Fine tuning run parameter values in rule-based machine learning[⋆]

Sotiris Moschoyiannis[1] and Vasily Shcherbinin[2]

[1] Department of Computer Science, University of Surrey, Guildford, Surrey, GU2 7XH (`s.moschoyiannis@surrey.ac.uk`)
[2] Department of Computer Science, University of Surrey, Guildford, Surrey, GU2 7XH (`v.shcherbinin@surrey.ac.uk`)

Abstract. *Rule-based machine learning* focuses on learning or evolving a set of rules that represents the knowledge captured by the system. Due to its inherent complexity, a certain amount of fine tuning is required before it can be applied to a particular problem. However, there is limited information available to researchers when it comes to setting the corresponding run parameter values. In this paper, we investigate the run parameters of *Learning Classifier Systems* (LCSs) as applied to single-step problems. In particular, we study two LCS variants, XCS for reinforcement learning and UCS for supervised learning, and examine the effect that different parameter values have on enhancing the model prediction, increasing accuracy and reducing the resulting rule set size.

Keywords: condition-action rules · learning classifier systems · reinforcement learning · supervised learning · fine tuning LCS

1 Introduction

The key feature behind *Rule-based machine learning* (RBML) is that it evolves arounds rules and therefore the outcome is interpretable by both computers *and* humans. RBML approaches include *learning classifier systems* (LCSs), *association rule learning* and *artificial immune systems*. The focus here is on LCSs which have been used in a variety of settings, from games [1] to industrial control [2] to policy [3] to transport [4] to controlling Boolean networks [5].

However, rule-based approaches involve a number of parameters and the quality of the outcome largely relies on assigning them appropriate values. There is distinct lack in guidance on how the run parameters should be set in order to build a good LCS, for both reinforcement learning and supervised learning problems. Here, the investigation is aimed at *single-step* problems; that is, learning problems where the reward from effecting the action on the environment is known immediately rather than after several steps, as in multi-step problems.

[⋆] This research was partly funded by EIT Digital IVZW, *Real–Time Flow* project, activity 18387–SGA2018. We thank Matthew R. Karlsen for fruitful discussions. Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In this paper we study the interplay of the LCS with the data being learned in benchmark (single-step) problems. In particular, we focus the investigation on XCS, the most popular LCS-variant for reinforcement learning, and UCS the classic variant for supervised learning problems. UCS was specifically built for knowledge extraction (learning patterns in complex datasets) and provides a contrast to reinforcement learning XCS, which allows to see the effects of various parameter values from a different LCS angle.

We focus on the run parameters suggested in seminal work in the field of LCS such as [6],[7] and use the values therein as a starting point. Neither work gives information on the source of the values. We aim to fill this gap in this paper by providing explicit guidelines on how to set the values of these key parameters of LCS for both reinforcement and supervised learning. We test both algorithms against balanced and unbalanced, generalisable problems.

The remainder of the paper is structured as follows. Section 2 provides a brief overview of LCS, XCS and UCS. Section 3 introduces the two benchmark problems used in the study while Section 4 outlines the methodology and experiments we carried out. In Section 5 we discuss the results for both learning algorithms and provide explicit guidelines for setting their run parameters. We conclude and consider future work in Section 6.

2 Rule-based Machine Learning: an overview of LCSs

Learning classifier systems (LCSs) evolve around rules - matching, predicting, updating, discovering. A rule follows an **IF : THEN** expression and comprises two parts; the *condition* (or *state*) and the *action* (or *class* or *phenotype*).

A rule is associated with a number of parameters: (i) the *reward prediction* p estimates the reward if this rule's action is eventually effected on the environment, (ii) the *prediction error* ϵ, and (iii) a fitness F which reflects the quality of the rule. These parameters are updated in every iteration of the learning cycle (cf Section 2.1, 2.2). A *classifier* is a rule with associated parameters.

The condition of a rule is expressed in a ternary alphabet; an attribute can have a value of 0 or 1, but also a # which stands for the 'don't care' value, i.e., it can match both 0 or 1. This is what allows to generalise rules and make them applicable to more than one instance - this in turn allows to capture useful data relationships without overfitting the rules to training data. For example, the rules 110:1 and 111:1 can be generalised to 11#:1 (e.g., cloudy and windy points to taking an umbrella irrespective of evening or morning). The action part of a rule represents the predicted or recommended action.

The LCS functional cycle, shown in Figure 1 comprises three components: the environment, the learning machine and, optionally, rule compression (post-processing). The environment provides as input a series of sensory situations – typically abstracted to a dataset containing so-called training instances.

The first step of the learning cycle is to receive input from the environment **(1)**. The next step is to identify and choose all rules in the *rule population* [P] whose condition part matches that of the training instance (environment input)

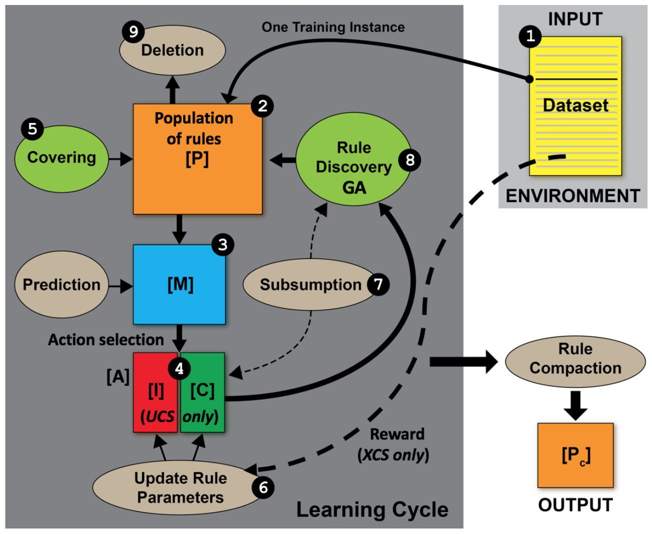


Fig. 1. The learning cycle for a generic LCS, adapted from [7].

(2). Matching compares the current input to all rules in the population [P]. If all specified attributes match, the rule is moved to the *match set* [M] **(3)**.

The next step produces the *action set* [A] **(4)** which comprises all actions associated with rules whose condition matched the input. This set forms the input used in the GA and also specifies which rules are updated after each iteration. We will have more to say on this in Sections 2.1 and 2.2.

