A Guide to the Implementation Details of CLARION

by

Xi Zhang

June 2004

List of Figures

Contents

Chapter 1

General Implementation structure

Before describing the implementation of ACS, it is necessary to talk about the overall implementation structure of CLARION. We embed a CLARION model as a "brain" in each subject in an experiment to do learning and reasoning, that is, CLARION is in charge of overall learning and reasoning process. Current implementation of CLARION is based on Object-Oriented Design and Object-Oriented Programming and involves the concepts and technologies in Object-oriented software engineering. Currently CLARION system is developed by Java Programming Language. Each component in CLARION is encoded as a class with the same name as described in CLARION. The overall software (implementation) structure is hierarchical according to what CLARION model specifies. See the figure 1 on overall implementation structure.

For a system to work properly, it is necessary to start it properly. To start CLARION properly, it is necessary to initialize all of the parameters and options described in CLARION model properly. To do it, there are two ways: one is through the GUI (Graphical User Interface) part of CLARION software system to manually set all of the necessary options and parameters properly. Another, more efficient way, is to load an already existing configuration of the options and parameters as current

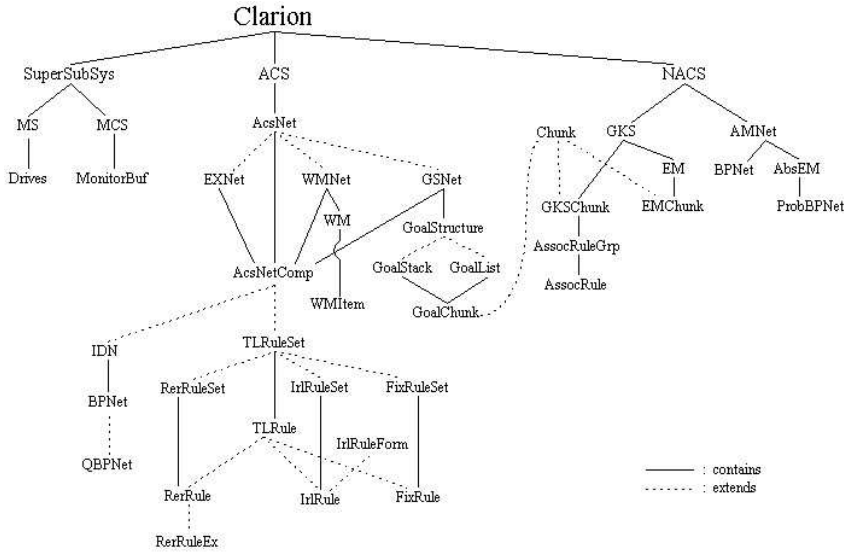

Figure 1.1: Overall implementation structure

system configuration and if necessary, do some additional minimal changes using GUI. When all of the parameters in CLARION are initialized properly, CLARION is ready to work. An existing configuration is a system default configuration or a previous manual configuration using GUI. Each specific task for simulation has a group of existing configurations those are located in the specific task configuration subdirectory of the root configuration subdirectory under the system root directory. They are organized hierarchically corresponding to the hierarchy of CLARION model.

1.1 General data structures of CLARION

1.1.1 The class *Clarion*

This class is on the top of the implementation structure and is the class to manage and control the overall learning and reasoning process described in CLARION model.

1.1.2 The class *Global*

CLARION is an integrative system which has many interconnected components. Each interconnected component needs communicate, coordinate and interact with other components. To achieve this, a convenient and efficient way is to construct a class *Global* to manage all of the parameters or variables (data structures) globally used within the whole CLARION system. Each object of *Global* class is associated with an object of a task agent class *TaskAgent* and so, its initialization and instantiation is inside the *TaskAgent* object. The initialization and instantiation of *Global* class is an important part of the whole system initialization. The class *Global* can be accessed by all of the classes implementing the components in CLARION model to get the required information.

