An Algorithmic Description of ACS2

Martin V. Butz[1] and Wolfgang Stolzmann[2]

[1] Department of Cognitive Psychology
University of Würzburg, Germany
butz@psychologie.uni-wuerzburg.de
[2] DaimlerChrysler AG
Research and Technology
Berlin, Germany
wolfgang.stolzmann@daimlerchrysler.com

Abstract. The various modifications and extensions of the anticipatory classifier system (ACS) recently led to the introduction of ACS2, an enhanced and modified version of ACS. This chapter provides an overview over the system including all parameters as well as framework, structure, and environmental interaction. Moreover, a precise description of all algorithms in ACS2 is provided.

1 Introduction

Anticipatory learning classifier systems (ALCSs) are a new type of classifier system. The major addition in ALCSs is that they comprise the notation of anticipations in their framework. Doing that, the systems predominantly are able to anticipate perceptual consequences of actions independent of a reinforcement prediction. Thus, ALCSs are systems that are able to form a complete anticipatory representation, that is, they build an environmental model. The model specifies which changes take place in an environment after the execution of a specific action with respect to the current situation. The essential intention behind the framework is that the representation of an environmental model allows faster and more intelligent adaptation of behavior or problem classification. By anticipating the consequences of actions with the evolving model, the system is able to adapt its behavior faster and beyond the capabilities of reinforcement learning methods (Stolzmann, Butz, Hoffmann, & Goldberg, 2000, Butz, 2001a).

The system ACS2 is derived from the original ACS framework as introduced in Stolzmann (1997) and Stolzmann (1998). Moreover, ACS2 embodies the more recently introduced genetic generalization mechanism (Butz, Goldberg, & Stolzmann, 2000). This paper provides a precise algorithmic description of ACS2. The description starts in a top down manner detailing first the overall learning cycle. The following subsections specify the single parts of the cycle in more detail. This article should be read in conjunction with Butz (2001a) in which a more comprehensive introduction of ACS2 is provided as well as a previous version of this algorithmic description. The interested reader is also referred to the other cited literature above for further background.

The next section gives an overview of ACS2's framework, rule structure, and environmental interaction. Section 3 provides the actual algorithmic description. We hope that the description in combination with the explanations about framework, structure, parameters, and environmental interaction facilitates research with ACS2. We would like to encourage feedback regarding potential problems or ambiguities. Moreover, the usage of the available ACS2 code is encouraged (Butz, 2001b).

2 Environment, Knowledge Representation, and Parameters

Before rushing into the algorithmic description, we provide an overview of the basic environmental interaction of ACS2, as well as its internal structure. Moreover, a list of all parameters in ACS2 is provided with additional suggested parameter settings and possible hints of how to set the parameters with respect to a specific problem.

2.1 Environmental Interaction

Similar to reinforcement learning agents, ACS2 interacts autonomously with an environment. It perceives situations $\sigma \in \mathcal{I} = \{\iota_1, \iota_2, ..., \iota_m\}^L$ where m is the number of possible values of each environmental attribute (or feature), $\iota_1, ..., \iota_m$ are the different possible values of each attribute, and L is the string length. Note, each attribute is not necessarily coded binary but can only take on discrete values. Moreover, the system acts upon the environment with actions $\alpha \in \mathcal{A} = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ where n specifies the number of different possible actions in the environment and $\alpha_1, ..., \alpha_n$ are the different possible actions. After the execution of an action, the reinforcement program evaluates the action in the environment and provides scalar reward $\rho(t) \in \Re$ as feedback.

Figure 1 illustrates the interaction. Hereby, the *reinforcement program* is denoted by a separate module. In accordance to Dorigo and Colombetti (1997) we separate the reinforcement from the environment, since the reinforcement could not only be provided by the environment itself, but also by an independent teacher, or even ACS2 itself, viewing the system in this case as an adaptive agent with certain needs that produce an internal reinforcement once being satisfied. ACS2 could represent certain motivations and intentions that would influence the reinforcement provision. For example, certain environmental properties might be highly desired by ACS2 so that the achievement of one of the properties would trigger high reinforcement. Thus, to what extend the actual reinforcement is influenced by the ACS2 agent is highly problem dependent.

The figure also represents the basic knowledge representation and main intuition behind ACS2. Reinforcement and perceptual feedback trigger learning in ACS2. Reinforcement as well as anticipations are represented in a common model. The knowledge represented in the model controls the action selection, i.e. the behavioral policy. Moreover, the model is intended to be exploited for improving the behavioral policy. How the model is represented in ACS2 is addressed in the following section.