In addition to the parameters p, ϵ, F discussed earlier, rules in [P] are associated with parameters: θ_{age} - the number of iterations that occurred after the rule was created; θ_{exp} - the number of times the rule belonged to the action set [A]; *actionSetSize* - the size of the action set to which the rule belonged; n - the number of copies of a rule appearing in the population, called *numerosity*.

If the action set is empty, or the number of actions in it do not exceed the pre-set maximum number of actions, θ_{mna}, the *Covering* mechanism is applied **(5)** – this allows to introduce new rules to the population [P] (*rule discovery*). During the process, some of the attributes in the rule's condition and action are generated using values taken from the current environment input; the rest are set to #, with probability $P\#$. Since LCS rule populations start off empty, covering is also used for population initialisation. If there is at least one action in the action set, the next step is to update the rule parameters for the classifiers that have made it into the action set [A] **(6)** (cf Sections 2.1 and 2.2).

Additionally, a mechanism called *Subsumption* is often applied **(7)** in which pairs of rules are examined and whenever a rule (*subsumer*) matches all attributes of the other rule, is more general and just as accurate it absorbs this other, more specific, rule. An *error threshold* ϵ_0 determines the maximum error a rule may have and still be able to subsume another.

Next, a *genetic algorithm* (GA) is applied **(8)**. It operates on the action set [A], with a frequency determined by θ_{ga} to produce new rules by crossover and mutation. When activated, the GA runs just once. Parent rules are selected to produce two child rules. With probability χ a two-point crossover operates on these offspring rules. Each integer pair is then mutated with a probability μ. Several selection mechanisms exist - recent literature suggests preference towards *tournament* selection, where a portion of classifiers is randomly selected from [A] and the classifier with the highest fitness is chosen to be the parent.

The last step involves a deletion mechanism which is applied to cap the population at N rules **(9)**. Whenever the population size, which is the sum of all rule numerosities, exceeds N, rules are deleted from the population until the size limit is met. Classifier experience θ_{exp} must exceed a threshold θ_{del} or fitness to impact deletion likelihood. The population cap, combined with the selective deletion of rules, results in an overall drive to improve the quality of the population.

Optionally, a post-processing step is applied to the rule population after training. This is known as *rule compaction* (see Fig. 1) and it aims to remove rules that are inexperienced, inaccurate, have a low fitness but whose error ϵ does not exceed the specified limit ϵ_0.

2.1 eXtended Classifier System (XCS)

LCS can function by receiving only the utility of the function effected on the environment (*reinforcement learning*). The *eXtended Classifier System* (XCS) is an accuracy-based LCS developed originally by [8] to target reinforcement learning problems. The key differentiating characteristics of XCSs follow.

Firstly, once the match set [M] has been created, a *prediction array* is created, which supplies the fitness-weighted predicted rewards for each of the actions within [M]. This is used in *action selection* – in *explore mode*, with probability p_{expl}, the system randomly selects from the available actions in [M]; otherwise, in *exploit mode* it chooses the action with the highest predicted reward p as indicated by the array. Certain rules within the match set (more than one possibly) contain the action that was selected. These rules form the action set [A] (Fig 1).

Secondly, there is a feedback mechanism which detects the effect of the selected action when executed on the environment and grants a particular *reward* or *payoff*. On a single-step problem such as those addressed in this paper, this reward is used directly in updating the rule parameters.

The updating can only happen in [A]. Parameter updates follow the time-weighted recency average (TWRA) principle [7], which can be described as:

$$Average_{New} = Average_{Current} + \beta(Value_{Current} - Average_{Current}) \quad (1)$$

This is directly inspired by the Widrow-Hoff update, which is commonly used in artificial neural networks (ANNs) for updating the weights of the inputs:

$$\nu_{i+1} = \nu_i + \beta(u_i - \nu_i) \quad (2)$$

When the value u_i is received, the equation above is used to record the value of variable ν_i at each iteration i. The value of ν_{i+1} would thus be a combination of the old value of ν and the newly received value of u. The split balance between using old and new values is controlled by β which is called the *learning rate* parameter. If β is 0, then no learning takes place; if $\beta = 1$, then only the new value is stored and there is no recollection of the past values. The optimal value for β lays therefore in the range from [0, 1]. The reward prediction p is given by:

$$p \leftarrow p + \beta * (r - p), \qquad 0 \leq \beta \leq 1 \tag{3}$$

where r is the reward received from the environment. The error ϵ between the reward received by the environment and the predicted reward is given by:

$$\epsilon \leftarrow \begin{cases} \epsilon + \dfrac{(|r-p| - \epsilon)}{\theta_{exp}}, & \text{if } \theta_{exp} < \dfrac{1}{\beta} \\ \epsilon + \beta * (|r-p| - \epsilon), & \text{otherwise} \end{cases} \tag{4}$$

following the same concept of split balance described earlier for p, but this time applied to ϵ (cf Fig. 6). Fitness of a rule is based on accuracy [6], given by:

$$\kappa = \begin{cases} 1, & \text{if } \epsilon < \epsilon_0 \\ \alpha * (\dfrac{\epsilon}{\epsilon_0})^{-\nu}, & \text{otherwise} \end{cases} \tag{5}$$

where ϵ_0 is a predefined error threshold, as discussed before. If the error ϵ associated with this rule (classifier's error) is greater than the error threshold ϵ_0, its accuracy decreases, based on α and ν. When α is set to a low value (approaching 0), there is going to be an acute decrease in the accuracy value; when α is set to a higher value, ν drives the rate of increase in accuracy. If ν is set to a high value, the accuracy will decrease fast; if ν is set to a low value, accuracy will drop slowly. A form of fitness-sharing is applied in XCS given by:

$$\kappa' = \frac{\kappa}{\sum\limits_{cl \in [A]} \kappa_{cl}} \tag{6}$$

where cl denotes a classifier in the action set $[A]$, κ_{cl} the accuracy associated with cl, and κ the accuracy of the present classifier.

