Introducing Fault Tolerance to XCS

Hong-Wei Chen
Ying-ping Chen

NCLab Report No. NCL-TR-2007005
January 2007

Natural Computing Laboratory (NCLab)
Department of Computer Science
National Chiao Tung University
329 Engineering Building C
1001 Ta Hsueh Road
HsinChu City 300, TAIWAN
http://nclab.tw/

Introducing Fault Tolerance to XCS

Hong-Wei Chen and Ying-ping Chen
Department of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
{hwchen, ypchen}@nclab.tw

January 31, 2007

Abstract

In this paper, we introduce fault tolerance to XCS and propose a new XCS framework called *XCS with Fault Tolerance* (XCS/FT). As an important branch of learning classifier systems, XCS has been proven capable of evolving maximally accurate, maximally general problem solutions. However, in practice, it oftentimes generates a lot of rules, which lower the readability of the evolved classification model, and thus, people may not be able to get the desired knowledge or useful information out of the model. Inspired by the fault tolerance mechanism proposed in field of data mining, we devise a new XCS framework by integrating the concept and mechanism of fault tolerance into XCS in order to reduce the number of classification rules and therefore to improve the readability of the generated prediction model. The workflow and operations of the XCS/FT framework are described in detail. A series of N-multiplexer experiments, including 6-bit, 11-bit, 20-bit, and 37-bit multiplexers, are conducted to examine whether XCS/FT can accomplish its goal of design. According to the experimental results, XCS/FT can offer the same level of prediction accuracy on the test problems as XCS can, while the prediction model evolved by XCS/FT consists of significantly fewer classification rules.

1 Introduction

Since the introduction of learning classification systems (LCS) [1], there have been a lot of studies and investigations on the architecture and performance of LCS. In recent years, XCS [2] has become one of the main representatives of LCS. XCS evolves rules (i.e., classifiers) through which the system gradually improves its ability to obtain the environmental reward. XCS mines the environment for the prediction pattern, which is expressed in the form of classifiers. The repeatedly refined prediction pattern allows the XCS system to make better and better decisions for action. According to the operations, XCS is potentially applicable to data mining problems because it is capable of evolving maximally accurate, maximally general rules in many environments [3]. Reports regarding applying XCS to data mining problems can be found in the literature [4, 5, 6].

In 2001, Pei et al. [7] indicated that real-world data tend to be diverse and dirty. For the real-world application, frequent pattern mining [8] often results in a large number of frequent item sets and rules, which reduce not only the efficiency but also the effectiveness of data mining since the users have to go through a large number of mined rules in order to find useful ones. As a consequence, Pei et al. [7] demonstrated that the *fault tolerance mechanism* was able to discover more general, more interesting, and more useful knowledge for data mining purpose. Additionally, they considered discovering knowledge over large amount of real-world data as fault-tolerant data mining, which should be a fruitful direction for future data mining research.

Similar situations also occur when XCS is applied to data mining. In real-world applications, although XCS can evolve accurate rule sets, it generates a large amount of classification rules. The condition of too many classification rules lowers the readability, and as a result, people cannot get desired knowledge from such a classification model. In this paper, we attempt to provide a remedy to alleviate this situation. Particularly, inspired by fault tolerance, we develop a new framework of XCS, called *XCS with Fault Tolerance* (XCS/FT) by introducing the concept of fault tolerance into XCS such that the readability can be improved and the useful knowledge of the given data can be extracted.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of XCS, followed by an introduction of fault tolerance in section 3. Section 4 proposes the framework of XCS/FT. Sections 5 and 6 present and discuss the results on the experiments of N-multiplexers, respectively. Finally, section 7 gives a summary and draws conclusions.

2 XCS

XCS [2], introduced by Wilson, is a branch of learning classifier systems (LCS) [1]. Many aspects of XCS are transferred from ZCS [9], a "zeroth level" classifier system intended to simplify Holland's canonical LCS framework. The differences between XCS and ZCS are the definition of the classifier fitness, the GA mechanism, and the more sophisticated action selection that the accuracy-based fitness makes possible. XCS has become known as the most reliable Michigan-style learning classier system for solving typical data mining and machine learning problems. In the reminder of this section, we give an overview of the important components of XCS, including the representation, the performance component, the reinforcement component, the discovery component, the macroclassifiers, as well as the covering and subsumption deletion.

2.1 Representation

XCS evolves a set of condition-action rules, which represent the obtained knowledge from the environment. The format of a classifier, i.e., a rule, is

$$\begin{aligned} &< \text{Classifier} >::= < \text{Condition} >:< \text{Action} >: \\ &\qquad\qquad < \text{Fitness} >:< \text{Payoff prediction} >:< \text{Payoff error} > \end{aligned}$$

The following description explains each part of a classifier.

1. *Condition*: The condition part C is used to check if the classifier matches the environment event. In the binary case, C is coded in the ternary alphabet $\{0, 1, \#\}^\ell$ given a problem of ℓ attributes. The symbol # represents the **don't care** condition. In other words, # matches both 0 and 1.

2. *Action*: The action part A specifies the decided action when the condition matches the environment event.

3. *Payoff prediction*: The payoff prediction p estimates the average payoff after executing the action in response to the environment event.

4. *Prediction error*: The prediction error e estimates the average error of the payoff prediction.

5. *Fitness*: The fitness F reflects the scaled average relative accuracy of the classifier.

