Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1035–1042
DOI: 10.15439/2018F49
ISSN 2300-5963 ACSIS, Vol. 15

Interactive development of cyber physical systems using UETPN model

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*—This paper presents a novel approach to synthesise hybrid controllers. A two-phase multi-objective evolutive algorithm was used to generate Unified Enhanced Timed Petri Net (UETPN) models. These 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). The first phase of the algorithm uses Koza style genetic programming combined with multi-objective methods such as NSGA-II and SPEA2 to obtain an initial model. The second phase improves the initial model with recombining the fuzzy rules with genetic algorithm GA. In order to generate UETPN models (with GP), an intermediate language was designed, called UETPN Lisp. Four example are presented to exemplify the potential of the proposed framework.**

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

I. Introduction

HYBRID controllers have a substantial practical importance because almost every real application from simple temperature control to complex robotic agents can have both event-like inputs and outputs, and continuous ones. Well-known examples of problems solved by Genetic Programming (GP), such as the artificial ant and obstacle avoiding robot (presented by Koza at [1]), are formulated in a way that only one domain is involved. In this paper, a two-phased evolutionary algorithm is presented, which is capable of synthesising controllers for discrete event systems, discrete time systems and hybrid systems as well.

Unified Enhanced Timed Petri Net (UETPN) models are used as the target platform for the proposed evolutionary framework. They are based on Delayed Time Fuzzy Petri nets [2]. For an effortless expression of control algorithms, they were completed with mathematical operators. Their ability to competently model reactive applications is shown in [3]. They are fit for handling continuous (real number) variables and fuzzy logic variables and for performing simple arithmetical and logical operations. They are capable of modifying the execution (split, join, select or block) depending on some external or internal value.

The proposed platform generates a complete UETPN model with GP, and in the second phase, it tries to improve it by recombining the fuzzy rules with genetic algorithm (GA).

The overall proposed framework needs a fitness evaluator for the given problem as input. This fitness evaluator consists of a fitness function and a light-weigth simulator. The ouput is an UETPN model which can be employed to control the specified system.

In order to widen its applicability, the presented framework supports the usage of Pareto front-based multiobjective methods such as NSGA-II [4] and SPEA2 [5]. These methods can also help to reduce the bloat, an issue which typically affects GP [6]. The presented experiments highlight the general applicability of the framework. The proposed method is applied to four different problems, for which the fitness function has been changed.

II. Evolving Petri nets

Fuzzy Petri nets are applied in various fields: path-tracking control problems; adaptive task assignment; fault estimation, detection, and diagnosis for power systems; urban and rail traffic control and many more [2]. Despite the vast range of applications, surprisingly few attempts were made to generate them automatically. Wong in [7] presents a framework called LOGENPRO used to extract knowledge from databases. LOGENPRO uses GP applied to logical grammars, based on place-manipulation and transitions-manipulation operators (such as sequential or parallel division). These operators modify the predefined fuzzy Petri net (FPN). However, the overall process restricts the structure of the FPN. The overall result lacks any internal state, timing and inner loops. These restrictions limit the applicability of the proposed framework in control applications.

Nobile in [8] introduces a new type of PN called Resizable Petri Net. This have divided the places and transitions into two groups: hidden places (or transitions) and normal ones. In the proposed framework only the number of the hidden nodes varies, the resulted PNs resemble binary trees.

An entirely different approach is presented in [9], which evolves some parameters. Based on these parameters and a template the final PN is assembled. This approach allows the use of the traditional real-coded genetic algorithm. Nevertheless, it can be applied only in case the structure of the PN is known upfront. Another approach presented by [10] addresses a biological problem modelled by PNs. In this case,

TABLE I: Mapping table (i. e. MT) associated with *t4* of the first example (only the indices of the rules are marked)

$P9/P8$	X_{-2}	X_{-1}	X_0	X_1	X_2	ϕ
X_{-2}	ϕ	ϕ	ϕ	ϕ	ϕ	-2
X_{-1}	ϕ	ϕ	ϕ	ϕ	ϕ	-1
X_0	ϕ	ϕ	ϕ	ϕ	ϕ	0
X_1	ϕ	ϕ	ϕ	ϕ	ϕ	1
X_2	ϕ	ϕ	ϕ	ϕ	ϕ	2
ϕ	-2	-1	0	1	2	ϕ

the authors establish the number of the places based on the problem specification, traditional GA evolves only the arcs. The presented problem is solved, still, their approach is far too restrictive to be applied in a broader domain.

A more general idea is presented in [11], using traditional GA. The gene, a transition-place pair, is decoded as an arc linking them. They use their framework to synthesise PNs starting from a structural and behavioural specification. Following the evolutionary process, additional steps are needed to improve the result and to reduce its size. This approach has potential, yet we believe that GP is abler to discover and re-utilise building block in the case of a PN-like structure.