The fitness classifier (cf Fig. 6) can thus be calculated and updated using:

$$F \leftarrow F + \beta(\kappa' - F) \tag{7}$$

2.2 sUpervised Classifier System (UCS)

The *sUpervised Classifier System (UCS)* is an accuracy-based LCS developed by [9] specifically for supervised learning problems. In supervised learning, the correct action is available from the environment. This allows UCS to focus on data mining and knowledge discovery in database type problems, e.g., performance analysis of CBA in [10]. UCS and XCS share key principles such as a niche GA and defining fitness based on accuracy, but also have key differences.

Firstly, since every environment input comes with the correct action in supervised scenarios, the match set [M] is split into two sets - a correct set [C] and an incorrect set [I] (recall step **(4)** in Fig. 1), depending on whether the population rule's action also matches the current input action. If the current action is not in [M], covering is triggered to create a rule that matches the current action. The ability to cover without guessing a suitable action in UCS is also beneficial.

Secondly, the fact the correct action is knows simplifies the reward function. The measure of accuracy is simplified as the performance of each rule is explicitly available. Each classifier in UCS has an additional parameter, *correctTrack*, which is incremented every time a classifier is part of the correct set [C]. The classifier accuracy replaces the reward prediction p in XCS and is calculated as:

$$accuracy = \frac{correctTrack}{\theta_{exp}}$$

Fitness in UCS is simpler than XCS [7] and is given by: $fitness = accuracy^{\nu}$ where ν determines the pressure on the rules to be accurate (recall Sec. 2.1).

3 Benchmark problems

In this section we describe the benchmark problems used in this research.

3.1 Mutliplexer

The *Boolean n-bit Multiplexer* defines a set of learning problems that conceptually describe the behaviour of an electronic multiplexer (MUX), a device that can combine multiple analog or digital input signals and forward them into a single output signal [11]. The 6-bit multiplexer is shown in Figure 2.

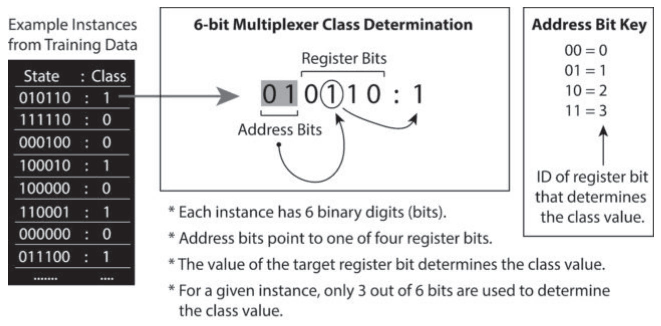
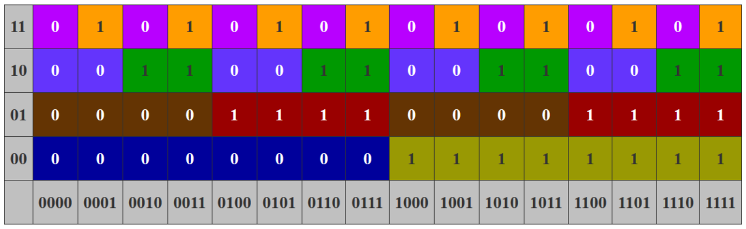

Example Instances from Training Data
State : Class
010110 : 1
111110 : 0
000100 : 0
100010 : 1
100000 : 0
110001 : 1
000000 : 0
011100 : 1
6-bit Multiplexer Class Determination
Register Bits
010110 : 1
Address Bits
Address Bit Key
00 = 0
01 = 1
10 = 2
11 = 3
ID of register bit that determines the class value.
* Each instance has 6 binary digits (bits).
* Address bits point to one of four register bits.
* The value of the target register bit determines the class value.
* For a given instance, only 3 out of 6 bits are used to determine the class value.

Fig. 2. 6-bit Multiplexer class determination - example run [7]

Multiplexer problems exhibit *epistasis*[3] (class is determined via interaction between multiple features, rather than a single one) and also heterogeneous[4] behaviour (for different instances, the class value is determined by a different subset of features) [7]. This elevates the complexity of the problem due to the non-linear association between the condition and the corresponding action.

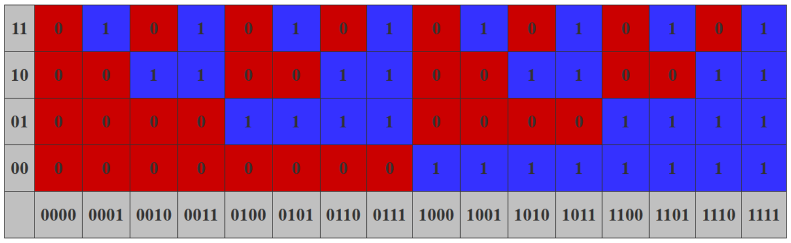
In addition, multiplexer problems contain knowledge-based patterns. Figure 3 shows the entire domain can be separated into 8 maximally general rules - theoretically this is the minimum (optimal) number of rules that the LCS must generate to completely cover the problem. Figure 4 shows the class symmetry,



11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Fig. 3. 6-bit Multiplexer patterns - address bits on vertical axis; register bits on horizontal; cells - actions. Shades represent rules that cover a pattern within the domain.

which is characteristic of a balanced problem. The multiplexer benchmark prob-



11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Fig. 4. 6-bit Multiplexer patterns - domain class symmetry

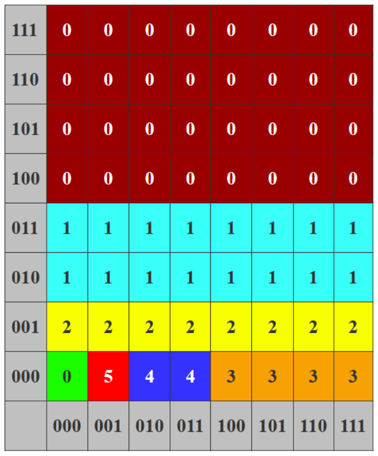
lem was chosen for use in this project as an example of a balanced, generalisable problem. Specifically, the 11-bit Multiplexer was used.

3.2 Position

The position problem benchmark is an example of an unbalanced multi-class problem where, given a binary string of length l as input, the output is determined by the value of the index position of the left-most one-valued bit. Although

[3] Type of gene interaction - a single gene is under the influence of other distinct genes.
[4] To be understood here as consisting of different or diverse components.

being a generalisable problem, the position benchmark differs from other benchmarks as it is unbalanced - this is clearly seen in Figure 5, where there are much more values for class 0 than for class 2, for example. The 9-bit Position benchmark was used in this paper.



