Hybrid robot controller synthesis with GP and UETPN

Attila O. Kilyen
Department of Automation
Technical University of Cluj-Napoca
Romania
Email: kilyen.attila.ors@gmail.com

Tiberiu S. Letia
Department of Automation
Technical University of Cluj-Napoca
Romania
Email: Tiberiu.Letia@aut.utcluj.ro

***Abstract*—Controller synthesis for robotic agents has been one of the leading topics of Genetic Programming (GP) for decades. In this paper, a novel approach is presented to synthesise hybrid controllers. It uses Koza style genetic programming (GP) to generate Unified Enhanced Timed Petri Net (UETPN) models. UETPN models combine capabilities of timed Petri-nets, fuzzy logic systems and simple arithmetic operators. They can handle both event-like and continuous inputs (and outputs). They can change their inner state and execution flow based on the existence of a particular input event, or a value provided by a continuous input channel. In order to generate UETPN models (with GP), an intermediate language was designed, called UETPN Lisp. Dynamic and static editing and custom tailored crossover operators improve the proposed evolutive system. A three-wheel robot is modelled with a dynamic system. In order to exemplify the potential of the presented framework, a solution to solve the problem of corridor navigation and line following is proposed.**

***Index Terms*—hybrid control, Petri nets, genetic programming**

I. Introduction

Providing controllers automatically for robotic agents is one of the oldest problems addressed by genetic programming and machine learning in general. Despite the vast number of related works, most of them approach the problem either as discrete event systems only ([1],[2]) or in a continuous-time based manner ([3]). These approaches do not take into consideration that even simple mobile agents, like a sumo robot, can have both types of inputs and outputs. This paper introduces a framework which unites methods from discrete event systems and discrete-time control systems, in a way that the paradigms of genetic programming are applicable.

Unified Enhanced Timed Petri Net (UETPN) models are the target platform of the framework presented here. They were proven to a be an effective and comprehensive way to model reactive application [4], and to describe classical control techniques such as an on/off controller and PID controller. Fuzzy rules can be defined as well, to express complex behaviour.

An intermediate language, called UETPN Lisp, is defined in order to generate UETPN models with GP. They are a successor of a previously used framework, called ETPNL, which was able [5] to synthesise Delay Time Petri nets and to control discrete event systems.

978-1-5386-2205-6/18/$31.00 ©2018 IEEE

In the presented experiment, the movement of a robot agent equipped with line sensors or distance sensors is simulated. The submitted work addresses two different tasks: the corridor navigator, which is a harder version of the obstacle avoidance, and the line follower. Previous works approach the corridor navigator problem more often ([1],[3]) than the line follower. Nevertheless, the line follower implies the difficulty of managing states of the program and controlling the motors based on them. The model used here is more detailed than the ones used in [1] and [3]. Thus the resulted controller can be directly applied to the robot, without the need for an additional controller.

II. Evolving robot controllers

Koza uses the problem of ”Obstacle-Avoiding Robot” in [6] to exemplify the efficiency of automatically defined functions. However, this is more similar to the artificial ant than a real-world robot control. A grid-based representation is used in [2] too, where the task of the robot is not only to avoid obstacles but also to follow the walls. A similar function and terminal set are used in [7], where the result obtained was applied to a real-world robot.

The article [1] describes a GP based setup to generate a controller for a simple robot with three infrared sensors. The task of the robot is to navigate in a corridor system without colliding with the obstacles. The authors use a high-level function set. The reduced search space yields robust results in a relatively small number of generations and a minor population size. However, using high-level functions excludes a lot of possibly advantageous solutions.

In contrast to [1], the authors of [3] use a more meticulous approach to control the speed of the motors. They use Cartesian Genetic Program with the objective of developing a controller for obstacle avoiding robot. The solution candidate programs always react to the current input only, they do not preserve inner state. A separate feedback controller is needed to control the actual speed of the motors.

Nordin in [8] presents a different approach. Online learning is used in a way that the movement and behaviour of the robot are not simulated but executed by a miniature robot.

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.