III. Unified Enhanced Timed Petri Net

The UETPN models ([12]) incorporate transitions that have associated mapping tables (MT) and optional arithmetic operators. The MT is an organised form of the fuzzy rules which determine the bahaviour of the transitions. The input token(s) can activate the rules and if there is no operator, these rules define the output of the transition and they decide whether a transition is executable. UETPN supports transitions with one or two input places and one or two output places.

The mapping of a transition can be defined as a function between the current marking of the pre-places and post-places.

MT can have different shapes based on the numbers of inputs and outputs. An MT with two inputs and one output is exemplified in Table I, while another with one input and two output in Table II. If the current marking of a place is marked with x_p, then one cell represents the following fuzzy rule:

$$IF\ x_{i0} is X_0 \wedge x_{i1} is X_{-2}\ THEN\ x_{o1} is X_1 \wedge x_{o2} is X_2 \quad (1)$$

Each place has an associated scale (s_k). The token t_k set in a place is always in $([-s_k, s_k] \cup \phi)$, where ϕ means *no information*. Petri nets express *no information* leaving the place empty, nevertheless, in the case of the UETPN a place always has a marking.

When a transition executes, the input tokens are fuzzified in the first step. The limits of the membership functions are defined based on the scale of the input place. Simple triangular membership functions are used. Secondly, the fuzzy rules in the MT are executed, the result is collected and defuzzified by the center-of-gravity method. The scale of the output place(s) determines the defuzzification intervals.

If an arithmetic operator is assigned to the transition, the following equation is applied:

$$map_i(x_{i1}, x_{i2}) = (x_{i1} \circ x_{i2}) \star FL_{MT}(x_{i1}, x_{i2}) \quad (2)$$

TABLE II: Mapping table (i. e. MT) associated with *t3* of the first example (only the indices of the rules are marked)

X_{-2}	X_{-1}	X_0	X_1	X_2	ϕ
$0, \phi$	$0, \phi$	ϕ, ϕ	$\phi, 0$	$\phi, 0$	ϕ, ϕ

where $\circ \in \{+, -, /, \times\}$, and $FL_{MT}(x_{i1}, x_{i2})$ stands for the result of the mapping table deffuzified in the interval $[-1, 1]$. The fuzzy rules can alter the original result, but, if all the conclusions are X_2, the result of the operator is unchanged. The obtained value is truncated based on the scale of the output place. Only the transitions with two input places can have operators.

The existence of every fuzzy rule is not compulsory for MT construction. ϕ signals the missing rules. The MT also has ϕ columns and rows, which supports the definition of rules even if one (or both) of the input tokens are ϕ.

Another role of the MT is to decide whether a transition is executable (referred to as enabledness). A transition is allowed to fire if there is at least one fuzzy rule with $non-\phi$ consequence (in the MT) which applies to the current input marking.

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

A precise description of communication with the outside world is pivotal for the UETPN to model hybrid controllers. UETPN models represent input channels as input places. The tokens set in these places can emerge from the exterior world (environment) only. The output channels are represented as output transitions. The output transitions do not have post-places, they send the tokens outside the current component. Multiple UETPN components can be connected in the previously described way.

A. Definition of UETPN

[3] contains not only the complete definition of the UTPN models, but some significant examples and applications also. In this section, only a brief introduction is given. The examples do not resolve a real-life problem, however they illustrate some of the capabilities of UETPN models (and they exemplify the UETPN-Lisp introduced in the next section as well).

The definition of UETPN is:

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

where:

- 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 $\delta : T \to D$, 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).

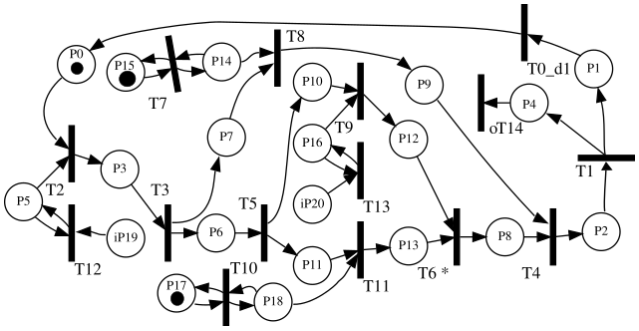

Fig. 1: Structure of the expression *(# (@ (@ i:br:0 (% (* i:br:1 c:2.0) c:0)) o:c:0) d:1)*

[t]