111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
011	1	1	1	1	1	1	1	1
010	1	1	1	1	1	1	1	1
001	2	2	2	2	2	2	2	2
000	0	5	4	4	3	3	3	3
	000	001	010	011	100	101	110	111

Fig. 5. 6-bit Position problem - example of an unbalanced generalisable problem

4 Experiments: methodology and implementation[5]

We started with the "recommended" parameter values suggested by [7,6] and then altered every parameter consecutively, changing value from the smallest to the highest recommended value whilst keeping other parameters constant. For every experiment, the resulting model cross-validation accuracy, number of rules in [P] and run time were recorded. Each experiment was repeated 10 times to minimise the LCS stochastic effect and the averaged values recorded.

k-Fold Cross-Validation was used to evaluate XCS and UCS models using a set of training data. The most common values for the number of folds k are 5 and 10 [12], as they have been shown "empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance" [13]. In our case, $\mathbf{k = 5}$ was chosen, as it allowed to generate sufficiently large training and testing data sets, representative of the data being learned.

The XCS algorithm implementation was created following closely [6], which, since its publication, has become the standard to follow when implementing XCS. Minor modifications in our implementation:
1. Order of updating rule parameters changed for convenience - see Fig. 6.

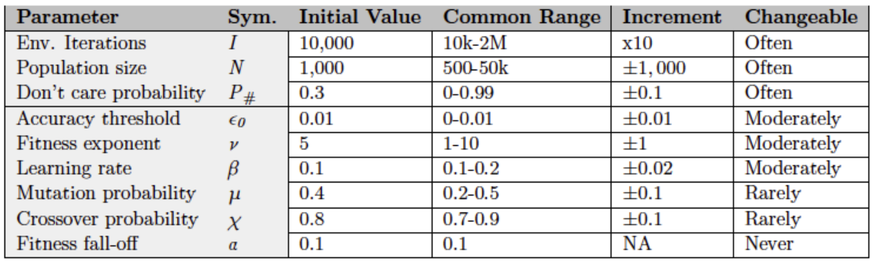
[5] The codebase used for our experiments is available on Gitlab by pointing your browser to: https://gitlab.eps.surrey.ac.uk/vs00162/lcs-suite

2. Calculation of accuracy is done via a power law function, as proposed in [14], rather than via an exponential function used in the original.
3. Covering occurs based on classifiers in [M], rather than the mean prediction of the population suggested in [15], to increase program execution speed.
4. Genetic algorithm is applied to the action set, as has been described in [16].
For UCS, the implementation follows [9].

```python
def updateXCSParameters(self, reward):
    """ Updating XCS prediction payoff, prediction error and fitness. """
    payoff = reward
    if self.action_cnt >= 1.0 / cons.beta:
        self.error += cons.beta * (abs(payoff - self.prediction) - self.error)
        self.prediction += cons.beta * (payoff - self.prediction)
    else:
        self.error = (self.error * (self.action_cnt - 1) +
                      abs(payoff - self.prediction)) / self.action_cnt
        self.prediction = (self.prediction * (self.action_cnt - 1) + payoff) /
                      self.action_cnt
    #Updating Fitness
    if self.error <= cons.offset_epsilon:
        self.accuracy = 1.0
    else:
        self.accuracy = cons.alpha * ((self.error / cons.offset_epsilon) ** (
            -cons.nu))
```

Fig. 6. Updating XCS prediction payoff, prediction error and fitness

The run parameters investigated in this work can be seen in Figure 7.



Parameter	Sym.	Initial Value	Common Range	Increment	Changeable
Env. Iterations	I	10,000	10k-2M	x10	Often
Population size	N	1,000	500-50k	$\pm1,000$	Often
Don't care probability	$P_\#$	0.3	0-0.99	±0.1	Often
Accuracy threshold	ϵ_0	0.01	0-0.01	±0.01	Moderately
Fitness exponent	ν	5	1-10	±1	Moderately
Learning rate	β	0.1	0.1-0.2	±0.02	Moderately
Mutation probability	μ	0.4	0.2-0.5	±0.1	Rarely
Crossover probability	χ	0.8	0.7-0.9	±0.1	Rarely
Fitness fall-off	α	0.1	0.1	NA	Never

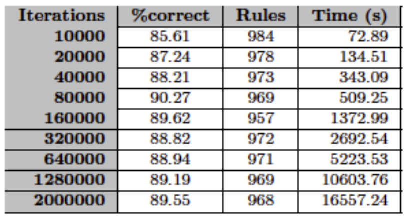
Fig. 7. Run parameters for LCS under investigation

5 Results - Guidelines for run parameters values

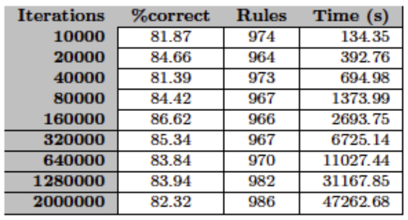
Parameter I - number of iterations.
The iteration parameter (I) specifies the maximum number of learning iterations before the evaluation of the algorithm commences. During a single iteration, a input is received from the environment and the LCS performs a learning cycle over it. The learning continues until either a specific number of iterations is

reached or the training accuracy reaches a certain threshold. Here, we opted for the former as this allows for a more fair test and comparison.