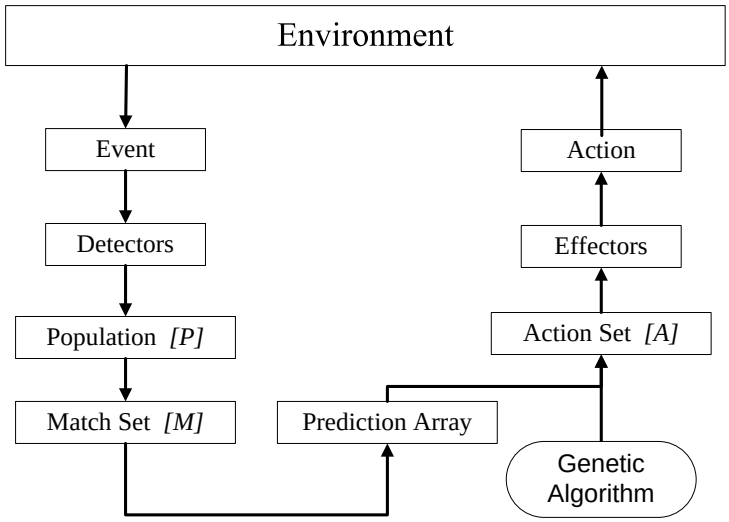

Figure 1: Framework of XCS.

2.2 Performance Component

The performance component presents the overall framework of XCS, which is shown in Figure 1. Initially, the population of XCS starts with randomly generated classifiers, potentially useful "seed" classifiers, or no classifiers. When an event occurs, a match set $[M]$ is generated from those classifiers which match the event in the whole population $[P]$. Then, the system prediction is computed for each action. The system prediction for each action is placed in the prediction array for action selection. If the current action candidates do not include all actions, random classifiers are generated through the covering operation. The system selects an action from the prediction array and forms an action set $[A]$. Finally, the chosen action is executed, and an environmental reward may be returned.

2.3 Reinforcement Component

The reinforcement component consists of updating the parameters of classifiers in the action set $[A]$ to achieve higher accuracy and to complete mappings of the problem space. The definition of variables and the procedure for updating the parameters are shown in Figure 2. With experimental results, Wilson [10] indicated if the prediction update (line 13) in comes before the error update (line 11), the prediction of a classifier in its very first update immediately predicts the correct payoff, and consequently the prediction error is set to zero. This operation can lead to faster learning in simple problems but may result in misleading for more complex problems. A more conservative strategy which puts the error update first seems to work better on hard problems [11]. More information on the update of XCS parameters can be found in the literature [10, 12].

2.4 Discovery Component

The discovery component is used to generate new classifiers in the system. Genetic discovery is invoked on the current action set $[A]$ when the average time exceeds a threshold θ_{ga} since the last GA application (recorded in the time stamp ts) upon the classifiers in $[A]$. GA selects two parental classifiers with a probability proportional to their fitness values. Two offspring classifiers are generated by reproducing, crossing, and mutating the parents. The offspring are inserted into the population. Parents stay in the population competing with their offspring.

1: // α: The fall of rate in the fitness evaluation.
2: // β: The learning rate for updating fitness, prediction, prediction error, and action set size estimate in XCS classifiers.
3: // P: Environment return payoff.
4: // ε_j: Prediction error of classifier j
5: // p_j: Prediction error of classifier j
6: // k_j: Accuracy of classifier j
7: // F_j: Fitness of classifier j
8:
9: **procedure** UPDATE OF PARAMETERS
10: // Error update
11: $\varepsilon_j \leftarrow \varepsilon_j + \beta(|P - p_j| - \varepsilon_j)$;
12: // Prediction update
13: $p_j \leftarrow p_j + \beta(|P - p_j|)$;
14: $k_j \leftarrow \exp[(\ln \alpha)(\varepsilon_j - \varepsilon_0)/\varepsilon_0] \times 0.1$;
15: // Calculate relative accuracy
16: $k'_j \leftarrow k_j/(\sum k[A])$;
17: // Calculate fitness
18: $F_j \leftarrow F_j + \beta(k_j - F_j)$;
19: **end procedure**

Figure 2: The update of parameters.

Free mutation is applied in which each attribute of the offspring condition is mutated to the other two possibilities with an equal probability. Uniform crossover is applied as the crossover operator. Parameters of the offspring are inherited from their parents; except for the experience counter *exp*, which is set to one, the numerosity *num*, which is set to one, and the fitness F, multiplied by 0.1 being rather pessimistic about the quality of the offspring. If the population size S exceeds the limit N, classifiers are removed from the population [13].

2.5 Macroclassifiers

In XCS, macroclassifiers are a type of classifiers with the numerosity parameter *num* [2]. One macroclassifier with numerosity *num* is the structural equivalence of *num* regular classifiers. Whenever XCS generates a new classifier, at the initialization step or at later stages, the population is scanned to examine whether the new classifier has the same condition and action as any existing macroclassifier does. If the system has rules with the same condition and action, the new classifier is not actually inserted into the population. Instead, the numerosity of the existing macroclassifier with the same condition and action is incremented by one. Otherwise, the new classifier is added to the population with its own numerosity field set to one. Similarly, when the macroclassifier suffers a deletion, its numerosity is decremented by one, instead of being actually deleted. If the numerosity of a macroclassifier becomes zero, the system removes the macroclassifier from the population.

2.6 Covering and Subsumption Deletion

XCS uses the genetic algorithm (GA) to generate new classifiers, and in addition, covering is another method to introduce new classifiers into the population. When an environment event occurs, the match set is determined accordingly. If the match set does not contain all possible actions defined for the environment, the system will generate new classifiers through the covering

operator to cover all possible actions. When a classifier is created through covering, its condition is made to match the current system input (i.e., the environment event), and it is given an action chosen at random. Each attribute in the condition is mutated to **don't care** (#) with a probability. Finally, the system puts the newly generated classifier into the population.

