to evaluate the DelFly in simulation was an $8 \times 8 \times 3m$ room with textured walls, floor and ceiling. A 0.8 0.8m window was placed in the centre of one wall. Another identical room was placed on the other side of the windowed wall to ensure the stereo algorithm had sufficient texture to generate matches for the disparity map when looking through the window.

As it is not the purpose of this research to focus on the vision systems, the environment was rather abundantly textured. A multi-coloured stone texture pattern was used for the walls, a wood pattern was used for the floor and a concrete pattern used for the ceiling as shown in Figure 7. The identically textured walls ensure that the behaviour must identify the window and not any other features to aid in its task.

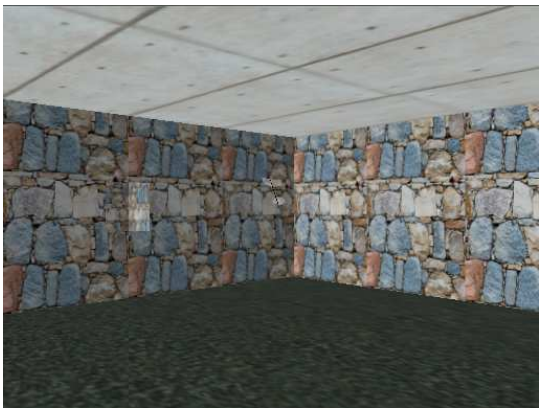

Fig. 7. Virtual $8 \times 8 \times 3m$ room used to evaluate DelFly fly-through-window task showing: virtual DelFly Explorer, textured walls used for stereo vision and target $0.8 \times 0.8m$ window.

B. Experimental Set-up

The evolved DelFly behaviour should be robust and therefore must fly through the window as often as possible. To evaluate this, each individual behaviour must be simulated multiple times in each generation defined by parameter k. Each run is characterised by a randomly initiated location in the room and a random initial heading.

Initially, it was observed that by randomly changing the initialisations in every generation made it difficult for evolution to determine if the behaviour in subsequent generations improved due to its behaviour or due to the initialisation. To remedy this initial conditions are held over multiple generations until the elite members of the population (characterised by P_e) are all successful. Once all the elite members are successful in a given initialisation run, the initial condition in question is replaced by a new random initialisation. Each simulation run is terminated when the DelFly crashes, flies through the window or exceeds a maximum simulation time of $100s$.

For the EA to converge to a near-optimum solution the Crossover rate must be high enough to push the optimisation to exploit the local maxima. Additionally, the mutation rate must be high enough to explore the state space while not too high to prematurely exit current solutions. The characteristic parameters for optimisation shown in this paper are shown in Table I. The parameter combination selected is naturally only one realisation of many possibilities. The relatively large number of runs per individual selected should promote the development of robust flight behaviour. This however increases the total simulation time needed to evaluate each generation hence affecting the choice of population size.

The maximum tree depth is measured with the root node as depth 0. The maximum tree size can be determined by $maxchildren^{maxdepth}$. So a tree depth of 6 with at most 6 children per Composite was used resulting in an upper limiting tree size of over 46000 nodes. This is however not likely as the node type selected in the trees is chosen at random over Composite, Condition and Action.

TABLE I
PARAMETER VALUES FOR THE EVOLUTIONARY COMPUTATION

Parameter	Value
Max Number of Generations (G)	150
Population size (M)	100
Tournament selection size (s)	6%
Elitism rate (P_e)	4%
Crossover rate (P_c)	80%
Mutation rate (P_m)	20%
Headless-Chicken Crossover rate (P_{hcc})	20%
Maximum tree depth (D_d)	6
Maximum children (D_c)	6
No. of simulation runs per generation (k)	6

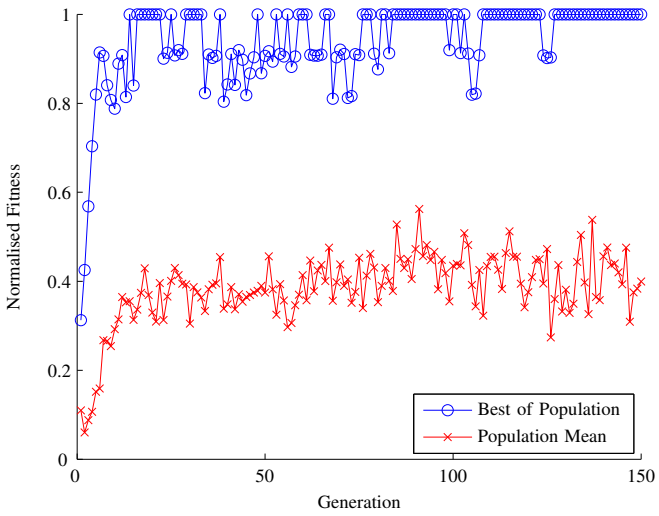

Fig. 8. Progression of the fitness score of the best individual and the mean of the population throughout the genetic optimisation. The fitness value is the mean of the k simulation runs from each generation.

C. Optimisation Results

The main parameter which dictates the progress of the genetic optimisation is the mean fitness of the population. Figure 8 shows the population mean fitness as well as the mean fitness of the best individual in each generation. It can be seen in Figure 8 that at least one member of the population is quickly bred to fly through the window quite often. Additionally, as the generations progress and new initialisations are introduced the trees have to adjust their behaviour to be more generalised. The mean fitness also improves initially and then settles out at around the 0.4 mark. The fact that this value doesn't continue to increase suggests that the genetic diversity in the pool is sufficient to avoid premature conversion of the EA.