Results shown in Figures 8 and 9 demonstrate the relationship between number of iterations and accuracy, rule set size, and time for both benchmark problems, for both XCS (reinforcement learning) and UCS (supervised learning). It



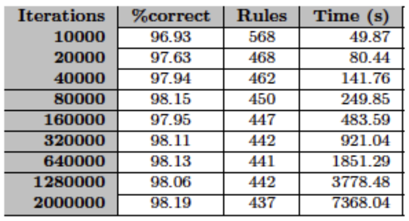
Iterations	%correct	Rules	Time (s)
10000	85.61	984	72.89
20000	87.24	978	134.51
40000	88.21	973	343.09
80000	90.27	969	509.25
160000	89.62	957	1372.99
320000	88.82	972	2692.54
640000	88.94	971	5223.53
1280000	89.19	969	10603.76
2000000	89.55	968	16557.24



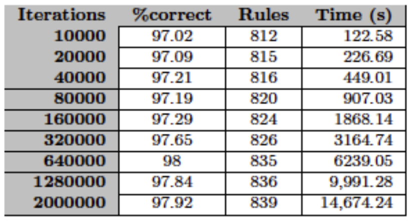
Iterations	%correct	Rules	Time (s)
10000	81.87	974	134.35
20000	84.66	964	392.76
40000	81.39	973	694.98
80000	84.42	967	1373.99
160000	86.62	966	2693.75
320000	85.34	967	6725.14
640000	83.84	970	11027.44
1280000	83.94	982	31167.85
2000000	82.32	986	47262.68

x

Fig. 8. I in XCS (left) and UCS (right) on the 11-bit multiplexer problem.



Iterations	%correct	Rules	Time (s)
10000	96.93	568	49.87
20000	97.63	468	80.44
40000	97.94	462	141.76
80000	98.15	450	249.85
160000	97.95	447	483.59
320000	98.11	442	921.04
640000	98.13	441	1851.29
1280000	98.06	442	3778.48
2000000	98.19	437	7368.04



Iterations	%correct	Rules	Time (s)
10000	97.02	812	122.58
20000	97.09	815	226.69
40000	97.21	816	449.01
80000	97.19	820	907.03
160000	97.29	824	1868.14
320000	97.65	826	3164.74
640000	98	835	6239.05
1280000	97.84	836	9,991.28
2000000	97.92	839	14,674.24

Fig. 9. I in XCS (left) and UCS (right) on the 9-bit Position problem

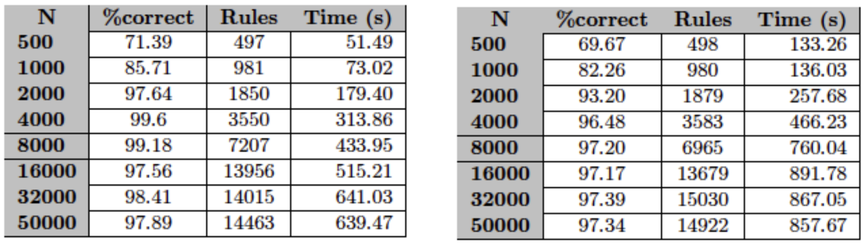
can be seen that the number of iterations and the time it takes to complete the learning cycle are directly proportional. This is true for both benchmark problems and is not affected by the algorithm type. The increase in the number of iterations causes a decrease in the amount of rules generated, indicating that the population is becoming more optimised and generalised. It also has a positive effect on the accuracy of the algorithm.

Guideline. For problems where the input has a length of 20 attributes or less, acceptable results may be achieved with [**10,000-50,000**] iterations. For larger and more complex problems, a larger number of iterations should be considered.

Parameter N - Rule population size.

The maximum population size parameter (N) specifies the allowed maximum number of rules in the population. Once N is exceeded, the deletion mechanism is activated. It is usually one of the first parameters to be tweaked in an LCS and has an important influence on the result obtained by the system.

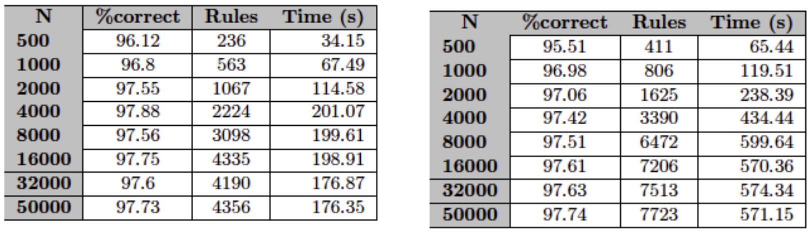
Results shown in Figures 10, 11 demonstrate the relationship between number of iterations and accuracy, rule set size, and time for both benchmark problems.



N	%correct	Rules	Time (s)
500	71.39	497	51.49
1000	85.71	981	73.02
2000	97.64	1850	179.40
4000	99.6	3550	313.86
8000	99.18	7207	433.95
16000	97.56	13956	515.21
32000	98.41	14015	641.03
50000	97.89	14463	639.47

N	%correct	Rules	Time (s)
500	69.67	498	133.26
1000	82.26	980	136.03
2000	93.20	1879	257.68
4000	96.48	3583	466.23
8000	97.20	6965	760.04
16000	97.17	13679	891.78
32000	97.39	15030	867.05
50000	97.34	14922	857.67

Fig. 10. N in XCS (left) and UCS (right) on the 11-bit multiplexer problem.



N	%correct	Rules	Time (s)
500	96.12	236	34.15
1000	96.8	563	67.49
2000	97.55	1067	114.58
4000	97.88	2224	201.07
8000	97.56	3098	199.61
16000	97.75	4335	198.91
32000	97.6	4190	176.87
50000	97.73	4356	176.35

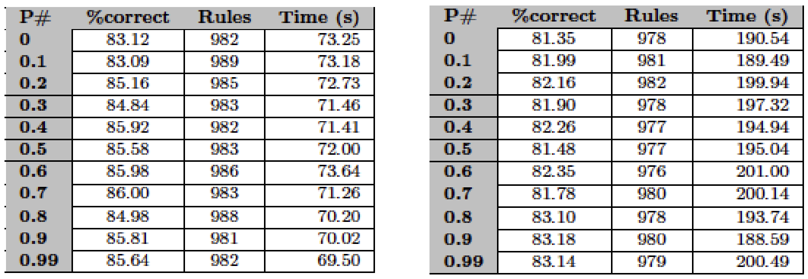
N	%correct	Rules	Time (s)
500	95.51	411	65.44
1000	96.98	806	119.51
2000	97.06	1625	238.39
4000	97.42	3390	434.44
8000	97.51	6472	599.64
16000	97.61	7206	570.36
32000	97.63	7513	574.34
50000	97.74	7723	571.15

Fig. 11. N in XCS (left) and UCS (right) on the 9-bit Position problem.

