

**Documentation of XCS+TS C-Code 1.2**

**Martin V. Butz**

IlliGAL Report No. 2003023  
October, 2003

Illinois Genetic Algorithms Laboratory  
University of Illinois at Urbana-Champaign  
117 Transportation Building  
104 S. Mathews Avenue Urbana, IL 61801  
Office: (217) 333-2346  
Fax: (217) 244-5705

# Documentation of XCS+TS C-Code 1.2

**Martin V. Butz**

Illinois Genetic Algorithms Laboratory  
Department of General Engineering  
University of Illinois at Urbana-Champaign  
butz@illgal.ge.uiuc.edu

## Abstract

This is the documentation of the XCS 1.2 C-code released on the IlliGAL web-page. The code includes the option to apply tournament selection as well as several other new features in comparison to the XCS1.1 release. Moreover, XCS parameters as well as experimental settings can be specified in a parameter file so that recompiling is not necessary anymore. Finally, an additional output file is generated that determines means and standard deviations over the run experiments. XCS 1.2 is written in ANSI C. This documentation explains the general outline of the code, all possible parameter manipulations, and how to test this implementation on other problems (environments).

## 1 Introduction

This paper serves as a manual for using the XCS code version 1.2 written in ANSI C that is freely available at the IlliGAL anonymous FTP-site. All instructions are suited for LINUX operating systems. However, the code is written in ANSI C so that the transfer to other programming environments should not cause any problems.

The code is based on the algorithmic description in Butz and Wilson (2001) including all features in Wilson's original XCS paper (Wilson, 1995), the enhancements in Wilson (1998), Kovacs' deletion method (Kovacs, 1999), and the tournament selection option published in (Butz, Sastry, & Goldberg, 2003). The code was extensively tested on several Boolean function problems as well as several multi-step maze problems.

The papers first gives information about how to download the code, extract the source code package, compile the code, and run the code. Next, the general code structure is described and all parameters are explained. Default parameter settings are suggested. It is also shown how XCS may be applied to other classification or multi-step problems. Finally, the structure of the performance output is explained.

## 2 How to Download, Extract, Compile, and Run the code

The package with the source code and some examples are available at the IlliGAL Anonymous FTP site in <ftp://ftp-illgal.ge.uiuc.edu/pub/src/XCS/XCS1.2.tar.Z>.

### 2.1 Download and Extract

After downloading the package into a directory, the directories and files can be extracted by typing:

```
uncompress XCS1.2.tar.Z
tar xvf XCS1.2.tar
```

When extraction the files a new directory, called XCS, will be created that contains the following:

```
Envs                classifierList.c  woodsEnv.h
Makefile            classifierList.h  xcs.c
actionSelection.c  env.h            xcs.h
actionSelection.h  paramDefault     xcsMacros.c
boolFktEnv.c       woodsEnv.c       xcsMacros.h
boolFktEnv.h
```

The subdirectory `Envs` contains the codes for several simple maze environments.

## 2.2 Compile

The compilation works with the provided `Makefile` that can be found in the created `XCS1.2` directory together with the source code. A simple call of `make` compiles the files and creates the executable `xcs1.2`. A call of `make clean` deletes all `*.o`, `*.out`, and `*~` files. A previous call of `make clean` before `make` is not necessary, though.

The `Makefile` can be modified by altering the following options:

- **Line 14:** `'CC'` allows to choose the preferred C compiler. By default, the `'gcc'` compiler is chosen.
- **Line 15:** `'CC_OPTS'` allows to choose diverse compiler options.
- **Line 16:** `'LINKFINAL'` specifies the linked utilities. The `'-lm'` flag is mandatory here.
- **Line 18:** `'ENVIRONMENT'` specifies the file name in which the environment code can be found. The default is set to `boolFktEnv` — the environment containing diverse Boolean function problems.
- **Line 19:** `'OUTPUTFILE'` allows to specify the name of the executable.

## 2.3 Run the Code

After the code is compiled with the intended environment, the program can be executed by typing:

```
xcs1.2
```

In a multi-step environment an additional attribute has to be specified, which is the name of the file in which the maze is coded. For example, to start the program with the `Woods1` environment, simply type (after recompiling with the `'ENVIRONMENT'` macro in `Makefile` set to `'woodsEnv'`):

```
xcs1.2 Envs/Woods1.txt
```

Any other maze can be specified in the same manner.