TABLE I: **Example of inference rules (i. e. MT) with and without bending the graphic surface (only the indices of the rules are marked)**

x_1/x_2	X_{-2}	X_{-1}	X_0	X_1	X_2	ϕ
X_{-2}	$2,2$	$2,2$	$2,2$	$2,1$	$2,0$	ϕ,ϕ
X_{-1}	$2,2$	$2,2$	$2,1$	$2,0$	$2,-1$	ϕ,ϕ
X_0	$2,2$	$2,1$	$2,0$	$2,-1$	$2,-2$	ϕ,ϕ
X_1	$2,1$	$2,0$	$2,-1$	$2,-2$	$2,-2$	ϕ,ϕ
X_2	$2,0$	$2,-1$	$2,-2$	$2,-2$	$2,-2$	ϕ,ϕ
ϕ	ϕ,ϕ	ϕ,ϕ	ϕ,ϕ	ϕ,ϕ	ϕ,ϕ	ϕ,ϕ

III. Unified Enhanced Timed Petri Net

The UETPN models use tokens to describe information, similarly to classic Petri net. The proposed development approach based on UETPN models compounds the features of other Petri net models. They are capable of handling continuous (real number) variables and fuzzy logic variables and to perform simple arithmetical and logical operations with them. UETPNs are also capable of controlling (split, join, select or block) the execution of the model depending on some internal calculus or external (input) variables. They are capable of making decisions based on the existence (or lack) of tokens in the input place of the transition. They can also handle and signal events that associate different kinds of variables (from/to outside the model). Therefore asynchronous and synchronous concurrent execution can be described.

Every place has a scale (s_k). The token t_k assigned to a place is always in $([-s_k, s_k] \cup \phi)$, where ϕ means *no information*. Classical Petri nets represent *no information* by leaving a place empty, however, from the perspective of UETPN a place always has marking.

The transitions of UETPN models incorporate a mapping table and optionally, an arithmetic operator. The mapping table is an organised collection of fuzzy rules, which apply to the input token(s), and determine the output token(s). Some transitions have delays.

UETPN allows transitions with one or two input places only and similarly, one or two output places.

A mapping of a transition is a function between the current marking of the pre-places and post-places. It can be written as follows (for two input and two output places):

$$\begin{aligned} map_i\colon & ([-s_{i1}, s_{i1}] \cup \phi) \times ([-s_{i2}, s_{i2}] \cup \phi) \\ & \to ([-s_{o1}, s_{o1}] \cup \phi) \times ([-s_{o2}, s_{o2}] \cup \phi) \end{aligned} \tag{1}$$

Table I exemplifies an MT. If the current marking of a place is marked with x_p, then one cell represents the following fuzzy rule:

$$IF\ x_{i0} is X_0 \wedge x_{i1} is X_{-2}\ THEN\ x_{o1} is X_1 \wedge x_{o2} is X_2 \tag{2}$$

When a transition executes, the input tokens are fuzzified in the first step. The limits of the membership functions depend on the scale of the input place, as Figure 1 illustrates. Secondly, the rules of the mapping table are executed, the result is summarized, and defuzzified by the center-of- gravity method. The defuzzification intervals depend on the scale of the output place(s). The ranges are similar to the one in the case of the fuzzification, presented above.

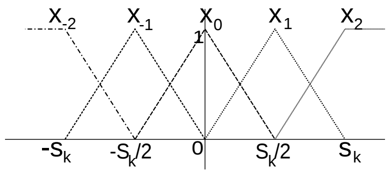


Fig. 1: Membership functions

When the transition has an associated arithmetic operator, the following equation is applied:

$$map_i(x_{i1}, x_{i2}) = (x_{i1} \circ x_{i2}) \star FL_{MT}(x_{i1}, x_{i2}) \tag{3}$$

where $\circ \in \{+, -, /, \times\}$, and $FL_{MT}(x_{i1}, x_{i2})$ stand for the result of mapping table deffuzified in the interval $[-1, 1]$. This means that the fuzzy rules are "bending" the result of the original operator. If all the conclusions are X_2, the outcome of the operator remains unchanged. The result is truncated based on the scale of the output place. Only the transitions with two pre-places can have operators.

The MT does not have to be complete, i.e. not every fuzzy rule has to exist. If the fuzzy rule does not exist, it is marked with ϕ. The MT also has ϕ columns and rows, which means that it is possible to define rules, even if one (or both) of the input tokens are ϕ.

The MT is used to decide whether a transition is executable. It is enabled to fire if there is at least one fuzzy rule with $non-\phi$ consequence (in the MT) which applies to the current input marking. In other words, a transition is executable if there is a $non-\phi$ cell which is activated after the input token(s) is defuzzified.