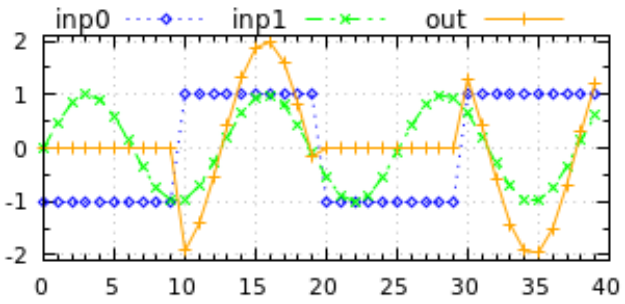

Fig. 2: Behavior of the first example

- $S = \{s_0, s_1, s_2 \ldots, s_m\}$ is a set of real numbers representing the scale factor set or bound set, α is a mapping such that $\alpha\colon P \to S$, 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 $\beta\colon T \to Map$, it assigns a mapping from the set Map to each transition

B. Example of a UETPN model

The first example illustrates how UETPN models can alter their execution based on the value of an (input) token. It demonstrates how mathematical operations can be implemented too.

Figure 1 represents the structure of the first example of a UETPN model. It has two input places and one output place. If the zeroth input channel has a positive non-ϕ token, the output equals two times the first input. Else, the output returns a constant zero token. The transition t3 has a key role in this behaviour. Its MT table returns a token to P6 if the value in P3 is positive, and to P7 if it is negative. Table II specifies the mentioned MT table. The constructions P17-t10-P18 and P15-t7-P14 are responsible for providing the constant values, P15 and P17 contain an initial token with the value of 2.0 and 0, respectively. T6 multiplies the input read from iP20 (via P12) with the constant from P17 (via P13). T4 has to have an MT table which enables if there is a non-ϕ token in P9 or P8, because only one main branch is executed simultaneously. Table I presents this MT. The transition oT14 fires the result to the outside world. Note, that t0 has one delay before the whole cycle starts over.

Figure 2 displays the behaviour of the described example. It can be observed that when the zeroth input is -1 (between the 0th and 10th time tick and between 20th and 30th time-tick), the output is zero. When the zeroth input is 1, the output doubles the sine wave of the first input.

The second example also alters its execution, however, in a completely different way. In contrast with the first example, the execution flow depends on whether a transition is executable. Note that the definition of the UETPN models does not prioritize the transitions, however, one can be sure that the executor tries to fire the transitions with the lowest delay at first. Another restriction is that in the case of conflicting transitions, the executor behaves in a deterministic way.

Figure 3 displays the structure of the second example. This example has two inputs and one output. All the places have 1.0 as scale. It contains an inner loop which executes until the outside environment inserts a token to iP11, which in turn is transmitted to P5. When a token appears at P4, t1 becomes executable, and the model stops its execution. The inner loop returns a token to the output oT9 if in the input place iP10 has a token which activates the X_0 rule of t5. Table III specifies the MT of t5. If t5 is not fireable, T4 starts firing. After T4 finished its firing (it has one tick delay) the whole cycle begins over via T3.

Figure 4 shows the behaviour of the second example. The output is activated when the zeroth input is in the interval $[-0.5, 0.5]$ which corresponds to the activation of rule X_0 after fuzzification. It is also observable that after an event appears in the first input (at the 30th time-tick), there is no more output regardless of the inputs.

Note, that the transitions t12 and t13 from the first example, t8 and t7 from the second example have the role in copying the new input token regardless of the existence or value of the previous inputs (stored in the buffer places after these transitions). These structures are not essential from the functional

TABLE III: Mapping table (i. e. MT) associated with *t5* of the second example (only the indices of the rules are marked)

$P2/P8$	X_{-2}	X_{-1}	X_0	X_1	X_2	ϕ
X_{-2}	ϕ	ϕ	-2	ϕ	ϕ	ϕ
X_{-1}	ϕ	ϕ	-1	ϕ	ϕ	ϕ
X_0	ϕ	ϕ	0	ϕ	ϕ	ϕ
X_1	ϕ	ϕ	1	ϕ	ϕ	ϕ
X_2	ϕ	ϕ	2	ϕ	ϕ	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

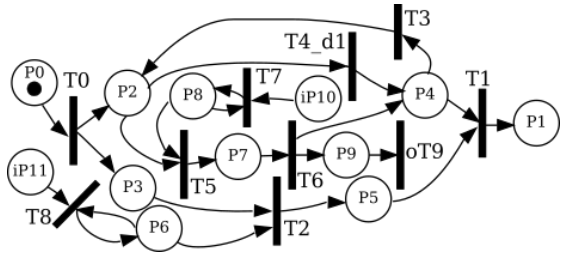

Fig. 3: Structure of *(& (#(?(@ i:eiz:0 o:c:0) d:1) d:0)i:enp:1)*

point of view, but since these models are direct correspondents to some UETPN Lisp expressions, it would be imprecise to omit them.