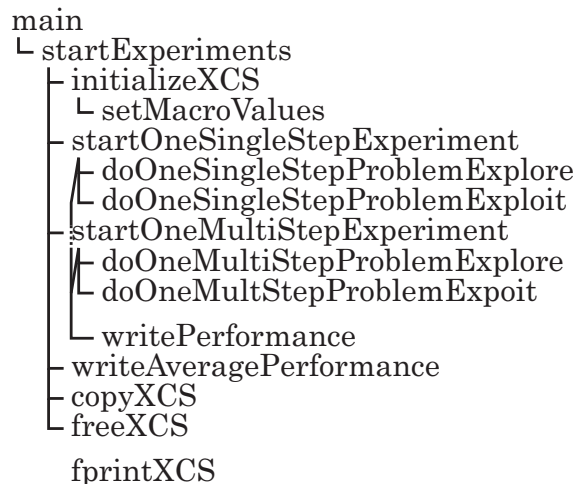


Figure 1: The code hierarchy of the code in `xcs.c`. Connections denote the functions that call more indented functions.

By default, parameter settings are read from the macros set in `'xcsMacros.h'`. Default parameters that specify environmental properties further are set in the `'env.h'` file. However, it is possible to manipulate nearly all parameter settings by specifying a parameter file when starting the program. `paramDefault` specifies one such parameter file. This file is read line by line. A line starting with a `#` symbol is ignored. Moreover, all lines are ignored that do not contain at least two strings separated by one or more spaces and/or tabulators. All other lines are parsed assuming that the first string specifies the name of the parameter to be set and the second string specifies the parameter value. Parameter names that are unknown are ignored producing an error message.

Performance output is written to two files ending with `.txt` and `.tab`. The name of the file can be specified in the `xcsMacros.h` file in the `'TAB_OUT_FILE'` function. Alternatively, parameter `'tabOutFile'` can be specified in the parameter file with the corresponding file name.

### 3 An Introduction to the Code

This section summarizes the different source code files of the XCS code. Furthermore, it gives an introduction to the implementation of the multiplexer and woods environments and it outlines how to plug-in other test environments. Finally, the possible parameter manipulations are listed.

#### 3.1 The XCS code

##### 3.1.1 `xcs.c`, `xcs.h`

In the `xcs.c` / `xcs.h` files the high level learning procedure is implemented. Moreover, the execution of multiple experiments is supported determining averages and standard deviations of the learning progress over several experiments. The `'main'` method is found here. Moreover, initialization tools, the function that parses the parameter file, and performance output methods are located here. Figure 1 sketches the correlation of the diverse functions. Function names found on one level are called from the functions on the next higher level.

The method `'startOneSingleStepExperiment'` controls the execution and monitoring of one classification problem. In order to monitor the classification progress execution switches between

exploration mode ('doOneSingleStepProblemExplore'), in which a normal learning iteration is executed, and exploitation mode ('doOneSingleStepProblemExploit'), in which the best classification is chosen and correctness of the chosen classification is monitored. No learning takes place in the exploitation mode except for covering. Similarly, the 'startOneMultiStepExperiment' controls the execution of one multi-step experiment including problem resets (teletransportation, cf. Lanzi, 1999). Again, execution switches between one exploration trial ('doOneMultiStepProblemExplore') and one exploitation trial ('doOneMultiStepProblemExploit'). In this case, parameter adjustments are also applied during exploitation as is usually done in the literature.

The header `xcs.h` specifies the 'xcs' struct that contains all parameter settings for the XCS system. All parameters in this struct can be specified in the mentioned parameter file referring to them by their name.

### 3.1.2 `classifierList.c`, `classifierList.h`

`classifierList.c` implements all functions related to the handling of classifier sets and classifiers itself. The creation of a random classifier list is implemented first. Next, the match process is specified with the possibly necessary covering mechanism. Action set generation and parameter updates in a set (usually the action set) can be found next. After that, the discovery component is implemented in which the genetic algorithm is applied to a set of classifiers (usually also the action set). Tournament selection as well as proportionate selection mechanisms can be found next. After that, crossover and mutation operators are implemented and finally insertion of offspring as well as subsumption. Next, several classifier list operations such as addition or deletion of a classifier from a list can be found as well as output operations of a classifier or a whole classifier list as well as a simple sorting routine for a classifier list. Finally, operations on the condition part are implemented. The condition part is implemented as a single connected list that specifies the specified attributes only. All unspecified attributes are implicitly set to don't care symbols.

The header `classifierList.h` specifies the structure of the condition part as well as the structure of a classifier and a classifier list.

### 3.1.3 `actionSelection.c`, `actionSelection.h`

Action selection is handled in these files. That is, the generation of a prediction array is specified as well as diverse selection methods partially dependent on the generated prediction array. Epsilon-greedy action selection is supported.