This definition of enabledness and the possibility to put ϕ in some cells of the MT facilitates the implementation of inhibitor arcs, reset arcs, and transitions, which are always enabled or always blocked.

The UETPN platform is designed with the intention of defining control components for hybrid systems, thus a clear way of communication with the outside world is indispensable. A UETPN control component has input channels which are represented as input places. Tokens can be set in these places, only by the exterior word (environment). The output channels are represented as output transitions. The output transitions do not have post-places, they send the tokens outside the current component. This achieves a manner of connecting multiple UETPN components.

A. Definition of UETPN

The definition of UETPN is:

$$\begin{aligned} UETPN = (&P, T, pre, post, D, S, EFS, Map, \\ & Inp, Out, \alpha, \beta, \delta, \mathbf{M}, M^0) \end{aligned}$$

where:

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.

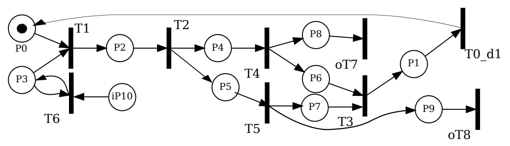


Fig. 2: Structure of the expression *(# (@ i:br:0 (% o:c:0 o:c:1)) d:1)*

- P is the place set, T is the transition set ($P \cup T = \emptyset$), while $pre \subset (P \times T)$ contains the arcs from places to transitions, $post \subset (T \times P)$ includes the arcs from transitions to places. D is the delay set. δ is a mapping $\delta : T \rightarrow D$, which associates delays to transitions. Their meaning corresponds to the ones from classic Petri nets.
- $Inp \subset P$ are the input places (channels), $Out \subset T$ are the output transitions (channels).
- $S = \{s_0, s_1, s_2 \dots, s_m\}$ is a set of real numbers representing the scale factor set or bound set, α is a mapping such that $\alpha\colon P \rightarrow S$, it assigns a scale to each place from the set P.
- **M** is the marking vector, while M^0 is the initial marking vector.
- EFS is the extended fuzzy set $\{X_{-2}, X_{-1}, X_0, X_1, X_2\} \cup \phi$,
- $Map = \{map_i | i = 0, 1, 2, \cdots, n\}$ is the set of mappings. A mapping consists of one particular fuzzy logic rule set, expressed in a form of mapping table (MT) and an optional arithmetic operator. β is a mapping $\beta\colon T \rightarrow Map$, it assigns a mapping from the set Map to each transition

B. Example of a UETPN model

Figure 2 illustrates the structure of a simple UETPN model. The given example has a simple functionality: it reads the input if the read input is positive, it sets a token to the zeroth output (oT7), otherwise, if it is negative, it puts a token to the first one (oT8). After that, it waits for one time-unit, and it starts again.

The *iP10* is the input place, where the outside environment may inject a token. *T6* has the role in resetting the content of *P3*, it replaces the existing token, regardless of its value, with the one from iP10. *T6* has an MT, which enables its execution if the current marking of *P3* is ϕ. It copies the token from *iP10* to *P3*.

T1 has the MT which requires the presence of a $non-\phi$ token both in *P6* and in *P3*, and it relocates the one in P3 to P2. The transition *T2* has the role in splitting the execution: if the token in *P2* is positive, then it yields the token to *P4*, meanwhile if it is negative, the token is set to *P5*. Table II shows the MT for *T2*. Following the execution of *T2*, *T4* or *T5* becomes enabled, as a consequence, depending on the sign of original input, either *oT7* or *oT8* output transition will fire. A token will remain in *P6* or *P7*. It is necessary for *T3* to have an MT which enables it to execute if one of the input places has $non-\phi$ token. $T0_d1$ has one time-unit delay when it finishes the execution, the entire cycle starts again.

TABLE II: **MR table for T2 from 2**

$P2$	X_{-2}	X_{-1}	X_0	X_1	X_2	ϕ
$P5$	X_0	X_0	ϕ	ϕ	ϕ	ϕ
$P4$	ϕ	ϕ	ϕ	X_0	X_0	ϕ

IV. UETPN Lisp

UETPN Lisp was conceived with the intention of using classic Koza style GP to synthesise UETPN models. Not only the structure, but the MTs and arithmetic operators have to be specified to execute these models. Meaning that the function and terminal set have to be defined in such a way that it fully specifies a UETPN model.