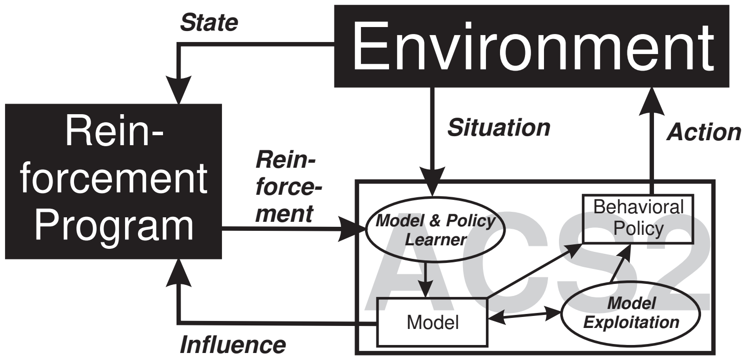

Fig. 1. ACS2 interacts with an environment perceiving environmental situations and executing actions in the environment. Reinforcement is provided by a separate reinforcement program that evaluates the current environmental state and might be more or less influenced by ACS2.

2.2 Knowledge Representation

As in other LCSs the knowledge in ACS2 is represented by a population $[P]$ of classifiers. Each classifier represents a condition-action-effect rule that anticipates the model state resulting from the execution of the specified action given the specified condition. A classifier in ACS2 always specifies a complete resulting state. It consists of the following components.

- C The *condition part* ($C \in \{\iota_1, \iota_2, ..., \iota_m, \#\}^L$) specifies the set of situations (perceptions) in which the classifier can be applied.
- A The *action part* ($A \in \mathcal{A}$) proposes an available action.
- E The *effect part* ($E \in \{\iota_1, \iota_2, ..., \iota_m, \#\}^L$) anticipates the effects that the classifier 'believes' to be caused by the specified action.
- M The *mark* ($M = (m_1, m_2, ..., m_L)$ with $m_i \subseteq \{\iota_1, \iota_2, ..., \iota_m\}$) records the properties in which the classifier did not work correctly before.
- q The *quality* (q) measures the accuracy of the anticipations.
- r The *reward prediction* (r) predicts the reward expected after the execution of action A given condition C.
- ir The *immediate reward prediction* (ir) predicts the reinforcement directly encountered after the execution of action A.
- t_{ga} The *GA time stamp* ($t_{ga} \in \mathbb{N}$) records the last time the classifier was part of an action set in which a GA was applied.
- t_{alp} The *ALP time stamp* ($t_{alp} \in \mathbb{N}$) records the time the classifier underwent the last anticipatory learning process (ALP) update.

aav The *application average* ($aav \in \Re$) estimates the ALP update frequency.
exp The *experience counter* ($exp \in \mathbb{N}$) counts the number of times the classifier underwent the ALP.
num The *numerosity* ($num \in \mathbb{N}$) specifies the number of actual (micro-)classifiers this macroclassifier represents.

The condition and effect part consist of the values perceived from the environment and '#'-symbols. A #-symbol in the condition called 'don't-care'-symbol denotes that the classifier matches any value in this attribute. A '#'-symbol in the effect part, called 'pass-through'-symbol, specifies that the classifier anticipates that the value of this attribute will not change after the execution of the specified action. All classifier parts are modified by the anticipatory learning process (ALP), the reinforcement learning application, and the genetic generalization mechanism, which are described in 3.6, sections 3.7, and 3.8, respectively.

2.3 Parameters

The following parameters control the various learning methods in ACS2. We first provide a list of all parameters and then reveal their usage and default values in further detail.

θ_i The *inadequacy threshold* ($\theta_i \in [0, 1]$) specifies when a classifier is regarded as *inadequate* determined by its quality q.
θ_r The *reliability threshold* ($\theta_r \in [0, 1]$) specifies when a classifier is regarded as *reliable* determined by q.
β The *learning rate* ($\beta \in [0, 1]$) is used in ALP and RL updates affecting q, r, ir, and aav.
γ The *discount factor* ($\gamma \in [0, 1)$) discounts the maximal reward expected in the subsequent situation.
u_{max} The *specificity threshold* ($u_{max} \in \mathbb{N}$) specifies the maximum number of specified attributes in C that are anticipated to stay the same in E.
ϵ The *exploration probability* ($\epsilon \in [0, 1]$) specifies the probability of choosing a random action similar to the ϵ-greedy policy in reinforcement learning.
θ_{ga} The *GA application threshold* ($\theta_{ga} \in \mathbb{N}$) controls the GA frequency. A GA is applied in an action set if the average delay of the last GA application of the classifiers in the set is greater than θ_{ga}.
μ The *mutation rate* ($\mu \in [0, 1]$) specifies the probability of changing a specified attribute in the conditions of an offspring to a #-symbol in a GA application.
χ The *crossover probability* ($\chi \in [0, 1]$) specifies the probability of applying crossover in the conditions of the offspring when a GA is applied.
θ_{as} The *action set size threshold* ($\theta_{as} \in \mathbb{N}$) specifies the maximal number of classifiers in an action set which is controlled by the GA.
θ_{exp} The *experience threshold* ($\theta_{exp} \in \mathbb{N}$) specifies when a classifier is regarded as *experienced* determined by exp.

Although seemingly influenced by a lot of parameters, studies showed that ACS2 is relatively robust to any chosen parameter setting. Usually, all parameters can be set to standard values. In the following, we specify default values and give the basic intuition behind the parameters.