### 3.1.4 `xcsMacros.c`, `xcsMacros.h`

The header specifies all macros used in the XCS system. Particularly, all default values of the XCS system are specified here. Moreover, the random number generator is implemented supporting the generation of a random number underlying a uniform distribution or a normal distribution as well as the time-dependent randomization of the generator. Additionally, conversion operations of string to double and integer are provided.

## 3.2 Environments

Provided with the code are an environment that implements several Boolean functions as well as an implementation of a maze environment, which offers the application of diverse maze (or Woods) environments. The environments are coded in the `.c` and `.h` files of `boolFktEnv` and `woodsEnv`,

respectively. In order to apply XCS to either one of them, the parameter 'ENVIRONMENT' in the Makefile needs to be set to the appropriate environment name and the code needs to be recompiled.

### 3.2.1 General Environment Framework

In order to test XCS on a problem, an environment file needs to be implemented and specified in the Makefile. The following functions must be implemented for successful application:

- **int isMultiStep()** must return true (1) if the implemented function is a multi-step problem, i.e., if problem instances depend on each other (as for example in the maze problems). If the problem is a classification problem, the method should return zero.
- **int setEnvParam(char \*type, double value)** is the function that supports parameter settings in the environment. A parameter name specified in *type* that was not recognized in `xcs.c` will be passed on to the linked environment. The method should return 1 if the parameter specified was recognized and was successfully set to the specified value. Otherwise, zero should be returned.
- **int getConditionLength()** returns the length of the problem instances.
- **int getPaymentRange()** returns the highest possible payment received.
- **int getNumberOfActions()** returns the number of actions (classifications) possible in the problem.
- **double doAction(char \*state, int action, int \*correct)** executes (or tests) the action in the current state (or the current problem instance). Parameter *\*correct* is set to one if the classification was correct and to zero otherwise in the case of a classification problem. In a multi-step problem, *\*correct* should be set to true if reset should be called (e.g. in the woods problem, if a food position was reached). Returns the immediate reinforcement received.
- **void resetState(char \*state)** is called after each problem instance in a single step (classification) problem and after a goal was reached or a maximal number of steps was reached in a multi-step problem.
- **int initEnv(FILE \*fp)** is called in the beginning of the runs to initialize the environment. Returns if the environment was initialized properly. If zero is returned, the program exits. The file pointer specifies a file that may contain additional information (as in the multi-step problem a maze file).
- **void freeEnv()** frees any allocated space for the environment. This function is called before the end of a run.
- **void fprintfEnv(FILE \*outfile)** writes parameter settings of the environment. This information is printed on the screen after initialization as well as in the generated output files.

Once these functions are implemented with the proper meaning, XCS can be tested on the new environment instead of on one of the provided environments. Since the output is independent of the implemented environments, performance and all other measures produced in the output files should be meaningful. The two implemented environments are explained in the following and may serve as further references.

### 3.2.2 The Boolean Function Environment

The environment supports the following Boolean functions:

1. Constant
2. Random
3. (Layered) Multiplexer
4. (Layered) Concatenated Multiplexer
5. Biased Multiplexer
6. Hidden Parity
7. (Layered) Count Ones

The respective functions can be chosen by setting the parameter of the chosen function equal to 1. Only one function may be chosen at a time. The struct 'booleanEnv' in the header `boolFktEnv.h` specifies all settings that can be manipulated by setting the parameter by the means of the parameter file. Per default, one experiment of the 20 multiplexer is run. The following manipulations are possible:

- **constantFunction** Specifies that a constant function is applied.
- **randomFunction** Specifies that a random function is applied.
- **parityFunction** Chooses the parity function.
- **multiplexerFunction** Executes the multiplexer or layered multiplexer function.
- **concatenatedMultiplexer** Executes the concatenated multiplexer function (possibly layered). The number of concatenated multiplexers is determined by the number of multiplexers of size 'multiplexerBits' fit into the chosen 'conditionLength' value. Remaining bits are irrelevant and are set randomly during a run.
- **biasedMultiplexer** Chooses the biased multiplexer function.
- **countOnesFunction** Specifies that the count ones function is executed.
- **samplingBias** Determines the proportion of positive instances presented.
- **addNoiseToAction** Specifies if and which type of noise should be added to the classification. Hereby, 0 adds no noise, 1 adds Gaussian noise to the payoff resulting from either classification, 2 only for class 0, 3 only for class 1, 4.XX probability of other outcome specified in .XX (implements the alternating noise, that is, an instance that was actually classified correctly is treated as an incorrectly classified and vice versa with probability .XX), 5.XX same but only for classification 0, 6.XX same but only for classification 1.
- **actionNoiseMu** The amount of skew in the Gaussian noise case.
- **actionNoiseSigma** The standard deviation in the case of Gaussian noise.
- **paymentRange** The maximum payoff provided.