Subsumption deletion [14] is a way to improve the generalization capability of XCS. There are two kinds of subsumption: GA-subsumption and action-subsumption. Using the GA-subsumption deletion, when new classifiers are generated, they are compared to their parent classifiers. If the parent classifiers with the the experience parameter, exp, exceeding a certain threshold are more general than the new classifiers, the new classifiers are subsumed by the parents. At the same time, the numerosity of the subsuming classifier is incremented. Otherwise, the system puts the new classifiers into the population. Using the action-subsumption deletion, for each classifier R in the action set $[A]$, if there exists a classifier Q more general than R, Q subsumes R, and the numerosity of Q is incremented.

3 Fault Tolerance

Because of its versatility and capability, XCS is a promising methodology for handling classification in data mining. However, in real-world data, noise and irregularities always exist and bring difficulties and challenges to the adopted classification model. In order to provide a remedy for this problem, fault tolerance is proposed in the realm of data mining such that the rules extracted from the given data can be more useful and may offer meaningful implications. Hence, if one would like to tackle data mining tasks with XCS, certain mechanisms are required to resolve similar situations. In this section, we will channel the concept and mechanism of fault tolerance in data mining into XCS.

3.1 Fault Tolerance in Data Mining

Data mining [8], also called knowledge and data mining or knowledge-discovery in database, is the process of automatically searching large volumes of data for patterns, such as association and classification rules. However, there usually exist some non-trivial frequent patterns and rules with both high support and confidence in the database because real-world data tend to be diverse and dirty. In other words, for commonly used data mining techniques, the extracted rules could have a high prediction accuracy but may cover only a very small subset of cases. Pei et al. [7] showed an example of the real-world data as Example 1.

Example 1: For students' performance in courses, one may find rules like ***R1*** and ***R2***.

$good(x, Y)$: Student x gets a good grade in course Y, where Y is one of the following courses:

- AI: Artificial intelligence.
- Algo: Algorithm.
- DBMS: Database management system.
- DM: Data mining.
- DS: Data structure.

R1: $good(x, \text{AI}) \wedge good(x, \text{Algo}) \wedge good(x, \text{DBMS}) \wedge good(x, \text{DS}) \rightarrow good(x, \text{DM})$
R2: A student who gets good grades in at least three out of the four courses: Artificial intelligence, Algorithms, Database management system, and Data structure also gets a good grade in Data mining. □

	AI	**Algo**	**DBMS**	**DS**	**DM**
$Student_1$	good	good	good	good	good
$Student_2$	N-good	good	good	good	good
$Student_3$	good	N-good	good	good	good
$Student_4$	good	good	N-good	good	good
$Student_5$	good	good	good	N-good	good

Table 1: Data for the scenario of Example 1. N-good means that the corresponding student does not get a good grade in the course.

R1 may have high accuracy, but it is likely that R1 covers only a very small subset of cases, since there are not many students who do well in all of the four courses. We can find R2 also have high accuracy in this case. R2 is a so-called *fault-tolerant rule* in the sense that it requires the data to match only a portion of the condition. R2 can be considered more general than R1 because every student satisfying R1 also satisfies R2 but not true vice versa. Thus, R2 can cover a larger subset of cases than R1 can. In this example, both rules have high accuracy, and we believe that R2 is more interesting and more useful. It is the reason that discovering knowledge over a large amount of real-world data calls for the fault-tolerance data mining [7]. Fault-tolerance extensions to the data mining technique should be able to help us gain useful insights into the given data.

3.2 Real-World Data for XCS

XCS has been shown able to learn accurate representations of oblique data [4] and real-value data input [5, 6]. Publications also have demonstrated favorable comparisons of XCS to the well-known machine learning algorithms, such as C4.5. However, how XCS handles the real-world data is seldom described or discussed. To further understand and investigate the situation when XCS faces real-world data, we consider the following case.

Example 2: The data for the scenario of Example 1 are shown in Table 1. □

Without loss of information, XCS has to use at least four rules to model such a data set. The first rule applies **don't care** to Artificial intelligence, keeps "good" for other three courses, and implies "good" in Data mining. The other rules are similar to the first rule with **don't care** for different courses. However, we can notice that in Example 1, it is possible to use only one rule, which cannot be expressed with one XCS classifier, to describe this data set. Furthermore, for human beings, getting information and realizing implications from a single rule is oftentimes easier than from four or more rules. As a consequence, we believe that equipped with fault tolerance, XCS should be able to extend its capability and to provide classification models using fewer rules with higher readability.

3.3 Fault Tolerance in XCS Viewpoint

In this section, we show how fault tolerance proposed in data mining can be integrated into XCS at the concept level such that the rules similar to R2 described in Example 1 can be expressed by XCS classifiers. Considering the condition part with five attributes of an XCS rule, C: $\{c_1$, c_2, c_3, c_4, $c_5\}$ and the action part with 2 attributes, $A = \{a_1, a_2\}$. We may have a rule R3 as in the following Example.

	$\mathbf{c_1}$	$\mathbf{c_2}$	$\mathbf{c_3}$	$\mathbf{c_4}$	$\mathbf{c_5}$	**class**
$Event_1$	v_1	v_2	v_3	v_4	v_5	a_2
$Event_2$	N-v_1	v_2	v_3	v_4	v_5	a_2
$Event_3$	v_1	N-v_2	v_3	v_4	v_5	a_2
$Event_4$	v_1	v_2	N-v_3	v_4	v_5	a_2
$Event_5$	v_1	v_2	v_3	N-v_4	v_5	a_2
$Event_6$	v_1	v_2	v_3	v_4	N-v_5	a_2

Table 2: Example of fault tolerance. N-x stands for all possible values except for value x.