IV. Model construction

The semi-automatic way of fabricating a UETPN model has two phases. In the first one, an initial model is generated with GP and UETPN Lisp. It constructs a complete UETPN model, which is executable and evaluable with a fitness function. However, the available MT tables are predefined, which limits the possible behaviours. The result of the first phase is not only the UETPN model itself but also a meta-data about the context of transitions. Based on this metadata the MTs of some transitions can be re-trained with GA. This preselection of the transitions is necessary because in most cases the models yielded by the first phase have so many transitions that it is infeasible to optimize all of the MTs with GA.

There can be several reasons to employ the second phase. First of all, if the fitness value produced by the result of GP is not satisfactory, there is a chance that the second phase will improve it. Secondly, the fitness itself can be adjusted, taking into account other evaluation criteria. Thirdly, the parameters of the problem can also be modified. In this case, the re-training of the UETPN model achieved in the first phase may lead to significant improvement.

Although the entirely automatic synthesis is possible, it is highly recommended to analyse and evaluate the solutions manually after the first phase. It is recommended to ensure that the found solution has the desired behaviour, because the GP often can find a workaround to achieve a high fitness value without satisfying the real fitness criteria. The length of the simulation for fitness evaluations is always a compromise, because the longer the simulation is, the more time it takes to run the algorithm. It is advised to execute the first phase several times and to choose a suitable solution before performing the second phase. Also, the types of transitions whose MTs are optimised have to be selected manually.

Finally, it has to be mentioned that in the case of problems which do not benefit from the effect of fuzzy rules, the second phase has no benefit. In these cases, the problem has to be solved in the first phase.

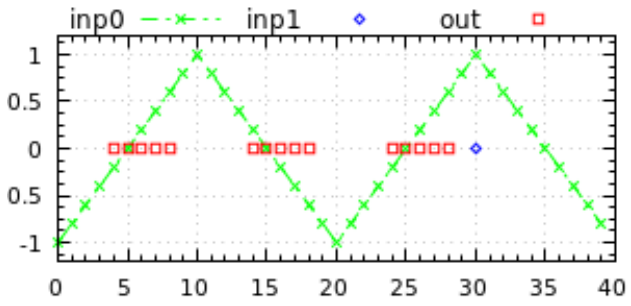

Fig. 4: Behavior of the second example

A. First phase:GP with UETPN Lisp

In the first phase, classical tree-based Koza style GP is used. The overall algorithm and the genetic-operators are not specified here, they are applied as in the literature ([13] and [1]). The focus of this section is on the UEPN Lisp, which is a small language used by GP framework. It presents the operators and the operands and exemplifies he conversation from UETPN Lisp to UETPN model.

UTEPN Lisp is transformed to UEPN model by breadth-first traversal. Every sub-expression is built up between two places. The algorithm starts by adding the two principal places. The root node is built between these. The first place has an initial token which begins the execution of the model. Both in case of the first example (Figure 1) and the second example (Figure 3), this starting token is placed in P0.

All of the operators of the UETPN Lisp have two operands. The sequence operator denoted by @ is the simplest one, it indicates that the two operands come after each other separated by a place. In the second example, the place P7 separates the transition T5 and T6, a structure which is the result of decoding a sequential sub-expression "*(@ i:eiz:0 c:c0)*". The selection operator (marked with *?*) builds up both of its children between its original starting and ending place. It can be noticed that either the transition t4 or the structure t5-P7-t6 executes from the second example. This is the result of a sub-expression of *(? (@ i:eiz:0 c:c0) d1)*. The loop operator (#) produces a similar structure, however, the direction of the second child is reversed. In the second example, T3 is the second child of a loop operator. T0 is the second child of a loop node in the first example.

The structures produced by the concurrency operator (&), positive-negative split operator (%), sum operator (+) and multiplication operator (*) are the same, the only difference are the MT tables and the mathematical operators associated with the transition. In the first example, the structure from P3 to P2 is the result of the sub-expression *(% (* i:br:1 c:2.0) c:0.))*, and the transition T3 and T4 are built as the part of the positive-negative split operator. The T3 has an MT which yields a token into one of its outputs only, based on the sign of the input. In the structure mentioned above, the fragment staring from P6 to P8 is built as the result of the sub-expression *(* i:br:1 c:2.0)*. The transitions T5 and T6 were added as a result of the multiplication operator.

In case of the second example, the root of the expression is a concurrency node. The transitions t0 and t1 are added as

part of this operator. As the name suggests, in this case, both branches are executed independently, and t1 becomes fireable if both finish their execution. If the sum or multiplication operator is used, T1 has the corresponding associated operator.

The most crucial operands are the input and the output operands. They connect the main flow of the model to the input place or output transition. They also have a type which specifies the MT tables used in the connection transition. For example, the input node *i:eiz:0* means that the zeroth input is read with a transition which is enabled if the input token activates the X_0 rule. In this case, the MT mentioned above belongs to t8 (second example), and iP10 is the zeroth input.