The inadequacy threshold θ_i is usually set to the standard 0.1. Even values of 0.0 showed to not negatively influence the performance of ACS2 since in that case genetic generalization takes care of inadequate classifiers. A classifier is regarded as inadequate once its quality q falls below θ_i. In general, since inadequate classifiers are deleted by the ALP (see section 3.6), θ_i should be set to a low value. The reliability threshold θ_r on the other hand determines when a classifier is regarded as reliable and consequently becomes part of the internal environmental model. The standard value is 0.9. Generally, the higher the value is set, the longer it takes to form a complete model but the more reliable the model actually is. A more crucial parameter is the learning rate β which influences the update procedure of several parameters. The usual value is 0.05 which can be regarded as a rather passive value. The higher β, the faster parameters approach an approximation of their actual value but the more noisy the approximation is. The discount factor γ determines the reward distribution over the environmental model. A usual value is 0.95. It essentially specifies to what extend future reinforcement influences current behavior. The closer to one, the more influence delayed reward has on current behavior. The specificity threshold u_{max} constraints the specialization mechanism, namely the ALP, to what extend it is allowed to specialize the conditions. A save value is always L, the length of the perceptual string. However, if knowledge is available about the problem, then the learning process can be strongly supported by a restricted u_{max} parameter. The exploration probability ϵ determines action selection and consequently behavior. The fastest model learning is usually achieved by pure random exploration. Further biases in action selection are not considered in this description. The interested reader should refer to (Butz, 2002) in this volume.

The following parameters manipulate the genetic generalization mechanism. The GA threshold θ_{ga} controls the frequency with which genetic generalization is applied. A higher threshold assures that that the ALP has enough time to work on a generalized set if necessary. A default threshold is 100. Lower thresholds usually keep the population size down but can cause information loss in the beginning of a run. The mutation rate μ is set unusual high since it is a directly generalizing mutation. The default is set to 0.3. Lower values decrease the generalization pressure and consequently decrease the speed of conversion in the population. Higher values on the other hand can also decrease conversion because of the higher amount of over-general classifiers in the population. The crossover probability χ is usually set to the standard value of 0.8. Crossover seems to influence the process only slightly. No problem was found so far in which crossover actually has a significant effect. The action set size threshold θ_{as} is more crucial since it determines the set size for the genetic generalization application. If the threshold is set too low, the GA might cause the deletion of important classifiers and consequently disrupt the learning process. If the size is set very high, the system might learn the problem but it will take much longer since the population size will rise a lot. However, the default value of 20 worked very well in all so far applied problems. Finally, the experience threshold θ_{exp} controls when a classifier is usable as a subsumer. A low threshold might cause the incorrect propagation of an over-general classifier. However, no negative effect or major influence has been identified so far. Usually, θ_{exp} is set to 20.

3 Algorithmic Description

The provided description approaches the problem in a top down manner. First, the overall execution cycle is specified. In the subsequent sections, each sub-procedure is specified in further detail.

The following notational constraints are used in the description. Each specified sub-procedure is written in pure capital letters. The interaction with the environment and particularly requests from the environment or the reinforcement program are denoted with a colon. Moreover, to denote a certain parameter of a classifier we use the dot notation. Finally, it is necessary to note that we do not use braces or anything to denote the length of an *if clause* or a *loop* but rather use indentation as the direct control.

3.1 Initialization

In the beginning of an ACS2 run, first, all modules need to be initialized. The environment *env* must be created and the animat represented by ACS2 needs to be set to a certain position or state in the environment and so forth. Also, the reinforcement program *rp* must be initialized. Finally, ACS2 must be initialized itself. Hereby, the parameter settings are determined, the time-step counter, referred to as t, is set, and the (in the beginning usually empty) population is created. After all initialization, which we do not clarify in further detail because of their strong problem and implementation dependence, the main loop is called.

START ACS2:

```
initialize environment env
initialize reinforcement program rp
initialize ACS2
RUN EXPERIMENT with population [P] and initial time t
```

3.2 The Main Execution Loop

The main loop *RUN EXPERIMENT* is executed as long as some termination criteria are not met. In the main loop, the current situation is first sensed (perceived as input). Second, the match set $[M]$ is formed from all classifiers that match the situation. If this is not the beginning of a trial, ALP, reinforcement learning, and GA are applied in the previous action set. Next, an action is chosen for execution, the action is executed, and an action set is generated from all classifiers in $[M]$ that specify the chosen action. After some parameter updates, ALP, reinforcement learning, and GA may be applied in $[A]$ if the execution of the action led to the end of one trial. Finally, after $[A]$ is stored for learning in the next step, the loop is redone. In the case of an end of trial, $[A]_{-1}$ needs to be emptied to prevent incorrect learning over a trial barrier (i.e. since the successive situation is unrelated to the previous one).

The main loop specifies many sub-procedures denoted in capital letters which are described below in further details. Some of the procedures are more or less trivial while others are complex and themselves call other sub-procedures. Each of the sub-sections try to specify the general idea and the overall process and then give a more detailed description of single parts in successive paragraphs.