Example 3: XCS rule for the data set shown in Table 2.
R3: $(c_1 = v_1) \wedge (c_2 = v_2) \wedge (c_3 = v_3) \wedge (c_4 = v_4) \wedge (c_5 = v_5) \rightarrow (\text{class} = a_2)$ [with fault tolerance: zero or one mismatched condition attribute] □

We conceptually integrate R3 with the fault tolerance mechanism. Firstly, R3 can match event 1 because all specific attributes are matched, as a classical XCS rule. Although events 2 to 6 have attributes different from the condition of R3, they still can be matched by R3 because the fault tolerance mechanism requires to match only part of the condition. Thus, R3 can match all the six events shown in Table 2. XCS rules equipped with certain fault tolerance mechanisms may result in covering more data items.

4 XCS/FT

Based on the concept of fault tolerance, in this section, we propose a new framework of XCS, called *XCS with Fault Tolerance* (XCS/FT). We will discuss the major components of XCS/FT, including the representation, the fault tolerance match operator, the fault tolerance increase operator, and the framework of XCS/FT.

4.1 Representation

In order to enable XCS classifiers to deal with the noisy real-world data, we modify the representation of XCS rules to make them capable of tolerating "faults", which stands for the mismatched condition attributes. For this purpose, we add a parameter, called *fault tolerance* (FT), into the classifier representation as

$$< \text{Classifier} >::= < \text{Condition} >:< \text{Action} >:< \text{FT} >: \\ < \text{Payoff prediction} >:< \text{Payoff error} >:< \text{Fitness} >$$

FT indicates how many condition attributes to be ignored when a rule matches an event. Rules with FT will have a better "fault capacity" than the standard XCS rules and will result in fewer classifiers when the covering operation is triggered. For example, if there are six dirty conditions as shown in Table 3, the covering operation will generate many redundant rules for the same action in the standard XCS, but only one rule with FT can cover all the generated rules.

The purpose of FT is to provide the fault capacity. The rules generated by covering for the diverse and dirty data in the standard XCS usually have high prediction accuracy, but they may cover only a very small subset of cases. For the XCS framework, the fitness values of these rules are usually low because of the corresponding support. Through the fault tolerance mechanism, we can collect the support strength of the dirty data to generate rules with high fitness.

	A	**B**	**C**	**D**	**E**	**Class**
$Event_1$	1	0	1	0	1	2
$Event_2$	N-1	0	1	0	1	2
$Event_3$	1	N-0	1	0	1	2
$Event_4$	1	0	N-1	0	1	2
$Event_5$	1	0	1	N-0	1	2
$Event_6$	1	0	1	0	N-1	2

Table 3: Example of a fault tolerance data set. For the binary alphabet of $\{0, 1\}$, N-0 stands for 1, and N-1 stands for 0.

4.2 Fault Tolerance Match

Because of the modification in the representation for fault tolerance, the match operation of XCS has to be modified accordingly such that the new representation can take effect. Hence, a new match operation, called *fault tolerance match* (FTM), is developed. The key idea of FTM is to take FT into consideration while matching the condition and event. The definition of variables and the pseudo code for FTM is given in Figure 3.

For each match operation, we reset a variable, called *ErrorCount*, to count how many attributes in the condition part of a classifier do not match the given environment event. We compare each attribute of the condition and the event. If a mismatch occurs, we increase *ErrorCount* by one. Then, we check if *ErrorCount* is larger than FT of the rule. If true, we claim the rule mismatch the event. Otherwise, the rule matches the event thanks to its fault capacity.

4.3 Fault Tolerance Adjustment

With the representation and the match procedure for fault tolerance, we now discuss the operation to adjust the fault capacity of rules: *fault tolerance adjustment* (FTA). Firstly, we reset a counter, which keeps a record of how many rules are selected. A global parameter, called FTC, is set for the "chance" to adjust the fault capacity of the rule. We compute the average fitness of the current population. For each iteration, we randomly select a rule from the population and compare its fitness to the average. If the fitness of the selected rule is smaller, we increase the fault capacity of the rule by one. The definition of variables and the pseudo code for FTA are shown in Figure 4.

Based on the idea of XCS, researchers want to make the match set $[M]$ to cover a lot of different actions in order to make the action decision more appropriate. Hence, we suggest to use FTA to adjust FT of rules before the match set is generated for the environment event.

4.4 Framework of XCS/FT

In this paper, we concentrate on the classification problem. We model the condition as attributes and the action as class in a rule. With this interpretation, we make XCS a classification system. The overall framework of XCS/FT is shown in Figure 5. Different from the original XCS framework, which is shown in Figure 1 and can be applied to many types of problems, XCS/FT focuses on the fault tolerance mechanism and problems of classification.

With the fault-tolerance capable representation and the corresponding operations, we can now describe the flow of XCS/FT. We consider the data set of a classification problem as the environment. To simulate the occurrence of events, data items of the data set to classify are selected randomly or sequentially as the system input (i.e., the environment event). Then, FTA is applied to increase the fault capacity of the current rule set as well as to possibly increase the

```
1: // cl: Classifier
2: // cl.C: The condition part of cl
3: // cl.FT: The maximum number of mismatched attributes allowed for cl
4: // e: Event of environment
5: // ErrorCount: The counter for mismatched attributes between cl.C and e
6:
7: procedure Fault Tolerance Match(cl, e)
8:     ErrorCounter ← 0;
9:     for each attribute x ∈ cl.C do
10:        if x is not # then
11:            if x mismatches e then
12:                ErrorCounter ← ErrorCounter + 1;
13:            end if
14:            if ErrorCounter > cl.FT then
15:                return false;
16:            end if
17:        end if
18:    end for
19:    return true;
20: end procedure
```

Figure 3: Fault Tolerance Match.

```
1: // cl: Classifier
2: // cl.fit: The fitness of classifier
3: // cl.FT: The maximum number of mismatched attributes allowed for cl
4: // FTC: The maximum number of times to apply FTA during rule discovery
5: // avgFit: The average fitness of the population
6:
7: procedure Fault Tolerance Adjustment(cl)
8:     counter ← 0;
9:     Compute the avgFit of the current population;
10:    while counter < FTC do
11:        Select a classifier cl at random
12:        if (cl.fit < avgFit) then
13:            cl.FT ← cl.FT + 1
14:        end if
15:        counter ← counter + 1
16:    end while
17: end procedure
```

Figure 4: Fault Tolerance Adjustment.