The P8 is a buffer place where each input node has a copy of the original input token, and t7 is an auxiliary transition, responsible for placing the new token to the buffer place. This role is essential in the case of models which use the same input multiple times. Other types of inputs are the blocking reader ("*br*"),non-blocking reader ("*nbr*"), enable if non-ϕ ("*enp*"), enable if phi ("*eip*"), enable if zero ("*eiz*"), enable if not-zero ("*enz*"). The type of the input is essential in case of the second example, where the zeroth input has " enable if zero" type. If the evolutive framework mutates it into "enable if not-zero" ("*enz*"), the behavior of the model is inverted. If changed to something else, the execution of the model would be dramatically different. Note that these input types help the framework to deal in a completely different way with event-like and continuous inputs . The ones with *"enable"* in their names block the token in the main flow without modifying its value. The *"reader"* ones copy the value of the input token into the main flow.

Similarly to input leaves, outputs also need auxiliary constructions. In the second example, P9 acts as a buffer place connecting T6 to the real output transitions oT9, which is the result of the leaf *o:c:0*. Outputs currently can be only *copy* type, however, defining new types is as easy as to define a new MT table. (This applies to the input types as well.)

Other operands are the delay nodes (*d:nr*), which insert transitions into the flow with a certain delay, the constant (*c:nr*) leafs that provides a mathematical constant, blocking leaf (*b*), whose role is to insert a transition which is never executable, negation leaf (*n*) which negates the sign of the tokens, inversion leaf (*v*), and memory leaf (*m:nr*) which delays the value of the token, but it does not block the execution of the model. Some of these (memory, inversion, constant) are implemented with complex structures, others are single transitions with unique MT tables and/or mathematical operators.

During the decoding, the created transitions are categorised as follows: input, output, split-starter, split-merger, auxiliary, others. These categories (one or more) can be selected for optimisation by the second phase. The early experiments showed that in the majority of the cases optimising the input, the output and the split-merger transitions yielded the same results as adding any other category to this group. What is more, widening the set of optimised transitions means larger search space for the second phase.

B. Second phase: GA for the fuzzy rules

Similarly to the previous section, the GA itself and the genetic operators are not presented here, they are used in the well-known way ([14]). Simple binary encoding is used. The user of the algorithm decides which type of transitions have to be optimised.

In order not to change the behaviour drastically, only the non-ϕ rules were marked to be re-trained by the presented GA, and they can trun to other non-phi rules. Three bits can represent the five possible rules. A three-bit unit was chosen as the gene, their sequence composes the chromosome for GA. Since three bits can represent eight values, three of them is not used. The crossover, the mutation and the creation of initial population were modified in order not to produce the unused combinations.

V. Experiments

A. Control of a first order system

A simple task was choosen to exemplify the working of the framework: control of a first order system. Firstly, a known structure is optimized. Secondly, the complete framework solves the problem.

The fitness functions used for these problems take into account the absolute error (e_a) and the steady state error(e_s) of the controlled system. All of the experiments use the same fitness function:

$$f(i) = 1/(1 + \alpha * e_a + \beta * e_s)$$

where α and β are constants set to 0.8 and 0.2.

Firstly, a proportional integral (PI) controller was manually defined in UETPN Lisp. The rules of the input, output and the split-merger transitions were optimised. After 50 runs, the average fitness was 19.46. One of the average results is presented on the upper part of the Figure 5.

Secondly, GP solves the same problem with the same fitness function. The after 50 runs, the average fitness of the results was 25.39. An average result is displayed in the middle part of in Figure 5. Not only that the second result has hogher fitness, but the difference is also conspicuous, the overshoot is better, and the overall error is smaller.

Thirdly, GA is applied to optimise the rules of the previous result. The average fitness in this case is 27.88, which means smaller increment, however, the differences (the lower part of the Figure 5) still can be noticed.

Additonally the result of the second step was re-trained with GA to control another modified first order discrete system. The solutions have superior performance than the orginal controller, however, they do not overperform a re-traned PI controller.