The other main parameter which defines the proficiency of the BTs is the tree size. The mean tree size of the population as well as the tree size of the best individual from each generation is shown in Figure 9. This figure shows that the average tree size began at about 5000 nodes and initially increases to 7000 before steadily dropping to around 1000 nodes at generation 50. The trees then slowly continue to reduce in size and eventually drop below 150 nodes. The best individual

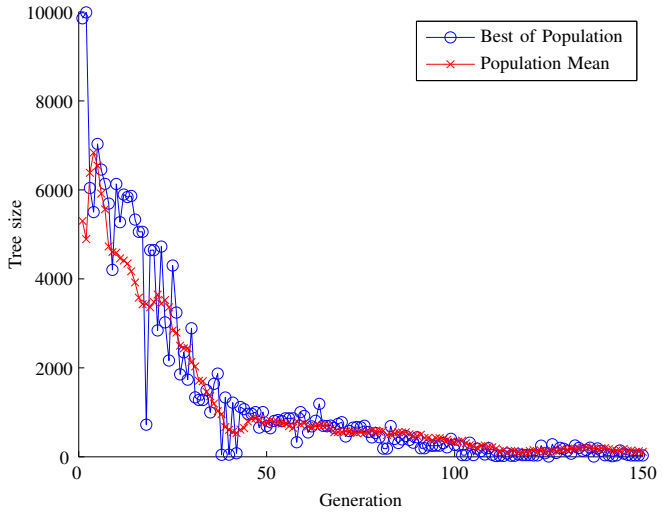

Fig. 9. Progression of the number of nodes in the best individual and the mean of the population.

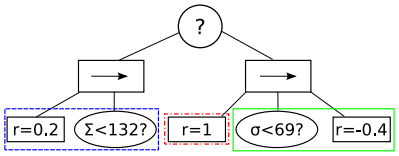

Fig. 10. Graphical depiction of genetically optimised BT. Different sub-behaviours of the flight encapsulated by boxes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity, Δ is the horizontal difference in disparity and r is the rudder deflection setting for the simulated DelFly II.

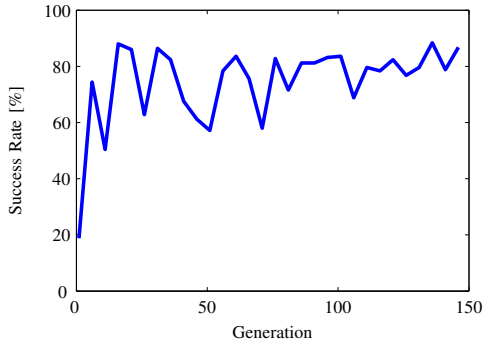

Fig. 11. Progression of the validation score of the best individual of each generation subjected to the same set of 250 spacial initialisations in the simulated room.

in the population oscillated around this mean value. The best individual after 150 generations had 32 nodes. Pruning this final BT, removing redundant nodes that have no effect on the final behaviour, resulted in a tree with 8 nodes. The structure of the tree can be seen graphically in Figure 10.

Figure 11 shows the progression of the validation success rate for the best individual of each generation. It can be seen that the score quickly increases and oscillates around about 80% success. In early generations the variation of success rate

TABLE II
SUMMARY OF VALIDATION RESULTS

Parameter	user-defined	genetically optimised
Success Rate	82%	88%
Tree size	26	8
Mean flight time $[s]$	32	40
Mean approach angle $[^\circ]$	21	34
Mean distance to centre $[m]$	0.08	0.15

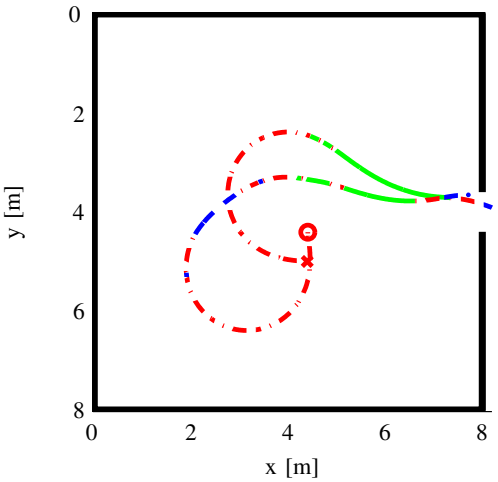

Fig. 12. Path of successful (x) and unsuccessful (o) flight initialisations of DelFly with the genetically optimised behaviour (top-down view). Line styles denote different decision modes: Solid - window tracking; Dash - default action in low disparity; Dash Dot - wall avoidance.

from one generation to the next is larger than later generations.

Figures 9 and 11 suggest that the population quickly converges to a viable solution and then continues to rearrange the tree structure to result in ever smaller trees. The fact that the best individual of each population does not improve much above the 80% mark possibly indicates that the selected initial conditions used for training are in-fact not representative for the full set of initial conditions. One method to make the initial conditions more difficult is to adapt the environment to actively challenge the EA in a sort of predator-prey optimisation. Alternatively, the fact that the behaviour does not continue to improve over the 80% mark may indicate that the sensory inputs used by the DelFly are not sufficient.

The optimised BT was put through the same validation set as used with the user-defined behaviour resulting in a success rate of 88%. The performance characteristics of the best individual from the optimisation as compared to those from the user-defined BT is summarised in Table II. The optimised BT has slightly higher success rate than the user-defined BT but with significantly less nodes. The results of the secondary parameters suggest that the genetically optimised behaviour typically has a sharper window entry angle and enters the window closer to the edge than the user-defined behaviour. It also has a longer time to window fly-through as it circles the room more often than the user-defined behaviour.