- **conditionLength** The condition length of the problem. This parameter must be set equal or larger than the number of relevant bits in the problem. If it is larger, the other bits do not matter and are generated randomly for each instance.
- **paritySize** The number of relevant bits in the hidden parity problem.
- **multiplexerBits** The number of address bits in the multiplexer problem ('conditionLength' should be equal or greater than  $(multiplexerBits + 2^{multiplexerBits})$ ).
- **payoffLandscape** If the problem is layered (applies only if the (concatenated) multiplexer problem is chosen).
- **bmpX** The x value of the xy-biased multiplexer.
- **bmpY** The y value of the xy-biased multiplexer. The 'conditionLength' should be set greater or equal to  $bmpX + 2^{bmpX}(bmpY + 2^{bmpY} - 1)$ .
- **countOnesSize** The number of relevant bits for the count ones problem.
- **countOnesType** The type of the count ones problem (0=normal count ones, 1=layered count ones).

### 3.2.3 Maze Environment

The maze environment implemented in `woodsXCS.c` and `woodsXCS.h`. Basically any standard maze or woods environment can be run by calling the initialization routine with a file that contains a proper maze or woods environment. Examples of such mazes can be found in the subdirectory `Envs`. Moreover, the environment can be controlled by several macros specified in `woodsXCS.h` (parameter setting with the parameter file is currently not supported):

- **WOODS\_LENGTH\_OF\_ONE\_ATTRIBUTE** Specifies if a perceptual attribute is coded by two or three bits.
- **WOODS\_PAYMENT\_RANGE** Specifies the reward when food is encountered.
- **WOODS\_ADD\_NOISE\_TO\_ACTION** Adds noise in the reinforcement of an action. The actions that encounter this noise are defined by the binary code of the specified value ( $0 \leq addNoiseToAction \leq 256$ ).
- **WOODS\_MU** The mean of the Gaussian noise added to the reward (if specified in `addNoiseToAction`).
- **WOODS\_SIGMA** The standard deviation of the Gaussian noise added to the reward (if specified in `addNoiseToAction`).
- **ENEMY\_EXISTS** If set to one, another agent is simulated that runs around in the maze at random.
- **RANDBIT\_EXISTS** If set to one, another random bit is added to the sensory string.
- **SLIPPROB** Specifies the probability of an action ending up in a neighboring position in positions that are slippery. Slippery positions are coded by '1' instead of '\*' in the environment text files.

- **RESET\_ON\_REWARD** Specifies if a reset should be executed when reward is encountered (usually set to one).

The default setting is found in the macro values specified in `woodsXCS.h` file. Several other macros are found here which specify a mapping from the current state to the binary sensory input. The alteration of these macros should not be necessary.

### 3.3 Available modifications of the XCS Code

Many parameters can be specified in order to change XCS's behavior, adjust it to the current problem, and specify output characteristics. This section describes all the parameters that can be modified.

- **tabOutFile** The name of the file output should be written to. Two files with this name and endings '.txt' and '.tab' will be generated.
- **nrExps** The number of experiments that should be run and over which the program will generate the final average performance measures.
- **maxNrSteps** The number of problem instances XCS learns from in a classification problem or the number of trials executed in a multi-step problem.
- **testFrequency** The number of successive learning steps after which statistics should be recorded.
- **maxPopSize** The maximal size of the population (in micro-classifiers).
- **initializePopulation** Determines, if the population should be initialized with randomly generated classifiers.
- **alpha** The drop off in accuracy for inaccurate classifiers.
- **beta** The learning rate.
- **gamma** The discount factor in multi-step experiments.
- **epsilon0** The threshold below which errors are neglected.
- **nu** The value of the power applied in the accuracy function.
- **thetaGA** The GA threshold below which a GA is not executed.
- **fitnessReduction** Specifies the fitness fraction that an offspring receives from its parents (1=no fitness reduction).
- **tournamentSize** The tournament size proportional to the current action set size.
- **forceDifferentInTournament** Forces that two different classifiers are selected in the selection process when using tournament selection
- **selectTolerance** The difference in fitness or error that is neglected in the tournament selection procedure. If there are several similarly best classifiers in a set, one of those classifiers is chosen at random.