The operators always have two operands (which can be a terminal or another operator with its operands). A UETPN Lisp program has one operator at least. It has no type system. Thus all of the operators can accept any terminal or sub-expression for operand. As consequence, it can be represented as a binary tree.

A breadth-first traversal algorithm performs the transformation from UETPN Lisp expression to UETPN model. Every node and leaf has a start and end place. The represented structure has to be built up between these two. The first step of conversion is to add two places (usually *P0* and *P1*), and the root node is built up between these. *P0* has an initial token (with 0 value), which initiates execution of the model.

A. Operators of UETPN Lisp

The sequence operator is the basic building block of the language. It is denoted by "@", and it means that the second operand comes after the first one, separated by one place. When the first sub-expression terminates its execution, it sets a token to the place in the middle, and the second one can start its execution. In Figure 2, the transition *T1* and the structure that begins at *T2* ends at *T3* are in a sequential relationship.

The loop operator (denoted by *#*) and the selection operator (indicated by *?*) do not build any additional construct to the Petri net. The loop operator calls its first operand, whose start and end place corresponds to the bound places of the operator. Meanwhile, the second child is decoded to the same places, but in reverse order. In Figure 2, the transition *T0_d1* is the second operand of the loop operator, and the structure from *T1* to *T3* is the first.

The selection operator calls both of his operands with its start place as start place and with its end place as end place. The structure between *P2* and *P1* in Figure 3 exemplifies this operator. One of the branches is the transition *T6_d1*, while the structure containing *T7* and *T8* is the other. Depending on which transition is enabled, only one branch executes.

The following operators: concurrency (&), addition (+), multiplication (*) and positive-negative split (%) have the same structure, the differences between them are the associated MTs operators. They have the same building algorithm. Figure 2 contains such a layout, where the P2 is the start place of the original operator, and P1 is the end place. The transition

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.

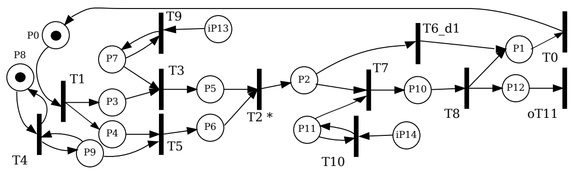


Fig. 3: Structure of expression *(# (@ (* i:br:0 c:2.0) (? (@ i:enp:1 o:c:0) d:1)) d:0)*

T1, T3 and the places P4, P5, P6, P7 are introduced when the positive-negative split operator is built up.

The central role of the concurrency operator is to bifurcate the execution. The two operands become two separate branches, and they run independently. However, at the end of the construct, the two bifurcated branches join (at the transition t_s).

The addition and multiplication operators work similarly to concurrency. They execute the two branches independently. The main difference is that the MT of transition contains only X_2 rules on the non-ϕ columns and rows, and they have addition or multiplication as associated arithmetical operators.

B. Operands of UETPN Lisp

Most of the terminals of the UETPN Lisp is connected as a single transition to the main branch of the model. However, some of them require auxiliary constructions.

The first terminal presented here is the input leaf denoted *i:ty:nr*, where *ty* stands for the input type, and *nr* specifies the input channel. This number is bound to the problem specification, for example, to a specific sensor. The *ty* stands for the input type, which defines the MT associated with the primary input transition.

In Figure 2, the structure *T6*, *P3*, *T1* represents the input leaf *i:br:0*. *T6* is the primary transition, its MT is defined by the type of the input leaf, blocking reader (abbreviated as "*br*") in this particular case. The place *P3* acts as a buffer for this input leaf. Similar structures can be observed in Figure 3 at the transitions *T3* and *T7*. These are associated with input leaves.

The blocking reader copies the value of the input token to the primary execution branch, if it is not ϕ, otherwise it blocks the main execution flow. The non-blocking reader ("*nr*") acts similarly in the case of non-ϕ tokens. If the token is ϕ, then it lets the token in the primary branch through without modification.

If the input is event-like, the input types enable if not ϕ ("*enp*") and enable if ϕ ("*eip*") are more suitable. These do not modify the token from the main branch. The *enp* is blocked until the input place has a token, respectively, the *eip* remains blocked until there is a no token in the input place.

Other types are: shift up ("*su*"), shift down ("*sd*"), enable if zero ("*eiz*"), enable if not zero (*enz*). As it was mentioned earlier, these types are predefined MTs, as a consequence, it is accessible to defined new ones if it is necessary.