This result highlights the fact that EAs typically only optimise the task explicitly described in the fitness function,

Fig. 13. Photograph showing the room environment used to test the DelFly Explorer for the fly-through-window task. Inset is collage of DelFly as it approaches and flies through window.

sometimes at the cost of what the user might think is beneficial. The successful flight shown in Figure 12 shows that the behaviour correctly avoids collision with the wall, makes its way to the centre of the room and then tracks into the window. Analysing the BT from Figure 10, the logic to fly through the window can be separated into three sub-behaviours:

- - slight right turn default action when disparity low
- .- max right turn to evade walls if disparity high (unidirectional avoidance)
- — if window detected make a moderate left turn

Although this very simple behaviour seems to be very successful, Figure 12 also highlights one pitfall of this solution. As the behaviour does not use the location of the window in the frame for its guidance it is possible to drift off centre and lose the window in frame and enter a wall avoidance turn quite close to the wall resulting in a collision.

These results show that based on the given fitness function and optimisation parameters the genetic optimisation was very successful. The resultant BT was both smaller and better performing than the user-defined tree.

VI. DelFly Onboard Flight Testing

The BT was implemented on the camera module of the DelFly Explorer which is equipped with a $STM32F405$ processor operating at $168MHz$ with $192kB$ RAM. The BT is placed in series with the stereo vision and window detection algorithms as was done in simulation and was found to run at $\sim 12Hz$. The commands were sent from the camera module to the DelFly Explorer flight control computer using serial communication. The DelFly flight control computer implements these commands in a control system operating at $100Hz$.

A. Test 3D Environment

The environment designed to test the MAV was a $5 \times 5 \times 2m$ room with textured walls. A $0.8 \times 0.8m$ window was placed in the centre of one wall. The area behind the window was a regular textured area. Artificial texture was added to the environment to ensure we had good stereo images from the DelFly Explorer onboard systems. This texture was in the form of newspapers draped over the walls at random intervals. A sample photograph of the room can be seen below in Figure 13.

TABLE III
Summary of the Reality Gap

Parameter	Simulated	Reality
Flight Speed $[m/s]$	0.5	0.5
Minimum Turn Radius $[m]$	1.25	0.5
Actuator Response Time $[s]$	2.2	¡1
Decision Loop Speed $[Hz]$	10	12
Actuator Deflection	Symmetric	Asymmetric
Environmental	No Disturbances	Drafts

B. Experiment Set-up

At the beginning of each run, the DelFly was initially flown manually and correctly trimmed for flight. It was then flown to a random initial position and pointing direction in the room. At this point the DelFly was set to autonomous mode where the DelFly flight computer implements the commands received from the BT. The flight continued until the DelFly either succeeded in flying through the window, crashed or the test took longer than $60s$. As the BT controls the horizontal dynamics only, the altitude was actively controlled by the user during flight which was maintained around the height of the centre of the window.

All flights were recorded by video camera as well as an Optitrack vision based motion tracking system [37]. Optitrack was used to track the DelFly as it approached and flew through the window to determine some of the same metrics of performance that were used in simulation. As a result, information on the success rate, flight time, angle of approach and offset to the centre of the window can be determined.

VII. Crossing the Reality Gap

The flight speed of the DelFly was set to $\sim 0.5m/s$, the same as was used in simulation. However, there were significant differences observed between the system simulated in SmartUAV and that in the flight tests. The most significant observations are summarised in Table III. In short, the turn radius was smaller and the actuator response was faster and asymmetric. Additionally, aileron actuation would result in a reduction in thrust meaning that active altitude control was required from the user throughout all flights. It was also observed that there were light wind drafts around the window which affected the DelFly's flight path. These drafts would typically slow down the DelFly's forward speed and push it to one side of the window.

With these significant differences between the model used to train the BTs and the real DelFly there was a clear reality gap present. Initially both behaviours were not successful in flying through the window. To adjust the behaviour to improve the performance we first considered the definition of success as defined by Jakobi [24]. In his paper he suggested that the performance of the robotic system should be judged on a subjective measure of how reliably the robot performs the task in reality with no consideration to how the behaviour achieves

the task objective. In the case of this paper, that would simply be defined as how often the DelFly flies through the window.

We initially tried to directly adjust the behaviour in reality without comparing it to the behaviour seen in simulation. To improve the fly-through-window performance we mainly considered the final portion of the flight but this proved ineffective. This results from the fact that the embodied agent's success is tightly coupled with interaction of the robot's sub-behaviours during the entire flight. For example, the way the DelFly wall avoidance sub-behaviour performed defined its approach to the window in such a way that the window approach sub-behaviour would be successful. This suggests then that to achieve a task reliably in reality the robot must behave similarly to that observed in simulation for all sub-behaviours.

The insight into what parameters to change and how, comes from the user's understanding of the BT. From this the user can identify individual sub-behaviours. The technique of grouping nodes into sub-behaviours can be seen in Figures 3 and 10. This segmentation of the behaviour helps to identify individual gaps simplifying the behaviour update process.

To demonstrate this let us first look at the evolved behaviour tree shown in Figure 10 which can be considered as made up of three sub-behaviours. Let us first look at the window detection sub-behaviour. We flew the DelFly around our test room and observed the window response value was never achieved with the certainty value of 69 (a lower value represents higher certainty that a window is in the frame). We increased the threshold of node 7 till the node would be activated by the window but false positives from other locations would not be likely.

Let us now investigate the wall avoidance sub-behaviour. This mode is entered when the total disparity is larger than a threshold set by node 3. Observing the behaviour in Figure 12, the DelFly tries to circle in around the centre of the room entering the wall avoidance mode at $\sim 4m$ from the wall in the $8 \times 8m$ room. This would suggest that the real DelFly should enter this mode at $\sim 2.5m$ in the real $5 \times 5m$ room so the threshold in node 3 should be changed accordingly.