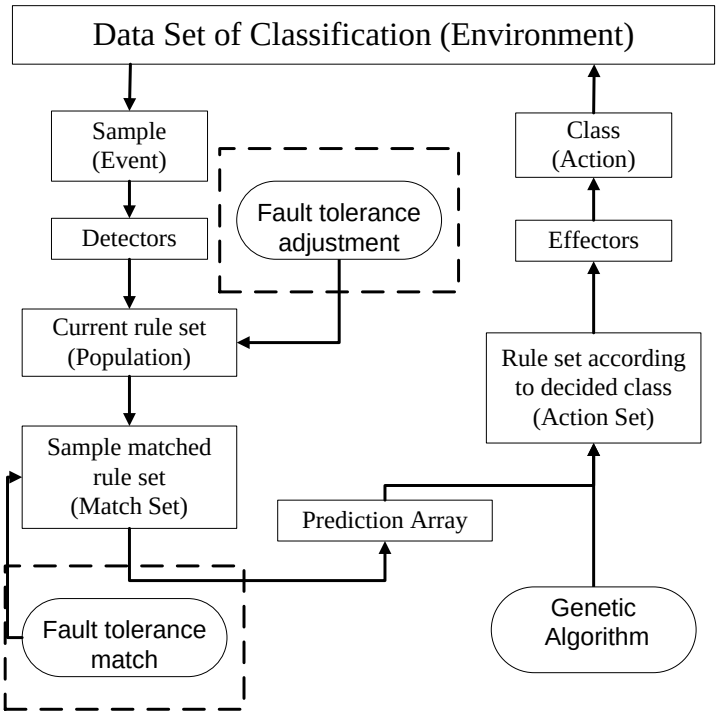

Figure 5: Framework of XCS/FT.

number of covering actions of the math set. After the application of FTA, by applying FTM, we take the rules which match the event as the match set and probably apply the covering operation. Compared to the operations conducted in the standard XCS, FTM may get the match set composed of more rules and different classes.

Moreover, the covering operation generates rules through mutating attributes of rules and assigning the class level at random. The additional rules generated by FTM might be more meaningful than that generated by covering. Therefore, we believe that FTM can provide a better match set than the standard match operation does. Then, we compute the prediction array of the match set and determine the class. The action set based on the selected class is generated, and the action is executed as the feedback to the environment. The system gets a reward from the environment for the action. According to the reward, we update the parameters of the action set to obtain a better classification system. The real-world data tend to be complex, and therefore, we use the following updating sequence: (1) prediction error; (2) prediction; (3) fitness in XCS/FT according to the suggestion for XCS. XCS/FT continues the workflow until the stop criteria is satisfied.

5 Experimental Results

In previous sections, we have briefly reviewed the original XCS framework, introduced the concept of fault tolerance, and described in detail the proposed XCS/FT. In this sections, in order to verify the effect of fault tolerance in XCS, we employ both XCS and XCS/FT to tackle the *boolean multiplexer function* and compare the *performance*, *system error*, and *population size*. Boolean multiplexer functions are defined for binary strings of length $\ell = k + 2^k$. The function value is determined by treating the first k bits as an address that indexes into the remaining 2^k bits, and the value of the indexed bit, either 0 or 1, is the function value. The *performance*

is the fraction of the last 50 exploit trials that were correct. The *system error* is the absolute difference between the system prediction for the chosen action and the actual external payoff, divided by the total payoff range (1000) and the average over the last 50 exploit trials. The *population size* is the number of macroclassifiers.

In the experiments, for the XCS part, we employ the XCS system publicly available on the Internet [15]. For the proposed XCS/FT, we modify the XCS system to include the mechanisms described in section 4 and establish the XCS/FT framework for testing. Both XCS and XCS/FT are used to handle the boolean multiplexer of four different sizes, including 6 bits, 11 bits, 20 bits, and 37 bits. Each experiment is conducted for 200 independent runs, and the statistics averaged over the 200 runs are reported in the following sections.

5.1 6-Multiplexer

Figure 6 shows the experimental results for the boolean multiplexer of 6 bits. As we can observe, XCS gets approximately 100% performance in 4000 expolit trails, and XCS/FT gets approximately 100% performance in 10000 expolit trails. For the system error, XCS gets approximately 0% system error in 3000 expolit trails, and XCS/FT gets approximately 0% system error in 10000 expolit trails. Finally, XCS evolves the population with 28.11 classifiers, and XCS/FT evolves the population with 27.79 classifiers.

Based on the experimental results for 6-Multiplexer, we can find that XCS and XCS/FT can achieve the same performance, system error, and population size when the exploit trails is appropriate. Furthermore, we can know that XCS converges faster than XCS/FT does. Overall, XCS performs slightly better than XCS/FT for 6-Multiplexer.

5.2 11-Multiplexer

Figure 7 shows the experimental results for the boolean multiplexer of 11 bits. From the results, XCS gets approximately 100% performance in 7000 expolit trails, and XCS/FT gets approximately 100% performance in 23000 expolit trails. As for the system error, XCS gets approximately 0% system error in 6000 expolit trails, and XCS/FT gets approximately 0% system error in 22000 expolit trails. For the size of the resulting rule set, XCS evolves the population with 80.59 classifiers, and XCS/FT evolves the population with 66.67 classifiers.