RUN EXPERIMENT($[P]$, t):

```
while(termination criteria are not met)
   σ ← env: perceive situation
   do
      GENERATE MATCH SET [M] out of [P] using σ
      if([A]_{-1} is not empty)
         APPLY ALP in [A]_{-1} considering σ_{-1}, σ, t, and [P]
         APPLY REINFORCEMENT LEARNING in [A]_{-1} using
                  ρ and max_{cl∈[M]∧cl.E≠{#}^L} (cl.q * cl.r)
         APPLY GENETIC GENERALIZATION in [A]_{-1} considering t
      act ← CHOOSE ACTION with an ε-greedy policy in [M]
      GENERATE ACTION SET [A] out of [M] according to act
      env: execute action act
      t ← t + 1
      rp: receive reward ρ
      σ_{-1} ← σ
      σ ← env: perceive situation
      if(env: is end of one trial)
         APPLY ALP in [A] considering σ, σ_{-1}, t, and [P]
         APPLY REINFORCEMENT LEARNING in [A] using ρ
         APPLY GENETIC GENERALIZATION in [A] considering t
      [A]_{-1} ← [A]
   while(not env: is end of one trial)
   env: reset position
   [A]_{-1} ← empty
```

3.3 Formation of the Match Set

The *GENERATE MATCH SET* procedure gets as input the current population $[P]$ and the current situation σ. The procedure in ACS2 is quite trivial. All classifiers in $[P]$ are simply compared to σ and all matching classifiers are added to the match set. The sub-procedure `DOES MATCH` is explained below.

GENERATE MATCH SET([P], σ):

```
initialize empty set [M]
for each classifier cl in [P]
   if(DOES MATCH classifier cl in situation σ)
      add classifier cl to set [M]
return [M]
```

The matching procedure is commonly used in LCSs. A 'don't care'-symbol # in C matches any symbol in the corresponding position of σ. A 'care' or non-# symbol only matches with the exact same symbol at that position. The *DOES MATCH* procedure checks each component in the classifier's condition $cl.C$. If a component is specified (i.e. is not a don't care symbol), it is compared with the corresponding attribute in the current situation σ. Only if all comparisons hold, the classifier matches σ and the procedure returns *true*.

DOES MATCH(cl, σ):

```
for each attribute x in cl.C
   if(x ≠ # and x ≠ the corresponding attribute in σ)
      return false
return true
```

3.4 Choosing an Action

In ACS2 usually an ϵ-greedy method is used for action selection. However, unlike non-generalizing reinforcement learning methods, it is not clear which action is actually the best to choose since one situation-action tuple is mostly represented by several distinct classifiers. In this description we chose to use the simple method that the action of the apparent most promising classifier is chosen. Since ACS2 also evolves classifiers that explicitely predict no change in the environment and there is no such thing as a waiting necessity in the problems addressed, those classifiers are excluded in the consideration. The decision is made in the provided current match set $[M]$.

CHOOSE ACTION([M]):

```
if(RandomNumber[0,1) < ε)
   return a randomly chosen action possible in env
else
   bestCl ← first cl in [M] with cl.E ≠ {#}^L
   for all classifiers cl ∈ [M]
      if(cl.E ≠ {#}^L and cl.q * cl.r > bestCl.q * bestCl.r)
         bestCl ← cl
   return cl.A
```

3.5 Formation of the Action Set

After the match set is formed and an action is chosen for execution, the *GENERATE ACTION SET* procedure forms the action set out of the match set. It includes all classifiers in the current match set $[M]$ that propose the chosen action *act* for execution.

GENERATE ACTION SET($[M]$, *act*):

```
initialize empty set [A]
for each classifier cl in [M]
   if(cl.A = act)
      add classifier cl to set [A]
return [A]
```

3.6 Anticipatory Learning Process

The application of the anticipatory learning process is rather delicate. Due to its simultaneous creation and deletion of classifiers, it needs to be assured that newly generated classifiers are added to the current action set but are not reconsidered in the current ALP application. Deleted classifiers need to be deleted from the action set without influencing the update process. The algorithmic description does not address such details. However, it is necessary to be aware of these possible problems.

The *APPLY ALP* procedure successively considers the anticipation of each classifier. If the anticipation is correct or wrong, the *EXPECTED CASE* or UNEXPECTED CASE is called, respectively. In the *UNEXPECTED CASE* procedure the quality is decreased so that it is necessary to check if the quality decreased under the inadequacy threshold θ_i. If the case, the classifier is removed (regardless of its numerosity *num* since all micro-classifiers are actually inadequate). When adding a new classifier, it is necessary to check for identical classifiers and possibly subsuming classifiers. Thus, another sub-procedure is called in this case. Finally, if no classifier in the action set anticipates the encountered change correctly, a covering classifier is generated and added. The method is usually called from the main loop. Inputs are the action set $[A]$ in which the ALP is applied, the situation σ_{-1} - action *act* tuple from which $[A]$ was generated, the resulting situation σ, the time t the action was applied, and the current population $[P]$.