It should be noted that it appears that evolution has optimised the DelFly behaviour to fly through windows in square rooms. The approach of avoiding walls at a fixed distance to line the DelFly up for the window entry would be more difficult if the window was not in the centre of the wall or if the room size changed. This reiterates the strong coupling between optimised behaviour and the environment that is characteristic of ER. It is therefore essential to vary the environment sufficiently to encourage the EA to converge to solutions robust to changes in the environment. Last but not least, applying this to the wall avoidance action, the simulated DelFly had a minimum turn radius of $1.25m$ which was much smaller in reality. A scaling factor was applied to increase the turn radius to that seen in simulation.

Using this approach, tuning these parameters took about 3 flights of about 3 minutes each to result in behaviour similar to that seen in simulation. The updated behaviour can be seen in Figure 14. This same approach was used with the user-defined BT with significantly more nodes and took a total of 8 flights of about 3 minutes each to tune the parameters to mimic the behaviour observed in simulation. The updated behaviour can be seen in Figure 15.

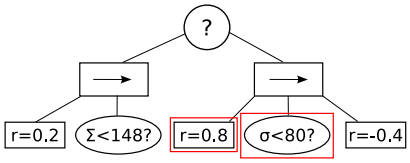

Fig. 14. Graphical depiction of genetically optimised BT after modification for real world flight. Encapsulating boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity, Δ is the horizontal difference in disparity and r is the aileron deflection setting for the DelFly Explorer.

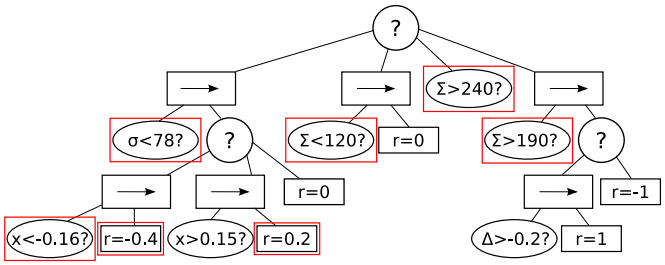

Fig. 15. Graphical depiction of user-defined BT after modification for real world flight. Encapsulating boxes highlight updated nodes. x is the position of the centre of the window in frame, σ is window response value, Σ is sum of disparity, Δ is the horizontal difference in disparity and r is the aileron deflection setting for the DelFly Explorer.

TABLE IV
SUMMARY OF FLIGHT TEST RESULTS

Parameter	user-defined	genetically optimised
Success Rate	46%	54%
Mean flight time $[s]$	12	16
Mean approach angle $[^\circ]$	16	37
Mean distance to window centre $[m]$	0.12	0.12

VIII. Flight Test Results

26 test flights were conducted for both the user-defined behaviour as well as the genetically optimised BT[1]. The results of the tests are summarised in Table IV.

It can be seen that the success rate of both behaviours is reduced for both behaviours but notably, the relative difference of the two behaviours is maintained. Additionally, the other performance parameters which are the characteristic behaviour descriptors are similar to that seen in simulation. This suggests that the user adaptation of the real behaviour to emulate the simulated behaviour was successful. The relative performance of the behaviours is also similar to that seen in simulation. The mean flight time of the behaviours was reduced but notably the relative flight times of the behaviours is the same as seen in simulation. The reduction in the time to success can be

[1]An accompanying video with some of the test flights can be found at: https://www.youtube.com/watch?v=CBJOJO2tHf4&feature=youtu.be

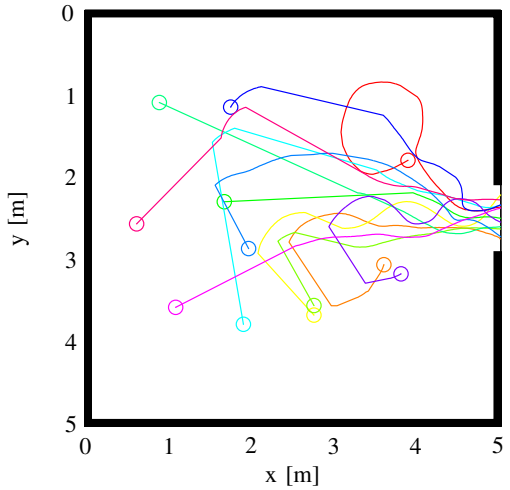

Fig. 16. Flight path tracks of the last 7 seconds of all successful flights for the user-defined behaviour. o represents start location of each flight.

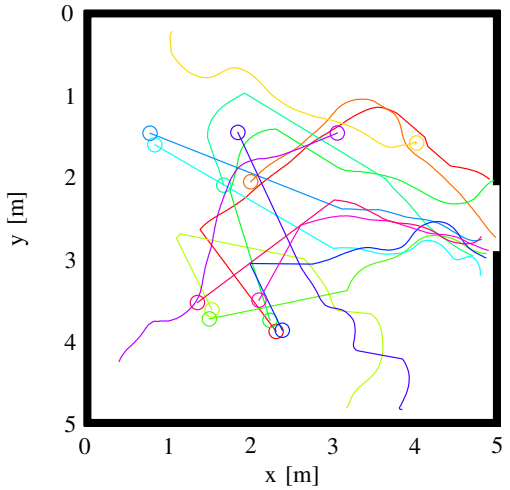

Fig. 17. Flight path tracks of the last 7 seconds of all unsuccessful flights for the user-defined behaviour. o represents start location of each flight.

explained by the reduced room size. The mean distance to the centre of the window was higher for the user-defined behaviour than observed in simulation. This can be the result of the drafts around the window pushing the DelFly to the edges of the window. This draft would also push the approaching DelFly into the window edge on some approaches.