It can be seen that an increase in the population size has yielded a dramatic increase in accuracy, following a logarithmic pattern. In the position problem it is almost constant (see Fig. 11 while a sharp rise is observed when N is smaller than 500. The results allow to draw several conclusions:
(1) Parameter N plays a vital role in a learning classifier system and it is crucial to get the value correct for the system to yield adequate results.
(2) If N is set too low, the LCS will not be able to maintain good classifiers and generate a good model, resulting in low accuracy of prediction of new instances.
(3) If N is set too high, unnecessary computational resources will be consumed without any benefit. This is not only inefficient but can also lead to longer time for the algorithm to converge to an optimal solution.
The challenge therefore is to find such value of N that it is as low as possible, but at the same time provides adequate accuracy.

Guideline. N in the range of [**1000, 4000**] will yield adequate accuracy for most classification tasks, with smaller problems requiring the lower-end values and more large and complex problems the higher-end. If rule compression is applied, the rule population size will need to be set on the generous side of the proposed range.

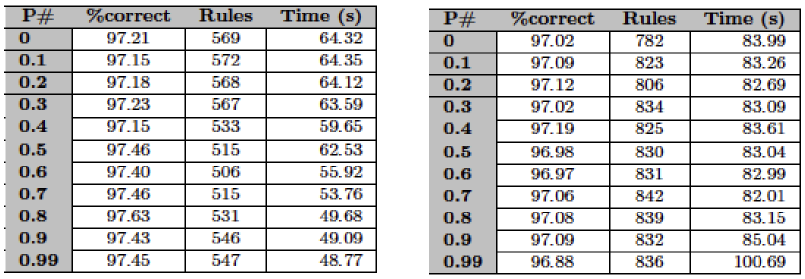
Parameter $P\#$ - "don't care" probability.
The parameter ($P\#$) specifies the probability of inserting a "don't care" symbol, denoted by #, into a rule classifier that has been generated during the covering stage. Results shown in Figures 12 and 13 demonstrate the relationship between $P\#$, algorithm accuracy, rule set size and time elapsed - without compression.



P#	%correct	Rules	Time (s)
0	83.12	982	73.25
0.1	83.09	989	73.18
0.2	85.16	985	72.73
0.3	84.84	983	71.46
0.4	85.92	982	71.41
0.5	85.58	983	72.00
0.6	85.98	986	73.64
0.7	86.00	983	71.26
0.8	84.98	988	70.20
0.9	85.81	981	70.02
0.99	85.64	982	69.50

P#	%correct	Rules	Time (s)
0	81.35	978	190.54
0.1	81.99	981	189.49
0.2	82.16	982	199.94
0.3	81.90	978	197.32
0.4	82.26	977	194.94
0.5	81.48	977	195.04
0.6	82.35	976	201.00
0.7	81.78	980	200.14
0.8	83.10	978	193.74
0.9	83.18	980	188.59
0.99	83.14	979	200.49

Fig. 12. $P\#$ in XCS (left) and UCS (right) on the 11-bit multiplexer problem.



P#	%correct	Rules	Time (s)
0	97.21	569	64.32
0.1	97.15	572	64.35
0.2	97.18	568	64.12
0.3	97.23	567	63.59
0.4	97.15	533	59.65
0.5	97.46	515	62.53
0.6	97.40	506	55.92
0.7	97.46	515	53.76
0.8	97.63	531	49.68
0.9	97.43	546	49.09
0.99	97.45	547	48.77

P#	%correct	Rules	Time (s)
0	97.02	782	83.99
0.1	97.09	823	83.26
0.2	97.12	806	82.69
0.3	97.02	834	83.09
0.4	97.19	825	83.61
0.5	96.98	830	83.04
0.6	96.97	831	82.99
0.7	97.06	842	82.01
0.8	97.08	839	83.15
0.9	97.09	832	85.04
0.99	96.88	836	100.69

Fig. 13. $P\#$ in XCS (left) and UCS (right) on the 9-bit Position problem.

The challenge is to set $P\#$ to a value which will enable the system to generate rules in the population that are not overly-general (such classifiers may maintain high fitness and succeed in certain cases, but may lead to degradation of overall performance [5]), but at the same time not overly-specific, since this will require more rules to provide a result that covers the entire problem map, which in turn might require an increase of N, leading to unnecessary increase in running time.

Guideline. For problems that are not evidently generalisable, $P\#$ must be set low – in the range of [**0, 0.3**], with more complex situations requiring $P\#$ to be set as low as 0. For problems with cleanly generalisable patterns or where high generality is needed, $P\#$ should be set at the higher end of the range, i.e, [**0.7, 0.99**]. In situations where determination of $P\#$ is too difficult, rule specificity limit (RSL) [17] is recommended.

Parameter ν - Fitness exponent.
The fitness exponent parameter (ν) is central to updating the fitness of a rule. Results shown in Figures 14 and 15 demonstrate that optimal values for ν for

ν	%correct	Rules	Time (s)
1	80.71	985	76.08
2	82.67	989	75.63
3	83.29	985	76.31
4	84.44	987	74.99
5	85.47	986	74.77
6	85.60	987	74.62
7	85.47	983	74.76
8	84.96	985	74.33
9	85.46	983	73.49
10	85.34	984	73.71

ν	%correct	Rules	Time (s)
1	73.64	980	133.43
2	76.40	986	133.57
3	78.63	982	135.58
4	80.20	979	132.78
5	81.60	981	136.21
6	83.63	971	138.00
7	85.40	968	138.22
8	85.49	971	139.90
9	85.60	962	140.89
10	86.12	966	140.00

Fig. 14. ν in XCS (left) and UCS (right) on the 11-bit multiplexer problem.

ν	%correct	Rules	Time (s)
1	96.86	550	49.69
2	97.04	557	50.11
3	97.04	554	54.78
4	96.80	563	50.15
5	97.17	544	48.97
6	96.88	563	49.34
7	96.95	549	49.28
8	96.97	548	49.35
9	97.07	543	49.13
10	96.78	552	49.48

ν	%correct	Rules	Time (s)
1	93.52	907	138.66
2	94.42	880	129.93
3	95.71	862	130.86
4	96.63	841	125.38
5	96.88	829	122.00
6	97.70	808	115.08
7	97.90	772	110.51
8	97.52	791	110.89
9	97.76	775	114.12
10	97.66	764	111.79

Fig. 15. ν in XCS (left) and UCS (right) on the 9-bit Position problem.