B. Artificial Ant Problem

The Artificial Ant is a classic GP problem used by Koza at [13] to illustrate GP. The previously presented UETPN Lisp has the disadvantage of being more general, in contrast to Koza's solution (and many other papers), which use specialised operators and operands. As it is anticipated, the

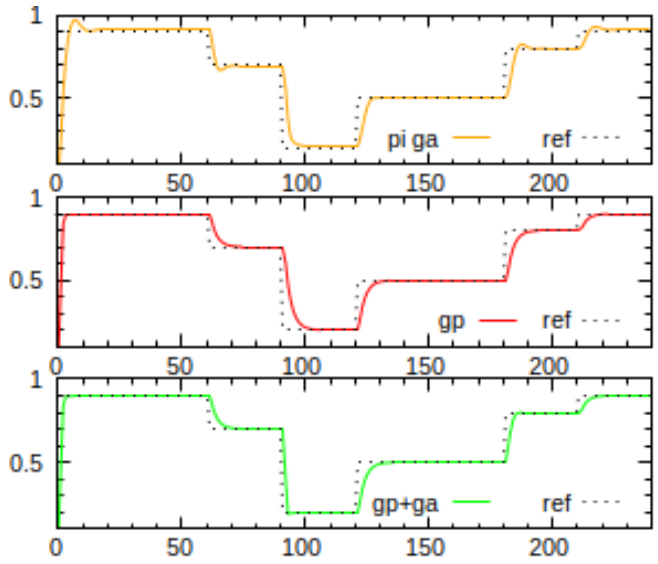

Fig. 5: Comparison of different results for the original system (PI optimized, GA only and GA+GP)

performance of the general framework does not match the specialised one.

An artificial ant is capable to turn left or right and to move forward. The single input is activated if food is ahead. The controller has three event-like outputs and one even-like input. The fitness function is the number of food units eaten. The ant is placed in the well-known Santa Fe trait.

The overall problem is known to be difficult to solve and it was used by several researchers to demonstrate features of their algorithm. A comprehenesive review of the problem can be found at [15]. The advantages of flexible genome show up in the case of this problem, because fixed length genome can represent only restricted number of program states.

Since this problem requires higher population number and iteration than the previous one, bloat presents a more serious complication. Bloat is the phenomenon when tree-sizes grow exponentially fast, wasting the computational time and filling up the overall genetic material of a population with useless fragments. The proposed framework implements traditional methods such as various form of Parsimony Pressure, which essentially means that the size of the individual is part of the fitness.

The problem with these methods are that it is hard to assess the size compared to the original fitness value. The demonstrated framework can also apply static and dynamic simplification over UETPN-Lisp. These methods replace the sub-expressions, which can be expressed simpler and delete the unused ones. Although they are effective on compacting the potential solutions, they cannot entirely solve the bloating phenomenon.

A robust method in the case of UEPN-Lips applied to the artificial ant problems is to use NSGA-II with two fitness functions. The first of them is the number of food eaten (f_f), the second one is the number of eaten multiplied by the size factor (f_s). This way the second objective favours candidates which are small and fit. This idea gives better results than the one presented [5] where the second objective evaluates only the size. The size factor (f_s) is calculated by:

$$f_s = (s_i - s_{prf})/(s_{max} - s_{prf})$$

where s_i is the size of the individual, s_{max} is the maximum allowed size, while s_{prf} is the preferred size. In these experiments $s_{max} = 500$ and $s_{prf} = 20$, outside these limits f_s is defined as 1 (if $s_i < s_{prf}$) or 0 (if $s_i > s_{max}$).

The original solution presented by Koza needs to evaluate $450 * 10^3$ individuals for an acceptable average solution ([15]), similar results can be obtained with $500 * 10^3$. The best algorithm proposed by [15] is equipped with problem-specific base language but also specialised crossover and mutation, and it has a far better success rate with $51 * 10^3$ evaluated individuals.

C. Room temperature control

This experiment is a hybrid application with two inputs: the reference temperature and the actual reading from the heat sensor in the room. There are two event-like outputs: one of them starts the heating, the other one stops it.

A discrete time system simulates the room temperature. It is based on the temperature differences:

- $\delta_{ht}[k] = t_{hw}[k] - t_r[k]$ is the difference between the temperature of the heating water (t_{hw}) and the room temperature (t_r).
- $\delta_o[k] = t_r[k] - t_o[k]$ is the difference between the outside temperature (t_o) and the room temperature.

The room temperature is simulated in the following way:

$$t_r[k+1] = t_r[k] + c_{ht} * \delta_{ht} - c_{wl} * \delta_o - c_{wi} * \delta_o$$

, where c_{ht} is the heating constant set to zero if the heating is turned off. c_{wl} is the wall constant, and c_{wi} is the window constant set to 0.0 zero of the window is closed. The opening and closing is the disturbance of the overall system. In this case, the temperature of the heating water is considered constant. Another important point is that the sensor reading is delayed compared to the simulated room temperature.

A elementary fitness function can be defined with the help of the sum of the error. However, this would lead to controllers which do not react to the input values but rather turn on/off the heating periodically. The problem can be solved by adding a lot of test scenarios with hectic temperature changes. This would lead to slower evaluation of a solution candidate, hence longer overall runtime. The approach used here evades the elongated run-time: the number of the minutes when the temperature is outside the interval of $t_{ref} \pm \delta$ is mesured, where δ was chosen to be 0.5. The behaviour of a solution found in the first phase (with GP) is presented in Figure 6. The controller turns the heater on only when it is necessary, and the heater stays on until a certain limit is reached. The desired behaviour is achieved, however the GA was not able to improve it.