As the experimental results we obtained for 6-Multiplexer, similar outcomes are also for 11-Multiplexer. However, the effect of introducing fault tolerance into XCS starts to appear. In this experiment, XCS/FT on average saves 17.27% of the population size over the 200 runs.

5.3 20-Multiplexer

Figure 8 demonstrates the experimental results for the boolean multiplexer of 20 bits. XCS gets approximately 100% performance in 20000 expolit trails, and XCS/FT gets approximately 100% performance in 60000 expolit trails. For the system error, XCS gets approximately 0% system error in 25000 expolit trails, and XCS/FT gets approximately 0% system error in 50000 expolit trails. For the population size, XCS evolves the population with 256.98 classifiers, and XCS/FT evolves the population with 208.25 classifiers. As we can see, for a larger problem, the effect of fault tolerance is more significant. In this experiment, XCS/FT on average saves 18.96% of the population size over the 200 runs.

5.4 37-Multiplexer

Finally, Figure 9 presents the experimental results for the boolean multiplexer of 37 bits. In this experiment, XCS gets approximately 100% performance in 175000 expolit trails, and XCS/FT

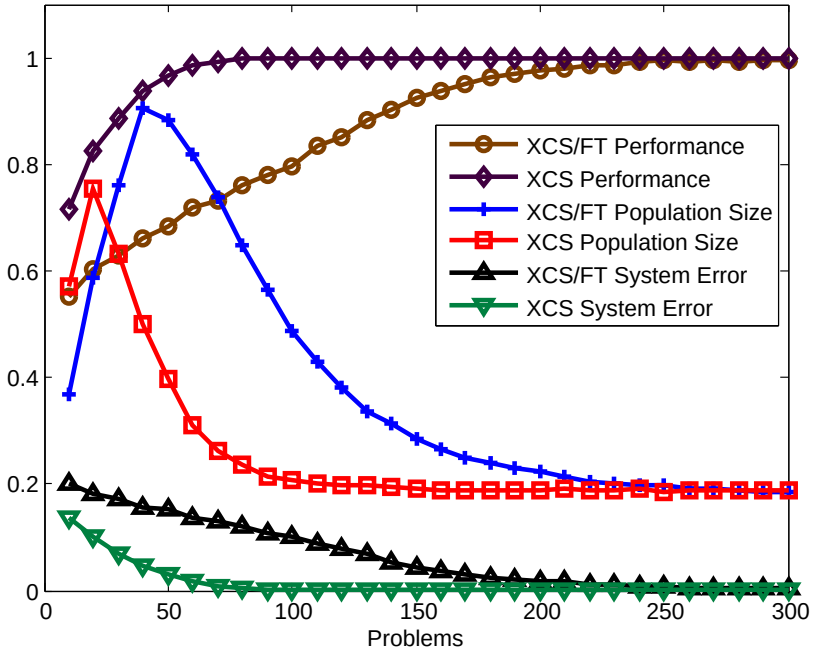

Figure 6: Experimental results for the 6-multiplexer. *Performance* is the fraction of the last 50 exploit trials that were correct. *System error* is the fraction of total payoff range. *Population size* is the number of macroclassifiers (divided by 150). *Problems* is in exploit trails (divided by 50). Parameters are $N = 400, \alpha = 0.1, \beta = 0.2, \gamma = 0.95, \theta_{ga} = 25, \varepsilon_0 = 10, \chi = 0.8, \mu = 0.04$, and $P_{\#} = 0.5$. The results are averaged over 200 runs.

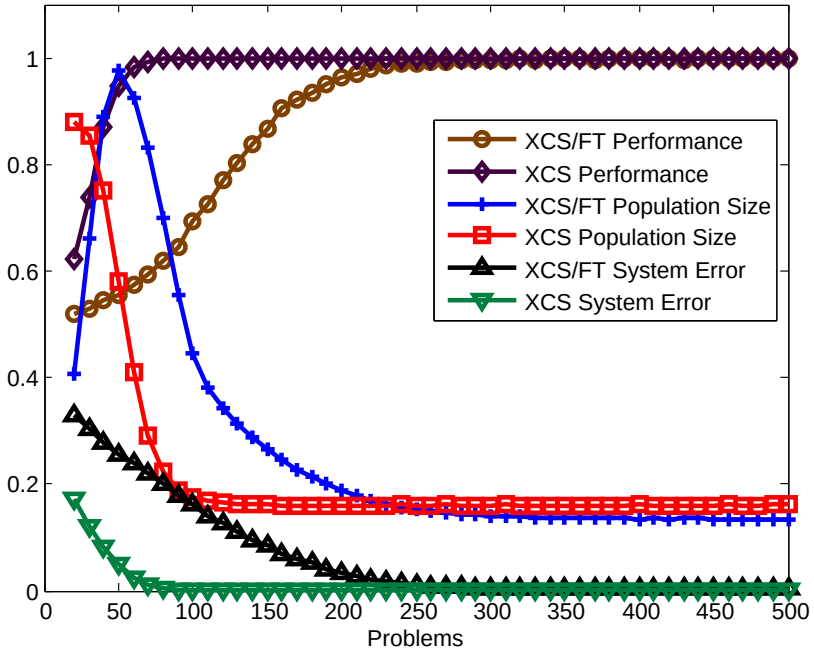

Figure 7: Experimental results for the 11-multiplexer. *Performance* is the fraction of the last 50 exploit trials that were correct. *System error* is the fraction of total payoff range. *Population size* is the number of macroclassifiers (divided by 500). *Problems* is in exploit trails (divided by 100). Parameters are $N = 800, \alpha = 0.1, \beta = 0.2, \gamma = 0.95, \theta_{ga} = 25, \varepsilon_0 = 10, \chi = 0.8, \mu = 0.04$, and $P_{\#} = 0.5$. The results are averaged over 200 runs.