The second terminal discussed here is the output leaf (*o:ty:nr*), which is the counterpart of the input leaf for outputs. Its role is it yields an output to a specified output channel. The output channel is specified by *nr*, similarly to the input leaf, *ty* stands for type. The only typed used currently is the copy type ("*c*"), with the possibility of defining others if necessary. The copy type yields the token from the main branch to the output without modifying it.

Examples of output leaves can be found in Figure 2, where the structure for *T5 -P9-oT8* stands for the *o:c:1*, while the structure *T4 -P8-oT7* for *o:c:0*.

Similarly to the input leaf, the auxiliary constructions are needed to use independently the same input or output channels multiple times.

One of the simplest leaves is the delay leaf (*d:nr*, where "nr" specifies the amount of the delay). This terminal inserts a transition with the specified amount of delay. Figure 2 contains an example, namely the transition *T0_d1* is built as a result of the leaf *d:1*. The default MT is assigned to the transitions constructed by delay leaves.

At this point every element of the expression *(# (@ i:br:0 (% o:c:0 o:c:1)) d:1)* is known, whose structure is displayed in Figure 2, and its behaviour is detailed in section III-B. One can foresee the response of the model based on the expression, without having to build the structure up.

Other simple leaves are the negation leaf ("*n*") and the blocking leaf ("*b*"). These two, similarly to the delay leaf, insert a single transition only. Nevertheless the associated MTs are different. In the case of the blocking leaf, all of the cells in the MT contain ϕ, which means that the inserted transition will never be executable.

A table, which changes the sign of the token from the main branch, is associated with the transition of the negation leaf. This type of leaf helps to achieve subtraction. There is also an inversion leaf ("*v*"), which returns one divided by the original token, but it has to be defined a more complicated way.

Constant leaf has the role of providing a mathematical constant. "*c:val*" abbreviates it, where *val* is the a real number, the value of the constant. The structure P8-T4-P9-T5 corresponds to a constant leaf in Figure 3. The constant leaf specifies the value of the initial token in P8. The execution of T4 is enabled if P9 has ϕ. Overall, it copies the value of the original token in T8 to P6 without consuming it.

At this point, every element of the expression *(# (@ (* i:br:0 c:2.0) (? (@ i:enp:1 o:c:0) d:1)) d:0)* is known, whose structure is presented in Figure 3. The behaviour of the model is evident from the expression itself: The value read from the zeroth input is multiplied by two. This value is set to the zeroth output if there is any non-ϕ token at the first input channel. Otherwise, the execution of the net blocks for one time-unit. When one of the possibilities is finished, the whole process starts over.

Memory leaf delays the value of the token without blocking the execution flow of the net. It is abbreviated as "*m:nr*", where *nr* is the number of time units by which the value of the token is delayed. If the current value (at the t-th moment) of the starting place is $t_{st}[t]$, then the marking of the end place can be expressed as $t_{end}[k] = t_{st}[k - nr]$. This type of operation

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.

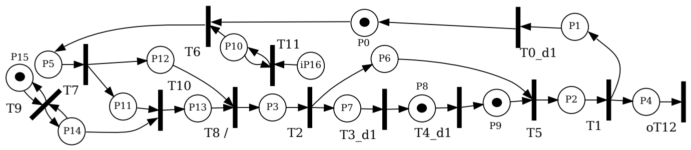


Fig. 4: Structure of expression *(# (@ (@ (@ i:br:0 v) m:2) o:c:0) d:1)*

is critical in the case of discrete control systems, for example, it is essential to define the PID controller.

An example of the structure of *m:2* leaf can be found between P3 and P2 in Figure 4. T3_d1 and T4_d1 have one time-unit delay and they memorise the past values of the input tokens. P8 and P9 have zero tokens as initial marking, which are the values returned by the leaf in the first two cycles. The T2-P6-T5 structure acts as a guard, preventing the leaf to leak tokens before P3 (the start place) has a non-ϕ marking.

In Figure 4, the structure of expression *(# (@ (@ (@ i:br:0 v) m:2) o:c:0) d:1)* is displayed. It behaves as follows: it reads and inverses the input, then the resulted value is shifted back with two time-units. It returns the result to the output and the whole execution starts again.