The time to success was lower for both behaviours as compared to the values observed in simulation. This is mainly due to the smaller room size used in reality. Notably, the user-defined behaviour showed the characteristic failure of being caught in corners. This happened 4/26 flights for the user-defined behaviour but not once in the genetically optimised behaviour. This is representative of the observations of the behaviour in simulation, a fundamental deficiency of the bi-directional wall avoidance in a room with corners. This observation additionally suggests that the behaviour seen in simulation is effectively transferred to the real DelFly. Figures 16 and 17 show the last 7 seconds of the user-defined behaviour for all flights grouped in successful and unsuccessful tests respectively. The Optitrack flight tracking system did not successfully track the DelFly in all portions of the room resulting in some dead areas but did accurately capture the final segment of the window approach. These plots show that the DelFly tried to approach and fly through the window from various areas of the room at various approach angles. Approaches from areas of high approach angle typically resulted in a failed flight as the DelFly would hit the edge of the window. Additionally, the crashes in the wall due to being caught in corners can also be seen. Figure 18 shows one full successful and unsuccessful flight of the DelFly user-defined behaviour.

Similarly, Figures 19 and 20 show the successful and unsuccessful flights of the genetically optimised behaviour as captured from the Optitrack system. In these figures it can be seen that the flight tracks of genetically optimised behaviour are tightly grouped with the same behaviour repeated over multiple flights. The DelFly always approaches from about the

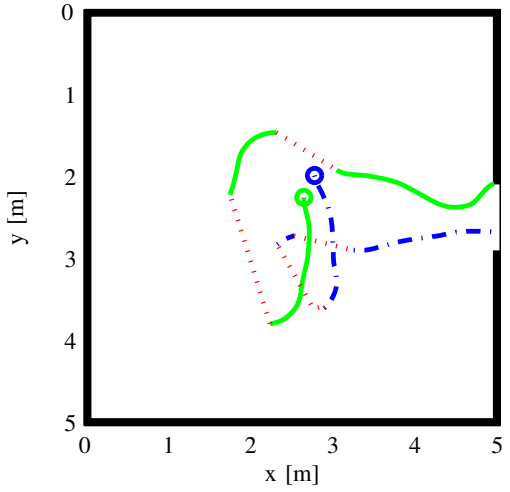

Fig. 18. Flight path tracks showing one complete successful (dash dot) and unsuccessful (solid) flight for the user-defined behaviour. o represents start location of test. Doted path shows where tracking system lost lock of the DelFly.

centre of the room with a coordinated left-right turn described earlier. It can be seen that some of the unsuccessful flights occur when the DelFly makes an approach from farther way than normal so the coordination of the left-right turning is out of sync causing the DelFly to drift off course and hit the window edge. Figure 21 shows one entire successful and unsuccessful flight of the genetically optimised behaviour in more detail. The typical failure mode was turning into the edge of the window in the final phase of the flight.

This is likely mainly due to the drafts around the window. Additionally, the faster decision rate of the BT in reality combined with the faster dynamics of the vehicle may play a role here as well. The fact that the real world test was conducted in a different sized room than tested in simulation would have an effect on the success rate. In the future it would be interesting to observe the converged behaviour if the

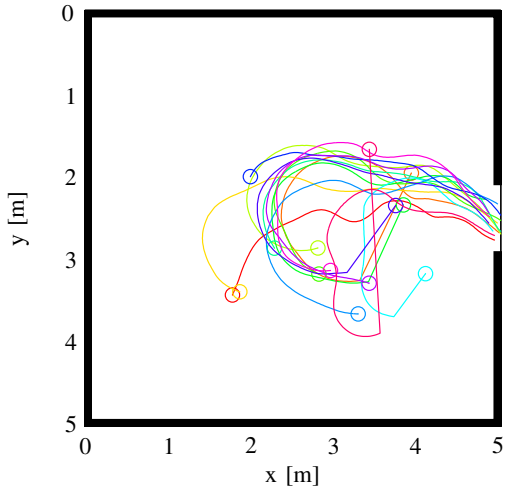

Fig. 19. Flight path tracks of the last 7 seconds of all successful flights for the genetically optimised behaviour. o represents start location of each flight.

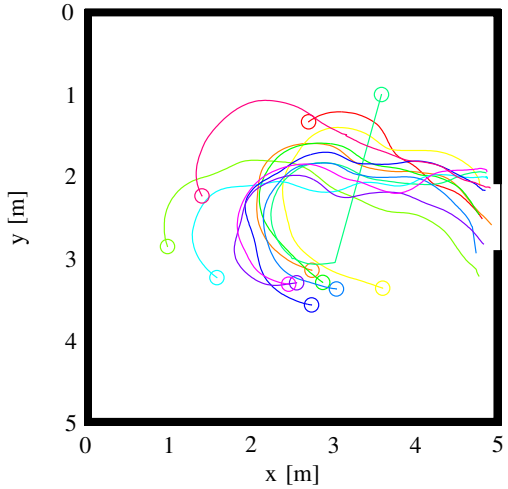

Fig. 20. Flight path tracks of the last 7 seconds of all unsuccessful flights for the genetically optimised behaviour. o represents start location of each flight.

simulated room were not kept constant during evolution. It is expected that this would result in behaviour more robust to changes in the environment. The failure mode of hitting into the window edge for both behaviours can be in part the result of the drafts observed around the window or in part due to the lack of detailed texture around the window. These external factors would affect the two behaviours equally so would not affect the comparison of behaviours.