In the second part of the experiment, the parameters of the room model were changed. The disturbance effect was higher

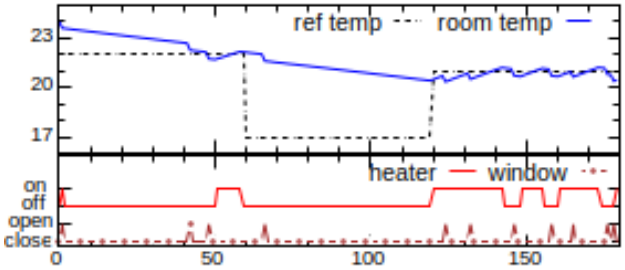

Fig. 6: The evolution of room temperature in case of GP

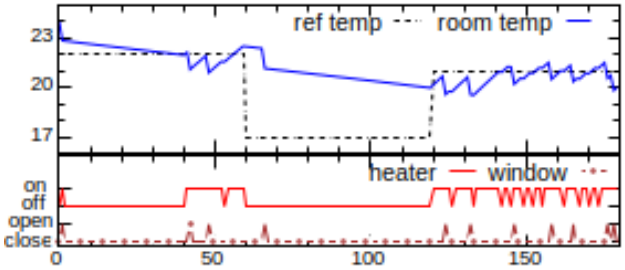

Fig. 7: The evolution of room temperature in case of the modified system

than in the previous experiment (presumably because the window is larger or the volume of the room is smaller compared to the window), while power of the heater was increased. In this case, the original controller performed poorer. The behaviour of a solution which has the same structure as the original one but it was re-trained with GA taking into consideration the new room model. The new behaviour of is shown in Figure 7. The presented solution was able to respond to the new challenge.

D. Room and Water Heater Temperature control

In this section, the desired controller has to control the temperature of the heating water, in contrast with the previous part where it was considered constant. This way the controllers have three outputs, first two of them have same functionality as in the previous section.The third has continuous behavior, aimed to control the water temperature in the heater tank. There are four inputs: the actual temperature of the room, the reference temperature of the room, the actual temperature of the water and the reference temperature of the water.

With the objective to simulate the temperature of the heating tank, the following intermediate variables are defined:

- $\delta_w[k] = t_{hw}[k] - t_pw[k]$ is the temperature difference between the heating water (t_{hw}) and the water from the pipe (t_{pw}).
- $\delta_{cmd}[k] = t_{max} - t_{hw}[k]$ is the difference between the maximum water temperature (t_{max}) and the current heating water.
- $\delta_t[k] = t_hw[k] - t_t[k]$ is the temperature difference between the heating water and the room.

The temperature of the heating water is given by the following discrete time equation:

$$t_{hw}[k+1] = t_{hw}[k] - c_h * \delta_w + c_{cmd} * u[k] * \delta_{cmd} - c_t * \delta_t$$

where c_h, c_{cmd}, c_t are constant, c_h is set to 0 if the heating is off. $u[k]$ is the output of the controller. The rest of the system is identical to the one mentioned before.

This system can be viewed as the combination of the first order system, and the setup presented in the previous section. Although it is self-explanatory that the framework can express the desired controller, it is hard to come up with the adequate fitness objective(s). The only known solution needs multi-objective optimization and far more evaluated individuals than any of the examples presented here.

The first objective is similar to the one in the previous section it counts the number of the minutes when the room temperature and the temperature of the water is off limit. The second fitness objective sums up the error and of the room temperature and the temperature of the water relative to the references. (The reference for the water temperature is constant 60). The third one is identical to the first one, except that it takes into account the size factor presented in the section related to the artificial ant. SPEA2 is known to handle better more than two objectives than NSGAII [5], hence it was chosen as the base algorithm for the experiment.

As it has turned out, the boiler is too weak compared to the room model. In practice, this means that when the heating is turned on in the room, the temperature of the water drops fast and constant temperature cannot be maintained for more than two or three minutes. In the real world, this would be a severe design problem, but the proposed framework overcame this flaw.

Figure 8 presents the behaviour of one solution found after GA is applied as well. This controller starts to heat up the water in the heating tank before the temperature of the room drops to a critical temperature. This way when the controller turns on the heating in the room, the water is already heated above the reference temperature, and the heating can remain on for a longer time. It is not the expected solution, but it is unarguably a creative one.

Figure 9 displays the operations of an another solution. This solution turns on the heating for a minute or two only in order to maintain the constant water temperature. This approach performs weaker in the first hour of the presented scenario than the previous one. However, in the rest of the time it has an adequate performance.