C. Problem specification

The scales of the places were not mentioned in the description of UETPN Lisp, because the language itself was not intended to specify the scales of each place separately. The default scale is part of the problem specification consisting of the number of the input, the scale of each input, the number of the output, the scale of each output and the default scale. The auxiliary construction for a particular input or output is scaled accordingly to the problem specification, while the rest of the places have the default scale.

The problem specification also has to provide the maximum amount delay, which can be associated with transitions. This limit applies to the memory leaf as well.

With this information one can not only convert the UETPN Lisp expression to UETPN model, but it is also possible to generate random trees. Besides the fitness function, the problem specification is the other necessary input information for the presented framework to produce a hybrid controller.

V. Genetic operators

The primary focus of this section is the exclusive operations on UETPN Lisp in the context of GP. The established operators are mentioned but not described here in detail.

The experiments use the ”ramped half and half” method to initialize the population. Classic subtree mutation is used. The usual configuration is the following: 19% of mutation, 19% of survival, 2% of elite, while the rest of the population is the result of one of the crossover operators (60%).

A. Static and dynamic editing

The initial experiments indicated that the presented language is prone to bloat. Huge expressions diminish the time performance of the framework so that even the use of small population needs hours to run.

The main difference between static and dynamic editing is, that the static editing is based only on the form of the expression itself, while the dynamic editing needs attributes recorded during the execution. Both of them have the purpose of reducing the size of the tree without changing its behaviour.

Static editing or simplification resembles the editing process described by Koza at [9]. It is realised based on the rules specific to UETPN Lisp. Some of these are related to mathematical operators. For example, calculations with constants are substituted with the result. Other operators have similar features, for example, in the case of the sequence operator, the expression *(@ d:1 d:1)* can be replaced with a single delay leaf with two delays.

Dynamic simplification (or editing) is done after the model is executed. The leaves which generate unused (not fired) transitions are replaced with blocking leaves.

The static and dynamic simplification supplement each other. However, dynamic simplification is feasible only after the fitness analysis. As a consequence, the simplification has three steps. Firstly, the static simplification is applied. Afterwards, the UETPN Lisp expression is converted to UETPN model, which is executed. As the second step, the dynamic simplification is employed. Finally, the static simplification is used again in order to clear away the unnecessary blocking leaves created by the dynamic simplification.

B. Crossover

The proposed framework employs three types of crossover. The standard subtree crossover ([9]) and the uniform crossover ([10]) are used conventionally, and they apply to any tree.

In contrast with ones mentioned above, the usage-based crossover is specific to UETPN Lisp. It works similarly to the standard crossover, the difference being the way how the subtrees are selected. During the execution of the UETPN model, it is recorded when a transition is fired, and it also counted, how many times it does (called the usage count). This information is used to select relevant subtrees.

The performance comparison of these crossover operators is a delicate issue because it is highly sensitive to the problem. In order to achieve a robust framework, the three crossover operator is used evenly proportional.

VI. Robot control

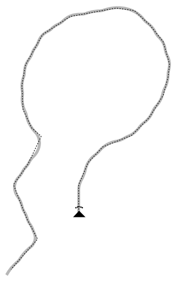
The same robot model was used to perform two different tasks.

A. Robot model

The robot motion model is detailed in [11]. The modelled robot consists of two wheels connected to DC motors and a third caster wheel. The model takes into account the forces produced by the two electric motors. It describes the dynamic behaviour of the chassis, based on its centre of mass. A resembling model is used and validated at [12].

The model presented at [11] has a linear part, described as state-space representation. It has two inputs: u_L and u_R, which

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.


(a) Line Follower

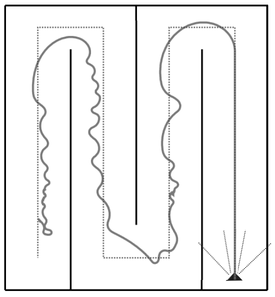

(b) Corridor navigator

Fig. 5: Behaviour of the found solutions

are the supply voltages for the two motors. The momentary speed (v_p) and speed of rotation (ω_p) of a selected point p of the robot are the output variables. The given state-space matrix is transformed into discrete form with $T_s = 0.025s$. As a result, the v_p and the ω_p are achieved for given input in every discrete time moment. Based on these, the position (x_p, y_p) and the current orientation (θ_p) of the robot can be easily calculated.