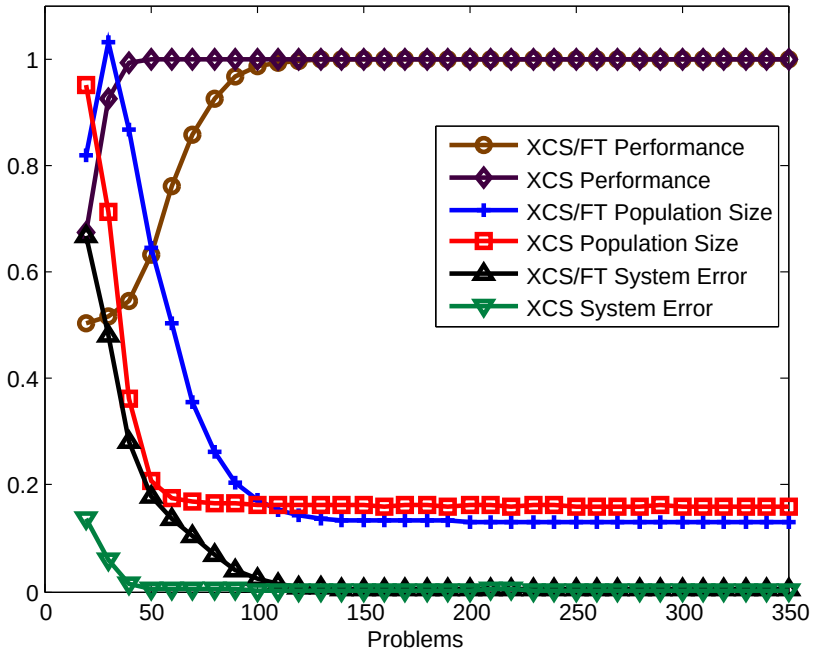

Figure 8: Experimental results for the 20-multiplexer. *Performance* is the fraction of the last 50 exploit trials that were correct. *System error* is the fraction of total payoff range. *Population size* is the number of macroclassifiers (divided by 1600). *Problems* is in exploit trails (divided by 500). Parameters are $N = 2000, \alpha = 0.1, \beta = 0.2, \gamma = 0.95, \theta_{ga} = 25, \varepsilon_0 = 10, \chi = 0.8, \mu = 0.04$, and $P_{\#} = 0.5$. The results are averaged over 200 runs.

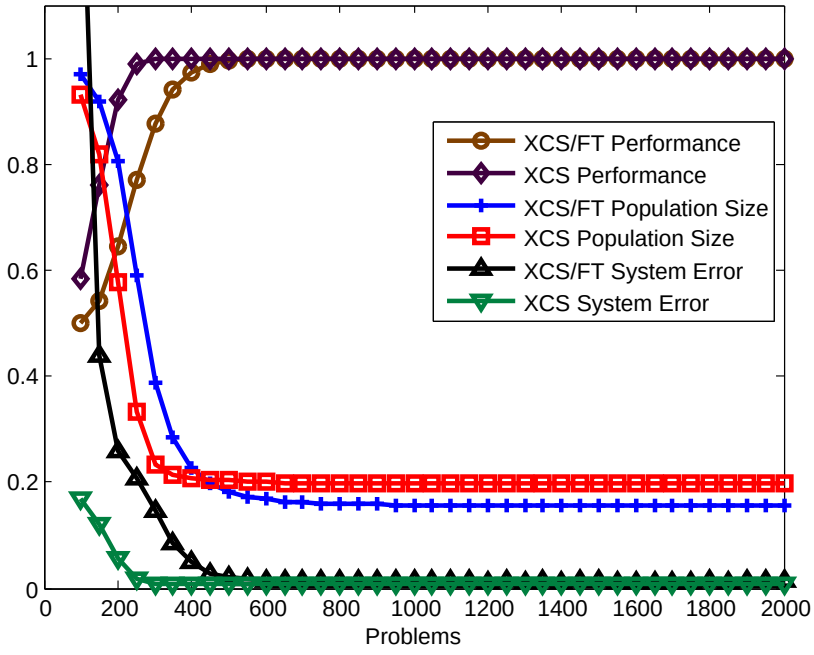

Figure 9: Experimental results for the 37-multiplexer. *Performance* is the fraction of the last 50 exploit trials that were correct. *System error* is the fraction of total payoff range. *Population size* is the number of macroclassifiers (divided by 4000). *Problems* is in exploit trails (divided by 500). Parameters are $N = 4000, \alpha = 0.1, \beta = 0.2, \gamma = 0.95, \theta_{ga} = 25, \varepsilon_0 = 10, \chi = 0.8, \mu = 0.04$, and $P_{\#} = 0.5$. The results are averaged over 200 runs.

	6-Multiplexer	11-Multiplexer
XCS	28.11	80.59
XCS/FT	27.79 (1.14%)	66.67 (17.27%)
	20-Multiplexer	37-Multiplexer
XCS	256.98	789.23
XCS/FT	208.25 (18.96%)	620.83 (21.34%)

Table 4: The average sizes of the final rule sets for different multiplexers with XCS and XCS/FT. $x\%$ is ((XCS-XCS/FT)/XCS) $\times$ 100%.

gets approximately 100% performance in 250000 expolit trails. For the system error, XCS gets approximately 0% system error in 150000 expolit trails, and XCS/FT gets approximately 0% system error in 250000 expolit trails. For the population size, XCS evolves the population with 789.23 classifiers, and XCS/FT evolves the population with 620.83 classifiers. For 37-Multiplexer, XCS/FT further saves 21.34% of the population size.