The fact that both the user-defined and genetically optimised behaviours were initially not able to fly through the window but after user adaptation were able to fly through more than 50% of the time shows that the reality gap was actively reduced by the user. These results show that it is feasible to automatically evolve behaviour on a robotic platform in simulation using the BT description language. This method gives the user a high level of understanding of the underlying behaviour and the tools to adapt the behaviour to improve

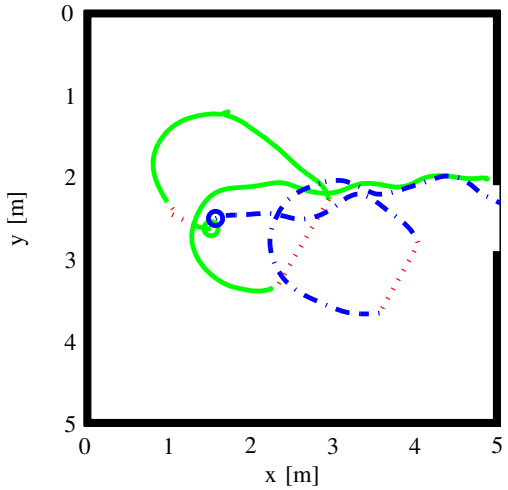

Fig. 21. Flight path tracks showing one complete successful (dash dot) and unsuccessful (solid) flight for the genetically optimised behaviour. o represents start location of test. Doted path shows where tracking system lost lock of the DelFly.

performance and reduce the reality gap. Using this technique an automated behaviour was shown to be at least as effective as, if not better than, a user-defined system in simulation with similar performance on a real world test platform.

IX. Discussion

A. Behaviour I/O Abstraction

In this paper we use the descriptive and user legible framework of the BT to improve the user's understanding of the solution strategy optimised through evolution. With this insight the user can identify and reduce the resultant reality gap when moving from simulation to reality. This approach therefore necessitates that the elements of the tree are also conceptually tangible for the user, as such a higher abstract level was used for the sensory inputs. Unlike standard approaches which use ANNs where raw sensor data is used as input, we first preprocess the data into a form that a user can understand. The only question is then, how do we determine what is the best set of inputs to the robotic platform that will facilitate a robust and effective solution to be optimised by evolution.

Now, compared to typical ER approaches, preprocessing the inputs may affect the level of emergence of the EA such as hat seen in Harvey et al. [21]. That paper demonstrated a robot completing an object detection task which was simplified by an EA to the correlation of just a few image pixels. This level of optimisation may not be possible if the inputs are preprocessed. However, preprocessing the input data typically reduces its dimensionality, thereby reducing the search space of the EA. This reduction in search space is crucial as we implement this technique on even more complex task and robotic platforms.

With regard to the actions, robotic outputs are typically not robust as they are susceptible to unmodelled simulator dynamics and changes in the operating environment. For example, in this paper we set the output of the BT to be

the rudder deflection which in hindsight is not a very robust parameter to control. It may have been more effective to have controlled the turn rate and have a low level, closed loop control system controlling the actuator deflection. The closed loop controller would reduce the BT's reliance on the flight model in simulation. This would make the behaviour more robust on the real robot inherently reducing the reality gap.

The concept of using nested loops to bound control systems in order to improve robustness is a concept long used in control theory. Considering the reality gap, recent work suggests that by limiting the EA to a set of predefined modules can actually improve the optimised behaviour to the eventual reality gap [19]. In this work, Francesca et al. compare an optimised FSM using a limited set of predefined modules to a traditional system using an ANN. The two systems performed similarly in simulation but the ANN performed significantly worse in reality whilst the FSM maintained its performance. Francesca et al. present their work in the context of the bias-variance trade-off where they suggest that the introduction of the appropriate amount of bias will reduce the variance of the optimised system thereby improving its generality. Bias can be introduced to an optimisation problem by limiting the representational power of the system, which in this case is achieved by limiting the options of the optimisation to a limited input-output state space [13]. This idea can also be considered in this paper where the limitation of the state space is not a hindrance or a limitation of the system but is in fact a benefit of this approach.

The abstraction of the behaviour from the low level sensor inputs and actuator output importantly not only introduces a bias but additionally shields the behaviour from the simulation mismatch causing the reality gap. The improved intelligibility in combination with the improved generalizability and robustness to the reality gap should ultimately make the approach presented in this paper more suitable for extensive use in real robots attempting complex tasks than conventional ER approaches.

B. Scalability

The task completed in this paper is more complex than other ER tasks typically quoted in literature. Yet in the larger scale of autonomous navigation this task is only just a start. To facilitate this growing task complexity we will recommend some points for future research. Firstly, it is interesting to investigate the implementation of memory and time to the BT.

Memory could be implemented as elements of the BlackBoard that are not outputs of the BT to the platform but rather just internal variables. Time could be added by including a Running state to the nodes where they would hold till the action is completed. Alternatively, an explicit Timer node could be added that would run for a given number of ticks. Another point worth consideration is the addition of a Link node to the BT framework. This node creates a symbolic link to a static BT branch outside of the tree. Evolution could select branches of its own behaviour which could be linked and reused in other parts of the tree. This should help the optimisation to reuse already developed behaviour effectively throughout the tree. This would provide the EA with not only the raw materials to build the behaviour but the ability to save combinations of these raw materials in a blueprint which can be reused at will. With that said, the technique described in this paper is dependent on the user's understanding of the underlying robotic behaviour, so how does this change with the growing task complexity? We showed in this paper that the BT can be broken down into sub-behaviours which helps the user to understand the global behaviour. The prioritised selection of behaviours based on their location in the tree creates an inherent hierarchical structure. This structure will automatically group the nodes of a sub-behaviour spatially in the tree.

This makes the identification of the sub-behaviours straight forward. Tuning of the sub-behaviours would be accomplished using a divide and conquer approach, one sub-behaviour at a time.