Application Average The *UPDATE APPLICATION AVERAGE* procedure uses the moyenne adaptive modifée technique to reach an accurate value of the application average as soon as possible. Also the ALP time stamp t_{alp} is set in this procedure. The procedure gets the to be updated classifier *cl* and the current time t as input.

APPLY ALP([A], σ_{-1}, act, σ, t, [P]):

```
wasExpectedCase ← 0
for each classifier cl in [A]
   cl.exp++
   UPDATE APPLICATION AVERAGE of cl with respect to t
   if(cl DOES ANTICIPATE CORRECTLY σ in σ_{-1})
      newCl ←  EXPECTED CASE of cl in σ, σ_{-1}
      wasExpectedCase ← 1
   else
      newCl ←  UNEXPECTED CASE of cl in σ, σ_{-1}
      if(cl.q < θ_i)
         remove classifier cl from [P] and [A]
   if(newCl is not empty)
      newCl.t_{ga} ← t
      ADD ALP CLASSIFIER newCl to [P] and [A]
if(wasExpectedCase = 0)
   newCl ← COVER TRIPLE σ_{-1}, act, σ  with time t
   ADD ALP CLASSIFIER newCl to [P] and [A]
```

UPDATE APPLICATION AVERAGE(cl, t):

```
if(cl.exp < 1/β)
   cl.aav ← cl.aav + (t − cl.t_{alp} - cl.aav) / cl.exp
else
   cl.aav ← cl.aav + β * (t − cl.t_{alp} - cl.aav)
cl.t_{alp} ← t
```

Check Anticipation While the pass-through symbols in the effect part of a classifier directly anticipate that these attributes stay the same after the execution of an action, the specified attributes anticipate a change to the specified value. Thus, if the perceived value did not change to the anticipated value but actually stayed at the value, the classifier anticipates incorrectly. This is considered in the *DOES ANTICIPATE CORRECTLY* procedure. Inputs are the to be investigated classifier *cl*, the situation σ_{-1} where *cl* was applied in, and the resulting situation σ.

DOES ANTICIPATE CORRECTLY(cl, σ_{-1}, σ):

```
for each position i
   if(cl.E[i] = #)
      if(σ_{-1}[i] ≠ σ[i])
         return false
   else
      if(cl.E[i] ≠ σ[i] or sigma_{-1}[i] = σ[i])
         return false
return true
```

Expected Case The structure of the expected case can be separated into two parts in which either a new classifier is generated or not. No classifier is generated if the mark of the investigated classifier *cl* is either empty or no difference is detected between the mark and the relevant situation σ. In this case, the quality of the successful classifier is increased and no new classifier is returned (denoted by return 'empty'). On the other hand, if differences are detected between mark and situation, offspring is generated. It is important to consider the case where the specialization requests to specialize too many attributes. In this case, generalization of to be specified attributes is necessary. If the offspring without specialization already reached the threshold u_{max}, it is necessary to generalize the offspring to allow specialization of other attributes. The $diff$ attribute has the structure of a condition part. Its creation is specified below in another sub-procedure. The handling of probability-enhanced predictions as published in $Butz, Goldberg, and Stolzmann (2001)$, which we do not address in this description, should be caught in line 3 if the mark is not empty. Moreover, the probabilities in the parental classifier would be updated in this method.

EXPECTED CASE(cl, σ):

```
 1 diff ← GET DIFFERENCES of cl.M and σ
 2 if(diff = {#}^L)
 3    cl.q ← cl.q + β * (1 − cl.q)
 4    return empty
 5 else
 6    spec ← number of non-# symbols in cl.C
 7    specNew ← number of non-# symbols in diff
 8    child ← copy classifier cl
 9    if(spec = u_max)
10       remove randomly specific attribute in child.C
11       spec--
12       while(spec + specNew > u_max)
13          if(spec > 0 and random number [0,1) < 0.5)
14             remove randomly specific attribute in child.C
15             spec--
16          else
17             remove randomly specific attribute in diff
18             specNew--
19    else
20       while(spec + specNew > u_max)
21          remove randomly specific attribute in diff
22          specNew--
23    specify child.C with diff
24    if(child.q < 0.5)
25       child.q = 0.5
26    child.exp ← 1
27    return child
```

Difference Determination The difference determination needs to distinguish between two cases. (1) Clear differences are those where one or more attributes in the mark M do not contain the corresponding attribute in the situation σ. (2) Fuzzy differences are those where there is no clear difference but one or more attributes in the mark M specify more than the one value in σ. In the first case, one random clear difference is specified while in the latter case all differences are specified.