The first solution performs better from the perspective of the first fitness objective, while the second fitness objective favours the second solution. In this experiment, multiple fitness scenarios were needed in order to reduce the number of solutions which perform a cyclic behaviour, which fits the average cases, without reacting to the current situation.

The presented framework is capable of finding an acceptable solution in 10% of the cases. The population size was 3200 combined with 200 iteration. This took in average five hours, more than 50 experiments were performed.

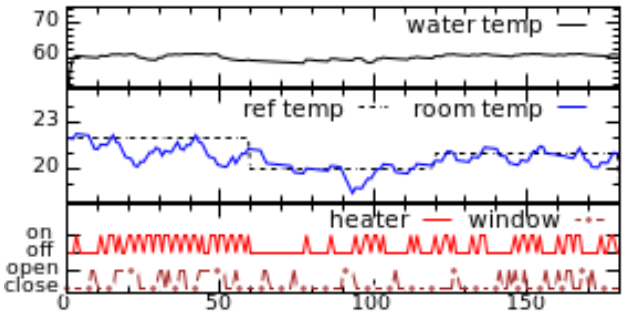

Fig. 9: The behavior of the second solution for Room and Water Heater Temperature control

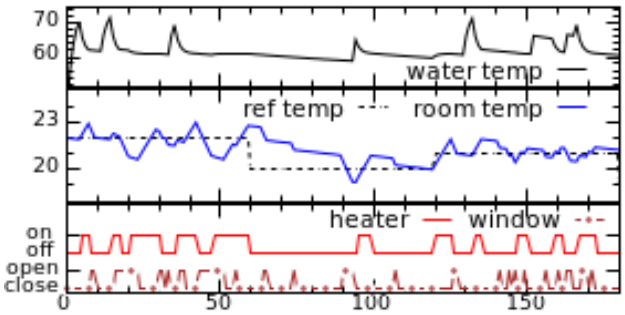

Fig. 8: The behavior of the first solution for Room and Water Heater Temperature control

VI. Conclusion

The current work focuses mostly on the capabilities of the framework and the effects of the two phases of the development. All of the presented experiments are reproducible with the code released [1] under an open-source license. Based on these experiments, it can be concluded that the given framework has the potential to generate controllers for hybrid systems. The presented experiments also highlight the fact that a complete base-language is not enough to tackle complex problems, the primary evolutive framework is at least that important.

The importance of multi-objective methods based on Pareto-fronts cannot be overstated. They have a strong focus on exploiting the known Pareto-front and on trying to improve in one way or another. This behavior is essential in case of controlling the bloat when shorter individuals are preferred, however, individuals with high fitness should not be deleted based on their size only. Yet, these algorithms have the disadvantage of being more disposed to early convergence to sub-optimal solutions. The explicit diversity control may be needed in the future.

Future development directions could focus on the compiling of the UETPN-model to machine code or Java Virtual Machine byte-code with the objective to make the evolution of a proposed solution faster. Another important direction is to conceive a method to deploy UEPN-models directly into micro-controllers with the aim to apply the presented result in real-life.

References

[1] J. R. Koza, "Genetic programming ii: Automatic discovery of reusable subprograms," *Cambridge, MA, USA*, 1994.
[2] K.-Q. Zhou and A. M. Zain, "Fuzzy petri nets and industrial applications: a review," *Artificial Intelligence Review*, vol. 45, no. 4, pp. 405–446, 2016.
[3] 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.
[4] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

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

[5] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.
[6] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler, "Multiobjective genetic programming: Reducing bloat using spea2," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 1. IEEE, 2001, pp. 536–543.
[7] M. L. Wong, "A flexible knowledge discovery system using genetic programming and logic grammars," *Decision Support Systems*, vol. 31, no. 4, pp. 405 – 428, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923601000926
[8] M. S. Nobile, D. Besozzi, P. Cazzaniga, G. Mauri *et al.*, "The foundation of evolutionary petri nets." in *BioPPN@ Petri Nets*. Citeseer, 2013, pp. 60–74.
[9] A. Gudelj, D. Kezić, and S. Vidačić, "Marine traffic optimization using petri net and genetic algorithm," *PROMET-Traffic&Transportation*, vol. 24, no. 6, pp. 469–478, 2012.
[10] J. Nummela and B. A. Julstrom, "Evolving petri nets to represent metabolic pathways," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 2133–2139. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068361
[11] T. Bourdeaud'huy and P. Yim, "Petri net controller synthesis using genetic search," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, Oct 2002, pp. 528–533.
[12] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
[13] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.
[14] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.
[15] D. Wilson and D. Kaur, "How santa fe ants evolve," *arXiv preprint arXiv:1312.1858*, 2013.