In order to facilitate the synthesis of the controllers, two intermediate variables are used as outputs: u_c and u_d. The control voltages are achieved by the equations: $u_L = u_c + u_d/2$ and $u_R = u_c - u_d/2$ where u_L and u_R are the input voltages for the left and right motor for the robot model.

B. Line follower

In order to simulate the line follower behaviour, firstly, the line itself is defined as a list of segments with a start- and an end-point. Each line sensor is defined as a coordinate relative to the base point of the robot. In the case of the presented experiment, five line sensors were used, which are symmetrically placed on the front of the robot. These were the inputs for the controller.

The raw fitness of the line follower is evaluated by breaking the original lines into smaller segments (5 cm). During the execution of the controller, the time unit when the robot touches a small segment is recorded. The raw fitness value is given by the number of the segments touched in a correct order.

The behaviour of a solution considered adequate (based on this criteria) is shown in Figure 5a. The presented framework can produce similar results with a population of 4000 individuals, 120 iteration around 7 hours of computational time.

C. Corridor navigation

The infrared distance sensors are defined by the relative coordinate and the relative angle to the current position and orientation of the robot. The calculated distance is transformed into output voltage based on the characteristics of a real sensor (Sharp GP2Y0A41SK0F). They can sense obstacles up to 40 cm. However, the output in the extremes of the range is very close to zero. The value of the output voltage is injected to the controller in each time unit. In the presented experiment, five sensors are used with the following relative angles: $0°, 10°, -10°, 45°, -45°$, hence the controller has five continuous inputs.

For fitness evaluation, a path made by segments was defined similarly to the line follower. In contrast to the line follower, the corridor navigator does not detect the track, which is only used for fitness evaluation. A solution candidate is penalised, in case the robot crashes into the walls of the corridor.

Figures 5b presents the behaviour of example solutions. These solutions were found in an experiment with a population of 4000 individuals, 150 iterations that took around 10 hours to run.

VII. Conclusion

The current work focuses mostly on the definition of the framework, which is released [1] under an open-source license. Besides the presented experiment, other more straightforward problems were addressed with success, such as the artificial ant problem or controlling a first-order dynamic system. Based on these experiments, can be concluded that the presented framework has the potential to generate controllers for hybrid systems successfully.

References

[1] B. Bonte and B. Wyns, "Automatically designing robot controllers and sensor morphology with genetic programming." in *AIAI*. Springer, 2010, pp. 86–93.

[2] C. Lazarus and H. Hu, "Using genetic programming to evolve robot behaviours," in *Proceedings of the 3rd British Conference on Autonomous Mobile Robotics and Autonomous Systems*, 2001.

[3] S. Harding and J. F. Miller, "Evolution of robot controller using cartesian genetic programming." in *EuroGP*, vol. 3447. Springer, 2005, pp. 62–73.

[4] T. S. Letia and A. O. Kilyen, "Unified enhanced time petri net models for development of the reactive applications," in *2017 3rd International Conference on Event-Based Control, Communication and Signal Processing (EBCCSP)*, May 2017, pp. 1–8.

[5] ——, "Evolutionary synthesis of hybrid controllers," in *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, Sept 2015, pp. 133–140.

[6] J. R. Koza, "Genetic programming ii: Automatic discovery of reusable subprograms," *Cambridge, MA, USA*, 1994.

[7] M. Ebner, "Evolution of a control architecture for a mobile robot," *Evolvable Systems: From Biology to Hardware*, pp. 303–310, 1998.

[8] P. Nordin and W. Banzhaf, "Real time control of a khepera robot using genetic programming," *Cybernetics and Control*, vol. 26, no. 3, pp. 533–561, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.6310

[9] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[10] R. Poli and W. B. Langdon, "On the search properties of different crossover operators in genetic programming," *Genetic Programming*, pp. 293–301, 1998.

[11] F. Dušek, D. Honc, and P. Rozsíval, "Mathematical model of differentially steered mobile robot," in *18th International Conference on Process Control, Tatranská Lomnica, Slovakia*, 2011.

[12] F. A. Salem, "Dynamic and kinematic models and control for differential drive mobile robots," *International Journal of Current Engineering and Technology*, vol. 3, no. 2, pp. 253–263, 2013.

[1]https://github.com/AttilaOrs/FuzzP

Authorized licensed use limited to: Wikipedia. Downloaded on June 22,2024 at 15:19:54 UTC from IEEE Xplore. Restrictions apply.