GET DIFFERENCES(M, σ):

```
diff ← {#}^L
if(M is not empty)
    type1 ← type2 ← 0
    for all positions i in σ
        if(M[i] does not contain σ[i])
            type1++
        else if(|M[i]| > 1)
            type2++
    if(type1 > 0)
        type1 ← random number [0,1) * type1
        for all positions i in σ
            if(M[i] does not contain σ[i])
                if(type1 = 0)
                    diff[i] ← σ[i]
                type1--
    else if(type2 > 0)
        for all positions i in σ
            if(|M[i]| > 1)
                diff[i] ← σ[i]
return diff
```

Unexpected Case The unexpected case is rather simply structured. Important is the criterion for generating an offspring classifier. An offspring is generated only if the effect part of the to be investigated classifier *cl* can be modified to anticipate the change from σ_{-1} to σ correctly by only *specializing* attributes. If this is the case, an offspring classifier is generated that is specialized in condition and effect part where necessary. The experience of the offspring classifier is set to one.

UNEXPECTED CASE(cl, σ_{-1}, σ):

```
cl.q ← cl.q − β * (cl.q)
SET MARK cl.M with σ_{-1}
for all positions i in σ
    if(cl.E[i] ≠ #)
        if(cl.E[i] ≠ σ[i] or σ_{-1}[i] = σ[i])
            return empty
child ← copy classifier cl
for all positions i in σ
    if(cl.E[i] = # and σ_{-1}[i] ≠ σ[i])
        child.C[i] ← σ_{-1}[i]
        child.E[i] ← σ[i]
if(cl.q < 0.5)
    cl.q = 0.5
child.exp ← 1
return child
```

Covering The idea behind covering is that ACS2 intends to cover all possible situation-action-effect triples. In the ALP, if such a triple was not represented by any classifier in the action set, covering is invoked. Covering generates a classifier that specifies all changes from the previous situation σ_{-1} to situation σ in condition and effect part. The action part A of the new classifier is set to the executed action *act*. The time is set to the current time t. An empty classifier is referred to as a classifier that consists only of #-symbols in condition and effect part. Note, since the experience counter is set to 0, the application average parameter *aav* will be directly set to the delay til its first application in its first application, so that the initialization is not particularly important. Moreover, the quality $cl.q$ is set to 0.5 and the reward prediction $cl.r$ is set to zero to prevent 'reward bubbles' in the environmental model.

COVER TRIPLE(σ_{-1}, act, σ, t):

```
child ← generate empty classifier with action act
for all positions i in σ
    if(σ_{-1}[i] ≠ σ[i])
        child.C[i] ← σ_{-1}[i]
        child.E[i] ← σ[i]
child.exp ← child.r ← child.aav ← 0
child.t_{alp} ← child.t_{ga} ← t
child.q ← 0.5
child.num ← 1
return child
```

Insertion in the ALP If the ALP generates offspring, insertion distinguishes between several cases. First, the method checks if there is a classifier that subsumes the insertion candidate cl. If there is none, the method looks for equal classifiers. If none was found, classifier cl is inserted as a new classifier in the population $[P]$ and the current action set $[A]$. However, if a subsumer or equal classifier was found, the quality of the old classifier is increased and the new one is discarded. The subsumption method is described in section 3.9 since the GA application uses the same method. Note, in the equality check it is not necessary to check for the identical action since all classifiers in $[A]$ have the same action.

ADD ALP CLASSIFIER(cl, [A], [P]):

```
oldCl ← empty
for all classifiers c in [A]
   if(c IS SUBSUMER of cl)
      if(oldCl is empty or c.C is more general than oldCl.C)
         oldCl ← c
if(oldCl is empty)
   for all classifiers c in [A]
      if(c is equal to cl in condition and effect part)
         oldCl ← c
if(oldCl is empty)
   insert cl in [A] and [P]
else
   oldCl.q ← oldCl.q + beta * (1 − oldCl.q)
   discard classifier cl
```

3.7 Reinforcement Learning

The reinforcement portion of the update procedure follows the idea of Q-learning (Watkins, 1989). Classifier's reward predictions are updated using the immediate reward ρ and the discounted maximum payoff predicted in the next time-step $maxP$. The major difference is that ACS2 does not store an explicit model but only more or less generalized classifiers that represent the model. Thus, for the reinforcement learning procedure to work successfully, it is mandatory that the model is specific enough for the reinforcement distribution. Lanzi (2000) formulizes this insight in a general classifier system framework. The procedure updates the reward predictions r as well as the immediate reward predictions ir of all classifiers in the action set $[A]$.

APPLY REINFORCEMENT LEARNING([A], ρ, maxP):

```
for each classifier cl in [A]
   cl.r ← cl.r + β * (ρ + γmaxP − cl.r)
   cl.ir ← cl.ir + β * (ρ − cl.ir)
```

3.8 Genetic Generalization

The GA in ACS2 is a genetic generalization of condition parts. Due to the modified generalizing mutation and the evolutionary pressures, the generalizing nature of the GA is realized. The method starts by determining if a GA should actually take place, controlled by the t_{ga} time stamp and the actual time t. If a GA takes place, preferable accurate, over-specified classifiers are selected, mutated, and crossed. Before the insertion, excess classifiers are deleted in $[A]$. Several parts of the processes are specified by sub-procedures which are described after the description of the main GA procedure.