- **crossoverType** The type of crossover to be applied (0=uniform, 1=1-point, 2=2-point crossover).
- **chiGA** The crossover probability.
- **muGA** The mutation probability.
- **doGeneralizationMutation** Specifies if purely generalizing mutation should be applied.
- **doNicheMutation** Specifies if niche mutation should be used.
- **doMAM** Specifies if the moyenne adaptive modifiée technique should be applied.
- **doGAErrorBasedSelect** Specifies if selection should be based on the error estimate of the classifiers instead of the fitness estimate.
- **delta** The fitness threshold, under which a sufficiently experienced classifier is considered as highly inaccurate (consequently enhancing its probability for deletion).
- **thetaDel** The experience required to use fitness in the determination of the deletion probability.
- **deletionType** Specifies the deletion type. Hereby, 0 = random deletion, 1 = fitness and action set size ( $|[A]|$ ) estimate bias, 2 = fitness bias only, 3 =  $|[A]|$  estimate bias only ; 4 = error bias only; 5 = error and  $|[A]|$  estimate bias.
- **dontCareProb** Specifies the don't care probability of an attribute of the condition part of a covering classifier or a randomly generated classifier.
- **doGASubsumption** Specifies if GA subsumption should be applied.
- **doActionSetSubsumption** Specifies if action set subsumption should be applied.
- **thetaSub** The experience threshold to qualify as a subsumer.
- **explorePro** Specifies the probability of choosing an action at random during exploration. Otherwise, the best action in the prediction array is executed.
- **teletransportation** Specifies the maximal number of steps executed in one exploration trial.

## 4 The Structure of the Output

The XCS implementation continuously produces output on the screen as well as in the output file specified in parameter 'tabOutFile' or, if no parameter file is generated, in the macro 'TAB\_OUT\_FILE'. To the output file name the ending '.txt' is appended. Moreover, before the program exits, another output file with ending '.tab' is generated that contains averages and standard deviations of the learning performance.

In particular, performance is recorded every 'testFrequency' steps. Error and performance are averaged over the last 'testFrequency' steps whereas all other measure are set to their current values. The different values are separated by a space.

The following performance measures are recorded: (1) the number of learning problems (trials) executed so far in this experiment, (2) the percentage of correct classifications over the last 'testFrequency' steps, or the average number of steps executed in one exploitation trial in the multi-step

setting, (3) the average reward prediction error over the last 'testFrequency' steps, (4) the current population size in macroclassifiers, (5) the average action set size estimate of the classifiers, (6) the average experience of the classifiers, (7) the average specificity in the condition parts of the classifiers, (8) the average prediction error estimate of the classifiers, (9) the average fitness of the classifiers. Experiments are separated by a line stating 'Next Experiment'.

The '.tab' file is structured similarly but records averages and standard deviations over the executed experiments. Thus, each above mentioned measure has two entries that reflect average and standard deviation. The population size values are divided by the maximal population size in order to normalize the value between zero and one. Moreover, the average specificities and standard deviations of the single attributes are recorded. Thus there are another '*conditionLength*'\*2 entries on each line.

## 5 A few Comments

The code is distributed for academic purposes without any warranty, either expressed or implied, to the extent permitted by applicable state law. We are not responsible for any damage resulting from its proper or improper use. By saying that, we want to emphasize that the code was extensively tested and we hope that no more mistakes are included.

If you have any comments or suggestions or identify any bugs, please contact the author.

## Acknowledgments

I am grateful to David E. Goldberg, Pier-Luca Lanzi, Xavier Llorca, Martin Pelikan, Kumara Sastry, and the whole IlliGAL lab for their support.

This work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant F49620-03-1-0129. Research funding for this work was also provided by a grant from the National Science Foundation under grant DMI-9908252. Additional funding was received from a Computational Science & Engineering fellowship. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government.

## References

- Butz, M. V., Sastry, K., & Goldberg, D. E. (2003). Tournament selection in XCS. *Proceedings of the Fifth Genetic and Evolutionary Computation Conference (GECCO-2003)*, 1857–1869.
- Butz, M. V., & Wilson, S. W. (2001). An algorithmic description of XCS. In Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds.), *Advances in learning classifier systems: Third international workshop, IWLCS 2000* (pp. 253–272). Berlin Heidelberg: Springer-Verlag.
- Kovacs, T. (1999). Deletion schemes for classifier systems. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, 329–336.
- Lanzi, P. L. (1999). An analysis of generalization in the XCS classifier system. *Evolutionary Computation*, 7(2), 125–149.

- Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2), 149–175.
- Wilson, S. W. (1998). Generalization in the XCS classifier system. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 665–674.