C. Evolution of Behaviour Trees for Behavioural Modelling

The BT framework could also be used to model existing behaviour of robots or animals. This would be in a similar vein as a recent ER study, in which the insight into the evolved neural controller's strategy was verified by constructing an equivalent FSM controller [9]. Instead of manually designing such a controller, EAs could be used to optimise a BT to best approximate the behaviour of a robot or animal. The BT, optimised to mimic reality, would give researchers increased insight into the underlying system dynamics. To mention a few examples, this approach can be applied to: self-organisation, swarming, emergence and predator-prey interaction.

X. Conclusion

We conclude that the increased intelligibility of the Behaviour Tree framework does give a designer increased understanding of the automatically developed behaviour. The low computational requirements of evaluating the Behaviour Tree framework makes it suitable to operate onboard platforms with limited capabilities as it was demonstrated on the 20g DelFly Explorer flapping wing MAV. It was also demonstrated that the Behaviour Tree framework provides a designer with the tools to identify and adapt the learnt behaviour on a real platform to reduce the reality gap when moving from simulation to reality.

Future work will investigate further into optimising the parameters of the Evolutionary Learning used in this paper. Multi-objective fitness functions and co-evolution of behaviour and simulated environment are interesting directions to investigate. Additionally, work will be done on investigating how Behaviour Trees scale within Evolutionary Learning, both in terms of Behaviour node types but also in task complexity. Additionally, automated control will be extended to height control for fully autonomous flight.

References

1. Matthijs H.J. J. J Amelink, Max Mulder, M. M van Passen, M M van Paassen, M M van Passen, and M M van

Paassen. Designing for Human-Automation Interaction: Abstraction-Sophistication Analysis for UAV Control. In S I Ao, Oscar Castillo, Craig Douglas, David Dagan Feng, and Jeong-A Lee, editors, *International MultiConference of Engineers and Computer Scientists*, volume I, pages 318–323, Hong Kong, mar 2008. IAE.

2. Peter J Angeline. Subtree Crossover: Building Block Engine or Macromutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B Fogel, Max Garzon, Hitoshi Iba, and Rick L Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, jul 1997. Morgan Kaufmann.
3. S Ansari, R Żbikowski, and K Knowles. Aerodynamic modelling of insect-like flapping flight for micro air vehicles. *Progress in Aerospace Sciences*, 42(2):129–172, 2006.
4. Josh C Bongard. Evolutionary Robotics. *Communications of the ACM*, 56(8):74–83, 2013.
5. Josh C Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–21, 2006.
6. J V Caetano, J Verboom, C C de Visser, G C H E de Croon, B D W Remes, C de Wagter, and M Mulder. Linear Aerodynamic Model Identification of a Flapping Wing MAV Based on Flight Test Data. *International Journal of Micro Air Vehicles*, 5(4):273–286, 2013.
7. Alex J Champandard. Behavior Trees for Next-Gen Game AI. In *Game Developers Conference '07*, pages 1–96, San Francisco, CA, 2007. GDC.
8. Franklin C Crow. Summed-Area Tables for Texture Mapping. *SIGGRAPH Computer Graphics*, 18(3):207–212, jul 1984.
9. G C H E de Croon, L M O Connor, C Nicol, and D Izzo. Evolutionary Robotics Approach to Odor Source Localization. *Neurocomputing*, 121(1):481–497, dec 2013.
10. G C H E de Croon, K M E de Clercq, R Ruijsink, B D W Remes, and Christophe de Wagter. Design, aerodynamics, and vision-based control of the DelFly. *International Journal of Micro Air Vehicles*, 1(2):71–97, 2009.
11. C de Wagter, S Tijmons, B D W Remes, and G C H E de Croon. Autonomous Flight of a 20-gram Flapping Wing MAV with a 4-gram Onboard Stereo Vision System. In *International Conference on Robotics and Automation*, pages 4982–4987, Hong Kong, jun 2014. IEEE.
12. Christophe de Wagter, Alison A Proctor, and Eric N Johnson. Vision-Only Aircraft Flight Control. In *Digital Avionics Systems Conference*, volume 2, pages 8.B.2 – 1–11, Indianapolis, IN, oct 2003. IEEE.
13. Thomas G Dietterich, Eun Bae Kong, and Dearborn Hall. Machine Learning Bias , Statistical Bias , and Statistical Variance of Decision Tree Algorithms. Technical report, Department of Computer Science, Oregon State University, Corvallis, OR, 1995.
14. R G Dromey. From Requirements to Design: Formalizing the Key Steps. In *First International Conference on Software Engineering and Formal Methods*, pages 2–11, Brisbane, sep 2003. IEEE.
15. K Erol, James Hendler, and Dana S Nau. HTN Planning: Complexity and Expressivity. In *Twelfth National Conference on Artificial Intelligence*, pages 1123–1128, Menlo Park, CA, 1994. AAIA Press.
16. Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from Architectures to Learning. *Evolutionary Intelligence*, 1(1):47–62, jan 2008.
17. Dario Floreano and Francesco Mondada. Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven robot. In D Cliff, P Husbands, J-A Meyer, and S Wilson, editors, *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pages 421–430, Cambridge, MA, 1994. MIT Press.
18. Dario Floreano and Francesco Mondada. Evolution of homing navigation in a real mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 26(3):396–407, 1996.
19. Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Vito Trianni, and Mauro Birattari. AutoMoDe: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, 8:89–112, 2014.
20. David E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, Boston, MA, 1989.
21. Inman Harvey, Phil Husbands, and D Cliff. Seeing the light: Artificial evolution, real vision. In D Cliff, J-A Meyer, and S Wilson, editors, *Proceedings of the Third International Conference on Simulation of Adaptive Behavior from Animals to Animats 3*, volume 1994, pages 392–401, Cambridge, MA, aug 1994. MIT Press Bradford Books.
22. Frederick W. P. Heckel, G. Michael Youngblood, and Nikhil S. Ketkar. Representational Complexity of Reactive Agents. In *Computational Intelligence and Games*, pages 257–264, Dublin, aug 2010. IEEE.
23. Frederick S Hiller and Gerald J Lieberman. *Introduction to Operations Research*. McGraw-Hill, New York, 9 edition, 2010.
24. Nick Jakobi. Evolutionary Robotics and the Radical Envelope-of-Noise Hypothesis. *Adaptive Behaviour*, 6(2):325–368, 1997.
25. Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics. In Federico Morán, Alvaro Moreno, Juan Julián Merelo, and Pablo Chacón, editors, *Advances in Artificial Life*, pages 704–720, Granada, jun 1995. Springer Berlin Heidelberg.
26. Michael J Jones and P Viola. Robust Real-Time Object Detection. *International Journal of Computer Vision*, 57(2):137–154, 2001.
27. Ryan C Julian, Cameron J Rose, Humphrey Hu, and Ronald S Fearing. Cooperative Control and Modeling for Narrow Passage Traversal with an Ornithopter MAV and