APPLY GENETIC GENERALIZATION ($[A]$, *t*)*:*

```
if($t - \sum_{cl \in [A]} cl.t_{ga}\ cl.num / \sum_{cl \in [A]} cl.num$ > $\theta_{GA}$)
    for each classifier $cl$ in $[A]$
        $cl.t_{ga} \leftarrow$ actual time $t$
    $parent_1 \leftarrow$ SELECT OFFSPRING in $[A]$
    $parent_2 \leftarrow$ SELECT OFFSPRING in $[A]$
    $child_1 \leftarrow$ copy classifier $parent_1$
    $child_2 \leftarrow$ copy classifier $parent_2$
    $child_1.num \leftarrow child_2.num \leftarrow$ 1
    $child_1.exp \leftarrow child_2.exp \leftarrow$ 1
    APPLY GENERALIZING MUTATION on $child_1$
    APPLY GENERALIZING MUTATION on $child_2$
    if(RandomNumber$[0, 1)$ < $\chi$)
        APPLY CROSSOVER on $child_1$ and $child_2$
        $child_1.r \leftarrow child_2.r \leftarrow (parent_1.r + parent_2.r)/2$
        $child_1.q \leftarrow child_2.q \leftarrow (parent_1.q + parent_2.q)/2$
    $child_1.q \leftarrow child_1.q/2$
    $child_2.q \leftarrow child_2.q/2$
    DELETE CLASSIFIERS in $[A], [P]$ to allow the insertion
                          of 2 children
    for each $child$
        if($child.C$ equals $\{\#\}^L$)
            next $child$
        else
            ADD GA CLASSIFIER $child$ to $[P]$ and $[A]$
```

Offspring Selection Offspring in the GA is selected by a Roulette-Wheel Selection. The process chooses a classifier for reproduction in set $[A]$ proportional to its quality to the power three. First, the sum of all values in set $[A]$ is computed. Next, the roulette-wheel is spun. Finally, the classifier is chosen according to the roulette-wheel result.

SELECT OFFSPRING([A]):

```
qualitySum ← 0
for each classifier cl in [A]
   qualitySum ← qualitySum + cl.q^3
choicePoint ← RandomNumber[0,1) * qualitySum
qualitySum ← 0
for each classifier cl in [A]
    qualitySum ← qualitySum + cl.q^3
    if(qualitySum > choicePoint)
       return cl
```

Mutation As has been noted before, the mutation process in ACS2 is a generalizing mutation of the condition part $cl.C$. Specific attributes in the conditions are changed to #-symbols with a certain probability μ. The process works as follows:

APPLY GENERALIZING MUTATION (cl):

```
for all positions i in cl.C
if(cl.C[i] ≠ #)
    if(RandomNumber[0,1) < μ)
       cl.C[i] ← #
```

Crossover The crossover application, as mutation, is only applied to the condition part. Crossover is only applied, if the two offspring classifiers $cl1$ and $cl2$ anticipate the same change. This restriction further assures the combination of classifiers that inhabit the same environmental niche. Our description shows two-point crossover.

APPLY CROSSOVER (cl1, cl2):

```
if(cl1.E ≠ cl2.E)
    return
x ← RandomNumber[0,1) (length of cl_1.C +1)
do
    y ← RandomNumber[0,1) (length of cl_1.C +1)
while(x = y)
if(x > y)
    switch x and y
i ← 0
do
    if(x ≤ i and i < y)
       switch cl_1.C[i] and cl_2.C[i]
    i++
while(i < y)
```

GA Deletion While the reproduction process uses a form of roulette wheel selection, GA deletion in ACS2 applies a modified tournament selection process.

Approximately a third of the action set size takes part in the tournament. The classifier is deleted that has a significantly lower quality than the others. If all classifiers have a similar quality, marked classifiers are preferred for deletion before unmarked classifiers and the least applied classifier is preferred among only marked or only unmarked classifiers. First, however, the method controls if and how long classifiers need to be deleted in $[A]$. The parameter $inSize$ specifies the number of children that will still be inserted in the GA process. Note, the tournament is held among the micro-classifiers. If a classifier is removed from the population, that is, if its numerosity reaches zero, the classifier needs to be removed from the action set $[A]$ as well as from the whole population $[P]$.

DELETE CLASSIFIERS($[A]$*,* $[P]$*, inSize):*

```
while(inSize + ∑_{cl∈[A]} cl.num > θ_as)
  clDel ← empty
  for each micro-classifier cl in [P]
    if(RandomNumber[0,1) < 1/3)
      if(clDel is empty)
        clDel ← cl
      else
        if(cl.q − clDel.q < −0.1)
          clDel ← cl
        if(|cl.q − clDel.q| ≤ 0.1)
          if(cl.M is not empty and clDel.M is empty)
            clDel ← cl
          else if(cl.M is not empty or clDel.M is empty)
            if(cl.aav > clDel.aav)
              clDel ← cl
  if(clDel is not empty)
    if(clDel.num > 1)
      clDel.num--
    else
      remove classifier cl from [P] and [A]
```

Insertion in the GA Although quite similar to the ALP insertion, the insertion method in the GA differs in two important points. First, the numerosity num rather than the quality q of an old, subsuming or identical classifier is increased. Second, the numerosity of an identical classifier is only increased if the identical classifier is not marked. Parameters are as before the to be inserted classifier cl, the action set $[A]$ classifier cl was generated from, and the current population $[P]$.