XCS and UCS differ. For XCS, optimal value appears to be $\nu = 5$; at this value, best balance between accuracy, rules generated and run time can be achieved. For UCS, results demonstrate that as the fitness exponent increases, algorithm accuracy increases, whilst the number of devised rules decreases even when rule compaction has not been switched on.

Guideline. Optimal values of ν for XCS and UCS differ. For XCS, optimal value appears to be $\nu = 5$, whilst for UCS working on cleanly generalisable problems and data, ν must be set high, closer to $\nu = 10$.

Parameter χ - Crossover probability.
The crossover probability parameter (χ) determines the probability of applying crossover to some selected classifiers during activation of the genetic algorithm. Results shown in Figures 16 and 17 demonstrate that increase in χ results in considerable decrease of the algorithm accuracy - best results are achieved when $\chi = 0.7$. The number of rules is also decreasing. This can be explained by the fact that by increasing χ the algorithm is encouraged to generate rules that

χ	%correct	Rules	Time (s)
0.7	85.63	989	115.33
0.8	85.81	981	115.39
0.9	84.47	982	104.39

χ	%correct	Rules	Time (s)
0.7	82.58	977	136.72
0.8	82.26	974	136.84
0.9	82.89	975	135.31

Fig. 16. χ in XCS (left) and UCS (right) on the 11-bit multiplexer problem.

χ	%correct	Rules	Time (s)
0.7	97.27	563	66.19
0.8	96.82	555	65.89
0.9	96.99	546	66.27

χ	%correct	Rules	Time (s)
0.7	97.28	836	84.43
0.8	96.74	841	84.35
0.9	97.02	836	84.63

Fig. 17. χ in XCS (left) and UCS (right) on the 9-bit Position problem.

will be more generalised, so more rules will be subsumed during the compression stage. The caveat is that more general rules do not necessarily guarantee a better resulting algorithm accuracy.

Guideline. If rule compression is not used, values in the range of [**0.7, 0.9**] will yield appropriate results; otherwise, the recommended value for χ is **0.7**.

Parameter μ - Mutation probability.
The mutation probability parameter (μ) determines the probability of mutation for every attribute of a classifier generated by the genetic algorithm. A range of values from $[0.2, 0.5]$ was investigated for this parameter, as per recommendations in [7]. Results shown in Figures 18 and 19 demonstrate that as the

μ	%correct	Rules	Time (s)
0.2	99.14	732	62.79
0.3	92.24	943	69.72
0.4	85.81	978	71.57
0.5	81.35	992	79.01

μ	%correct	Rules	Time (s)
0.2	99.51	723	141.22
0.3	94.19	902	181.50
0.4	82.07	980	191.51
0.5	76.42	993	200.94

Fig. 18. μ in XCS (left) and UCS (right) on the 11-bit multiplexer problem.

value of μ increases, the accuracy decreases, the rule count grows and the computation time increases. Also, note that for the multiplexer problem (with or without compression), the optimum value for μ appears to be 0.05, resulting in a prediction accuracy of 100%, which is a stronger result than when μ was set in the $[0.2, 0.5]$ range. A similar trend was observed for the position problem.

Guideline. μ proved to be domain specific, hence no single value can be recommended; rather, μ must be calculated using the formula $\mu = \frac{1}{l}$, as suggested in [18], where l is the number of attributes in the condition of the rules.

Guidelines for other parameters. $\beta = 0.1$ must be used for problems where noise is present, data is unbalanced, complexity is high, whilst $\beta = 0.2$ must be used for clean, straight-forward problems. The parameters α must be set to **0.1**. For clean problems, ϵ_0 must be set low, in the range [**0, 0.01**] – for

μ	%correct	Rules	Time (s)
0.2	97.7	446	54.74
0.3	97.48	493	60.26
0.4	97.27	576	66.25
0.5	96.23	649	70.71

μ	%correct	Rules	Time (s)
0.2	98.82	783	76.17
0.3	98.20	813	79.31
0.4	96.95	834	83.57
0.5	94.09	872	93.70

Fig. 19. μ in XCS (left) and UCS (right) on the 9-bit Position problem.

complex problems and problems that are expected to yield a high amount of error for each classifier, ϵ_0 must be set higher.

6 Concluding remarks and future work

The reader may notice an alignment of our guidelines for certain parameters with suggestions for XCS in [7] and [6], which effectively comprise the only closely related work. Therefore, it is worth noting that i) in addition to XCS, supervised learning (UCS) has also been addressed here, and ii) different benchmark problems have been considered, so the alignment reinforces our guidelines. In addition, we report on how rule-based machine learning algorithms, such as XCS and UCS, react to a problem with in-built class imbalance (recall Fig. 5).

The codebase used for our experiments is available on Gitlab by pointing your browser to: https://gitlab.eps.surrey.ac.uk/vs00162/lcs-suite .

Previous work has been concerned with the application of rule-based machine learning to learning individual passengers' preferences and making personalised recommendations for their onward journeys, in the event of major disruption [4], [19]. In this context, XCS was adapted to work with integer valued rather than boolean parameters, resulting in the so-called XCS*I*, as most variables in the problem domain were not binary. XCS and reinforcement learning has been applied to the problem of *controllability* in complex networks and more specifically to the control of random Boolean networks (RBNs) in [5]. In brief, XCS can be successfully used to evolve rule sets that can direct the network from any state to a target attractor state, or switch between attractor states. The network is understood to operate effectively at attractor states, hence this notion of controllability in complex networks. The rule-based machine learning algorithm in this case learns to develop a series of interventions, at most one at each time step, to successfully control the underlying network in what is a multi-step problem. The work in [20] then investigates whether the machine learning algorithm produces optimal paths to the specified attractor state.

There was little guidance in the literature in setting the run parameter values for XCSI [21] in the one-step problem (of making the correct recommendation to a passenger) as well as for XCS in the multi-step problem of directing a random boolean network to an attractor state. This was in part the motivation for the work presented in this paper. The lack of guidance on setting the run parameter values can be a barrier to adopting a rule-based machine learning approach and hence not being able to capitalise on the fact the outcome in this form of machine learning is interpretable.