Lightweight Ground Station. In *International conference on Autonomous agents and multi-agent systems*, pages 103–110, St. Paul, MN, may 2013. IFAAMAS.

28. Lukas König, Sanaz Mostaghim, Hartmut Schmeck, Lukas Konig, Sanaz Mostaghim, and Hartmut Schmeck. Decentralized Evolution of Robotic Behavior using Finite State Machines. *International Journal of Intellegent Computing and Cybernetics*, 2(4):695–723, 2009.
29. Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics. *Transactions on Evolutionary Computation*, 17(1):122–145, feb 2013.
30. John R Koza. Genetic Algorithms and Genetic Programming, 2003.
31. Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving Behaviour Trees for the Commercial Game DEFCON. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I Esparcia-Alcazar, Chi-Keong Goh, Juan J Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios N Yannakakis, editors, *Applications of Evolutionary Computation*, volume 6024, pages 100–110, Essex, 2010. Springer Berlin Heidelberg.
32. Lisa Meeden. Bridging the Gap Between Robot Simulations and Reality with Improved Models of Sensor Noise. In John R Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoym Deb, Marco Dorigo, David B Fogel, Max H Garzon, David E Goldberg, Hitoshi Iba, and Rick L Riolo, editors, *Genetic Programming*, pages 824–831, San Francisco, CA, jul 1998. Morgan Kaufmann.
33. Mitchell Melanie and M Mitchell. *No Title*. Cambridge, MA.
34. Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving Mobile Robots in Simulated and Real Environments. *Artificial life*, 2(4):417–434, jan 1995.
35. Brad L Miller and David E Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*, 9(3):193–212, 1995.
36. Ian Millington and John Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 2009.
37. Natural Point Inc. Optitrack, 2014.
38. S. Nolfi and D. Parisi. Learning to Adapt to Changing Environments in Evolving Neural Networks. *Adaptive Behavior*, 5(1):75–98, 1996.
39. Stefano Nolfi. Power and the Limits of Reactive Agents. *Neurocomputing*, 42(1-4):119–145, 2002.
40. Stefano Nolfi and Dario Floreano. Learning and Evolution. *Autonomous Robots*, 7:89–113, 1999.
41. Stefano Nolfi and Dario Floreano. *Evolutionary Robotics: The Biology, Intelligence and Technology*. MIT Press, Cambridge, MA, 2000.
42. Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to Evolve autonomous robots: Different Approaches in evolutionary robotics. In R Brooks and P Maes, editors, *Artificial Life IV*, pages 190–197, Cambridge, MA, jul 1994. MIT Press/Bradford Books.
43. Diego Perez, Miguel Nicolau, Michael O Neill, Anthony Brabazon, and Michael O'Neill. Evolving Behaviour Trees for the Mario AI Competition using Grammatical Evolution. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I Esparcia-Alcázar, Juan J Merelo, Ferrante Neri, Mike Preuss, Hendrik Richter, Julian Togelius, and Georgios N Yannakakis, editors, *Applications of evolutionary computation*, pages 123–132, Torino, apr 2011. Springer Berlin Heidelberg.
44. Pavel Petrovi. Evolving Behavior Coordination for Mobile Robots using Distributed Finite-State Automata. In Hitoshi Iba, editor, *Frontiers in Evolutionary Robotics*, chapter 23, pages 413–438. I-Tech Education and Publishing, Vienna, 2008.
45. Ágnes Pintér-Bartha, Anita Sobe, and Wilfried Elmenreich. Towards the Light: Comparing Evolved Neural Network Controllers and Finite State Machine Controllers. In *10th International Workshop on Intellegent Solutions in Embedded Systems*, pages 83–87, Klagenfurt, jul 2012. IEEE.
46. Sjoerd Tijmons, Guido de Croon, Bart Remes, Christophe de Wagter, R Ruijsink, Erik-Jan van Kampen, and Q P Chu. Stereo Vision based Obstacle Avoidance on Flapping Wing MAV. In *Euro Guidance, Navigation an Control Conference, EGNC 2013*, 2013.
47. Antti Valmari. The State Explosion Problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, pages 429–528. Springer Berlin Heidelberg, 1998.
48. Juan Cristóbal Zagal and Javier Ruiz-del Solar. Combining Simulation and Reality in Evolutionary Robotics. *Journal of Intelligent and Robotic Systems*, 50(1):19–39, mar 2007.
49. Jean-Christophe Zufferey and Dario Floreano. Fly-inspired visual steering of an ultralight indoor aircraft. *Transactions on Robotics*, 22(1):137–146, feb 2006.