ADD GA CLASSIFIER(cl, [A], [P]):

```
oldCl ← empty
for all classifiers c in [A]
   if(c IS SUBSUMER of cl)
      if(oldCl is empty or c.C is more general than oldCl.C)
         oldCl ← c
if(oldCl is empty)
   for all classifiers c in [A]
      if(c is equal to cl in condition and effect part)
         oldCl ← c
if(oldCl is empty)
   insert cl in [A] and [P]
else
   if(oldCl is not marked)
      oldCl.num++
   discard classifier cl
```

3.9 Subsumption

ACS2 looks for subsuming classifiers in the GA application as well as in the ALP application. For a classifier cl_{sub} to subsume another classifier cl_{tos}, the subsumer needs to be experienced, reliable, and not marked. Moreover, the subsumer's condition part needs to be syntactically more general and the effect part needs to be identical. Note again, an identical action check is not necessary since both classifiers occupy the same action set. The procedure returns if classifier cl_{tos} is subsumed by cl_{sub} but does not apply any consequent parameter changes.

IS SUBSUMER(cl_{sub}, cl_{tos}):

```
if(cl_sub.exp > θ_exp and cl_sub.q > θ_r and cl_sub.M is empty)
   if (the number of # in cl_sub.C ≤ the number of # in cl_tos.C)
      if(cl_sub.E is equal to cl_tos.E)
         return true
return false
```

4 Summary

This chapter gave a precise overview over the ACS2 system. Interaction, knowledge representation, and parameter identification should serve as a basic reference book when implementing a new problem and applying ACS2 to it. The algorithmic description revealed all processes inside ACS2 and should serve as a helpful guide to program an own version of ACS2 or develop an enhanced anticipatory learning classifier system out of the ACS2 framework. The description did not include any implementation details so that the system should be programmable in any programming language with the help of this description.

Acknowledgments We would like to thank the Department of Cognitive Psychology at the University of Würzburg for their support. The work was sponsored by the German Research Foundation DFG.

References

Butz, M. V. (2001a). *Anticipatory learning classifier systems*. Genetic Algorithms and Evolutionary Computation. Boston, MA: Kluwer Academic Publishers.

Butz, M. V. (2001b). *An implementation of the anticipatory classifier system ACS2 in C++* (IlliGAL report 2001026). University of Illinois at Urbana-Champaign: Illinois Genetic Algorithms Laboratory. http://www-illigal.ge.uiuc.edu/sourcecd.html.

Butz, M. V. (2002). Biasing exploration in an anticipatory learning classifier system. In Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds.), *Proceedings of the Fourth International Workshop on Learning Classifier Systems (IWLCS-2001)* Berlin Heidelberg: Springer-Verlag.

Butz, M. V., Goldberg, D. E., & Stolzmann, W. (2000). Introducing a genetic generalization pressure to the anticipatory classifier system: Part 1 - theoretical approach. In Whitely, D., Goldberg, D. E., Cantu-Paz, E., Spector, L., Parmee, I., & Beyer, H.-G. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)* pp. 34–41. San Francisco, CA: Morgan Kaufmann.

Butz, M. V., Goldberg, D. E., & Stolzmann, W. (2001). Probability-enhanced predictions in the anticipatory classifier system. In Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds.), *Advances in Learning Classifier Systems, LNAI 1996* pp. 37–51. Berlin Heidelberg: Springer-Verlag.

Dorigo, M., & Colombetti, M. (1997). *Robot Shaping, an experiment in behavior engineering*. Intelligent Robotics and Autonomous Agents. Cambridge, MA: MIT Press.

Lanzi, P. L. (2000). *Learning classifier systems from a reinforcement learning perspective* (Technical Report 00-03). Dipartimento di Elettronica e Informazione, Politecnico di Milano.

Stolzmann, W. (1997). *Antizipative Classifier Systems [Anticipatory classifier systems]*. Aachen, Germany: Shaker Verlag.

Stolzmann, W. (1998). Anticipatory classifier systems. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D., Grazon, M., Goldberg, D., Iba, H., & Riolo, R. (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference* pp. 658–664. San Francisco, CA: Morgan Kaufmann.

Stolzmann, W., Butz, M. V., Hoffmann, J., & Goldberg, D. E. (2000). First cognitive capabilities in the anticipatory classifier system. In Meyer, J.-A., Berthoz, A., Floreano, D., Roitblat, H., & Wilson, S. W. (Eds.), *From Animals to Animats 6: Proceedings of the Sixth International Conference* *on Simulation of Adaptive Behavior* pp. 287–296. Cambridge, MA: MIT Press.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* Doctoral dissertation, King's College, Cambridge, UK.