6 Discussion

The experimental results regarding the number of the evolved rule set are summarized in Table 4. We can observe that XCS and XCS/FT can evolve the final rule set with the similar rule numbers for simple problems, such as 6-Multiplexer. However, when the problem becomes more complicated, XCS/FT starts to provide the prediction model of significantly fewer rules. Hence, the ability of XCS/FT to provide models of fewer rules is empirically verified.

From the experimental results, we can also find that during the evolutionary process, XCS/FT has higher system error at the early stage than XCS does, and when the system converges, the XCS/FT system error also drops down to 0%. The fault tolerance mechanism makes XCS/FT converges more slowly. Furthermore, according to the changes in the population size, the fault tolerance mechanism can help XCS/FT to fully utilize the system resources because the population size of XCS/FT reaches the predefined maximum of the population size.

On the other hand, the boolean multiplexer problem is not the most appropriate testing problem to demonstrate the capability of XCS/FT, because there exits no "dirty" data item. Under such a condition, XCS/FT can still provide classification models of the same quality but a smaller size. Therefore, although it has to been further examined and carefully verified, we expect that XCS/FT can perform well on the real-world data containing noisy data items.

Overall, integrating the fault tolerance mechanism into XCS can be considered as adding an extra degree of freedom for the classifiers to express the encountered data items (i.e., environment events). Although fault tolerance is not totally "orthogonal" to **don't care** in a manner of speaking, by representing a classifier as a row in a table, we can think of **don't care** as the *vertical* matching relaxation (for values of attributes) and fault tolerance as the *horizontal* matching relaxation (for the number of attributes).

As a result of this interpretation, none of the theoretical results for XCS, such as that XCS can evolves maximally accurate, maximally general rule sets, has been overthrown by the experimental results for XCS/FT because the fundamental representation as well as the expressive capability of the classifier are changed. On the contrary, the ability of the XCS framework to evolve maximally accurate, maximally general rule sets is actually verified once more by the results of this study, since the XCS framework can indeed utilize the new equipment (rules with the fault capacity) to achieve the same performance (maximally accurate) with fewer rules (maximally general for the new framework).

7 Summary and Conclusions

In this paper, we first briefly reviewed XCS, followed by the introduction of the concept of fault tolerance. After channeling fault tolerance into XCS, we proposed the mechanisms of fault tolerance for XCS and described in detail the framework of XCS/FT. Finally, we implemented XCS/FT by modifying an existing XCS system and conducted a series of boolean multiplexer experiments for both XCS and XCS/FT. The experimental results confirmed that fault tolerance permitted the XCS classifier to be more expressive.

With the fault tolerance mechanism, XCS/FT can be applied to data mining and can explain a data set with the least rules. People may get the desired knowledge or information from the evolved classification model. Therefore, XCS/FT may be proven a useful technique for data mining applications, such as commerce, finance, or biology.

As for XCS/FT itself, interesting research topics and directions, including theoretical understanding and algorithmic improvement, are waiting to be explored. Research along this line should be continuously pursued and conducted in order to develop classification systems that are not only feasible in theory but also viable in practice to further advance all the related domains and disciplines.

Acknowledgments

The work was partially sponsored by the National Science Council of Taiwan under grants NSC-95-2221-E-009-092 and NSC-95-2627-B-009-001 as well as by the MOE ATU Program. The authors are grateful to the National Center for High-performance Computing for computer time and facilities.

References

[1] J. H. Holland, *Adaptation in natural and artificial systems.* University of Michigan Press, 1975, ISBN: 0-2625-8111-6.

[2] S. W. Wilson, "Classifier fitness based on accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995.

[3] T. Kovacs, "Evolving optimal populations with XCS classifier systems," Master's thesis, University of Birmingham, 1996, technical report No. CSRP-96-17.

[4] S. W. Wilson, "Mining oblique data with XCS," *Lecture Notes in Computer Science*, vol. 1996, pp. 158–176, 2000.

[5] ——, "Get real! XCS with continuous-valued inputs," *Lecture Notes in Computer Science*, vol. 1813, pp. 209–222, 2000.

[6] C. Stone and L. Bull, "For real! XCS with continuous-valued inputs," *Evolutionary Computation*, vol. 11, no. 3, pp. 299–336, 2003.

[7] J. Pei, A. K. H. Tung, and J. Han, "Fault-tolerant frequent pattern mining: Problems and challenges," in *Proceedings of 2001 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD-2001)*, May 2001.

[8] J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufmann, November 2005, ISBN: 1558609016.

[9] S. W. Wilson, "ZCS: A zeroth level classifier system," *Evolutionary Computation*, vol. 2, no. 1, pp. 1–18, 1994.

[10] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson, "How XCS evolves accurate classifiers," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2001, pp. 927–934.

[11] M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," *Lecture Notes in Computer Science*, vol. 1996, pp. 253–272, 2000.

[12] J. Horn, D. E. Goldberg, and K. Deb, "Implicit niching in a learning classifier system: Nature's way," *Evolutionary Computation*, vol. 2, no. 1, pp. 37–66, 1994.

[13] T. Kovacs, "Deletion schemes for classifier systems," in *Proceedings of the Genetic and Evolutionary Computation Conference 1999 (GECCO-99)*, July 1999, pp. 329–336.

[14] S. W. Wilson, "Generalization in the XCS classifier system," in *Proceedings of the Third Annual Conference on Genetic Programming (GP 98)*, 1998, pp. 665–674.

[15] M. V. Butz, "Java implementation of XCS," 2000, ftp://ftp-illigal.ge.uiuc.edu/pub/src/XCSJava/XCSJava1.0.tar.Z.