Latest developments in applications of rule-based machine learning include ExTRaCS [22], which is a slimmed down version of rule-based machine learning for supervised learning problems, as well as attempts to control dynamical systems such as predator-prey models or the well-known S-shaped growth population model [23]. This kind of systems involve real rather than boolean valued parameters and in a certain important sense control becomes continuous rather than discrete. In this context, an adaptation of XCS is necessary so as to work with real valued parameters, the so-called XCSR [14]. The challenge for XCSR is to identify the initialisation of the model which leads to a desired state. This is possible after determining the *control* or *driver* nodes [24] and the series of interventions needed for steering the network to that state.

Preliminary investigations in [25] have shown that XCSR can control nonlinear systems of order 2 by appealing to *adaptive action mapping* [26]. Further work is required to overcome issues of scaling up and the action chain learning problem, which is intrinsic in reinforcement learning in multi-step problems.

The RuleML hub for interoperating structured knowledge [27] considers condition-action rules, similar to those found in rule-based machine learning and LCS. This opens up the spectrum for considering 'web-based AI' type of architectures and applications.

Another possible direction for future work concerns evolving DMN rules using XCS. DMN-based rules [28] are expressive but may become extremely complex for a human to construct. Using XCS to evolve DMN rules could produce complex and adaptive rule sets without the need for human intervention in the system.

References

1. K. Shafi and H. Abbass, "A survey of learning classifier systems in games," *IEEE Computational Intelligence Magazine*, vol. 12, pp. 42–55, 02 2017.
2. W. N. L. Browne, "The development of an industrial learning classifier system for data-mining in a steel hop strip mill," in *Applications of Learning Classifier Systems*, pp. 223–259, Springer Berlin, 2004.
3. P. Krause, A. Razavi, S. Moschoyiannis, and A. Marinos, "Stability and complexity in digital ecosystems," in *IEEE Digital Ecosystems and Technologies (IEEE DEST)*, pp. 85–90, 2009.
4. M. R. Karlsen and S. Moschoyiannis, "Learning condition–action rules for personalised journey recommendations," in *RuleML + RR 2018*, LNCS 11092, pp. 293–301, Springer, 2018.
5. M. R. Karlsen and S. Moschoyiannis, "Evolution of control with learning classifier systems," *Applied Network Science*, vol. 3, pp. 3–30, Aug 2018.
6. M. V. Butz and S. W. Wilson, "An algorithmic description of xcs," *Soft Computing*, vol. 6, no. 3, pp. 144–153, 2001.
7. R. J. Urbanowicz and W. N. Browne, *Introduction to learning classifier systems*. Berlin, Germany. Springer, 2017.
8. M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," in *International Workshop on Learning Classifier Systems*, pp. 253–272, Springer, 2000.
9. E. Bernadó-Mansilla and J. M. Garrell-Guiu, "Accuracy-based learning classifier systems: Models, analysis and applications to classification tasks," *Evolutionary Computation*, vol. 11, no. 3, pp. 209–238, 2003.

10. J. Filip and T. Kliegr, "Classification based on associations (CBA) - A performance analysis," in *Proceedings of RuleML+RR 2018 Challenge*, 2018.
11. T. Dean, *Network+ Guide to Networks.* Course Tech Cengage Learning, 2013.
12. M. Kuhn and K. Johnson, *Applied predictive modeling.* Springer, NY, USA., 2016.
13. G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning: with applications in R.* Springer, New York, NY, USA, 2015.
14. S. W. Wilson, "Get real! XCS with continuous-valued inputs," in *Learning Classifier Systems*, pp. 209–219, Springer, 2000.
15. S. W. Wilson, "Classifier fitness based on accuracy," *Evolutionary Computation*, vol. 3, pp. 149–175, June 1995.
16. S. W. Wilson, "Generalization in the xcs classifier system," in *3rd Annual Genetic Programming Conference (GP-98)*, 1998.
17. R. J. Urbanowicz and J. H. Moore, "Exstracs 2.0: description and evaluation of a scalable learning classifier system," *Evolutionary Intelligence*, vol. 8, no. 2-3, p. 89–116, 2015.
18. K. Deb, "Multi-objective optimisation using evolutionary algorithms: An introduction," in *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*, pp. 3–34, Springer, 2011.
19. M. R. Karlsen and S. Moschoyiannis, "Customer Segmentation of Wireless Trajectory Data," in *Technical Report, University of Surrey*, 2019.
20. M. R. Karlsen and S. K. Moschoyiannis, "Optimal control rules for random boolean networks," in *International Workshop on Complex Networks and their Applications*, pp. 828–840, Springer, 2018.
21. S. W. Wilson, "Compact rulesets from XCSI," in *International Workshop on Learning Classifier Systems*, pp. 197–208, Springer, 2001.
22. R. J. Urbanowicz and J. H. Moore, "Exstracs 2.0: description and evaluation of a scalable learning classifier system," *Evolutionary intelligence*, vol. 8, no. 2-3, p. 89–116, 2015.
23. *Business Dynamics: System Thinking and Modeling fro a Complex World, publisher=McGraw-HJill Higher Education, author=Sterman, J. D., year=2000.*
24. S. Moschoyiannis, N. Elia, A. Penn, D. J. B. Lloyd, and C. Knight, "A web-based tool for identifying strategic intervention points in complex systems," in *Proceedings of the Games for the Synthesis of Complex Systems (CASSTING'16 @ ETAPS 2016)*, vol. 220 of *EPTCS*, pp. 39–52, 2016.
25. V. B. Georgiev, M. R. Karlsen, L. Schoenenberger, and S. Moschoyiannis, "On controlling non-linear systems with XCSR," in *13th SICC Workshop – Complexity and the City*, 2018.
26. L. P. T. K. Nakata, M., "Selection strategy for XCS with adaptive action mapping," in *15th Annual Conference on Genetic and Evolutionary Computation*, 2016.
27. H. Boley, "The RuleML Knowledge-Interoperation Hub," in *RuleML 2016*, LNCS 9718, pp. 19–33, Springer, 2016.
28. D. Calvanese, M. Dumas, F. M. Maggi, and M. Montali, "Semantic DMN: Formalizing decision models with domain knowledge," in *RuleML+RR 2017*, LNCS 10364, pp. 70–86, Springer, 2017.