1.1.3 The group of classes for task simulator

In the implementation of CLARION model, we assume the model is a part of an task agent in charge of the learning and reasoning processes. To simulate a specific human task, we need simulate the behaviors of each agent in this specific human task. To do it, we need not only simulate the learning and reasoning processes, but also need simulate the task-specific processes. In current implementation, the task-specific processes is implemented by a group of classes named task simulator. The task simulator contains the following three classes:

1. *Task*

   This class is the base class for all of the classes of concrete human tasks to be simulated. The purpose of the *Task* class, roughly, is to act as the experimenter in an empirical study: it defines the input the agents receive at each step, collects their output at each step, and controls any interaction

between the agents. To simulate a concrete human task, user codes are needed to extend this basic class.

2. *TaskAgent*

   This class is the base class for all of the classes of concrete human subjects to be simulated. *TaskAgent*, roughly, describes the generic behaviors of a subject in an experiment. Within this class, many methods (routines) are declared to simulate a subject's behaviors. To simulate the behaviors of a subject in an experiment, user codes are needed to extend this basic class.

3. *TaskClarion*

   This class is the base class for all of the classes implementing the task-specific options, parameters and processes described in CLARION model. This class defines the task-specific aspects of the Clarion model's behavior. Also, this class needs be extended for specific simulation.

From another perspective, it is obvious that CLARION cannot and also need not handle everything in details like any other systems since each task has its own specific stuff. In order for CLARION to simulate as many kinds of tasks as possible, in the implementation of CLARION model, we need a mechanism (the above called task simulator) something like an interface between CLARION and the task that can allow CLARION call the user specified routines automatically and correctly. Now, we construct the class *TaskClarion* to implement it. In this class, it declares and defines all of the required task-specific routines by CLARION. All of the routines are default set and in order to make CLARION simulate a task correctly, user should override the routines if a new version is available. If there is no new version for a routine, CLARION will use the default routine.

1.1.4 Input and output format

In CLARION, input to ACS (current state buffer) is a set of dimensions each with a set of allowable values, We use 2- dimensional array to represent it.

The output from ACS (action recommendations) is more complicate since in CLARION model, the overall reasoning process is under the control of ACS. So, the implementation of ACS actions are very important. In current implementation of ACS, the ACS actions have a unified format. It includes all action dimensions as output of all action types and allows user selection through GUI for each IDN/rule group.

- Four different types of actions:

  1. external actions
  2. goal structure actions
  3. working memory actions
  4. NACS control actions (part of external action)

- Unified action format

  currently, we use a 3-dimensional array to represent the unified action format.

  1. dimension 1 : indicates action type: External, GS, WM or NACS control.
  2. dimension 2 : indicates a dimension index on a specific action type.
  3. dimension 3 : stores currently activated values in the specific dimension.

- General format of each action type

  Each action may have 3 parts:

1. the TYPE dimension

   indicates the action is an external action, GS action, WM action or NACS control action.

   element 0 : external action

   element 1 : GS action

   element 2 : WM action

   element 3 : NACS control action

2. the dimension(s) for action specification

   defines what the action will do.

3. the dimension(s) for parameters

   some actions may use parameters when they are performed.

   The General format of each parameter dimension

   a "signal value" is added into the end of each of the parameter dimensions to indicate if the dimension allows multiple active values or not at current step. (in the following action formats, such "signal value"s are not shown.)

- Concrete action format

  1. External actions

     Since this kind of action is task-specific, the format of external actions should be specified by user. User should specify all of the dimensions in the action format by giving each dimension the name, the number of values.

  2. Goal structure actions

This kind of action is for internal use, that is, for CLARION system use. the format is the following:

(a) dimension 0 : action type set the element indicating GS action.

(b) dimension 1 : action specification

element 0 : DO NOTHING

element 1 : SET

element 2 : RESET

(c) dimension 2 : goal dimension

element 0 : goal 0 is currently active

. . . .

element n-1 : goal n-1 is currently active

(n is the number of goals user defines)

(d) other parameter dimensions

these dimensions are for the goal parameters along with the currently active goal. Since the goal parameters are task-specific, so, these dimensions should be defined by user. They are maybe empty sometimes.

3. Working memory actions

Also, this kind of action is for internal use. The format is the following:

(a) dimension 0 : action type

set the element indicating WM action.

(b) dimension 1 : action specification

element 0 : DO NOTHING

element 1 : SET

element 2 : RESET (one slot)

element 3 : RESET ALL

(c) dimension 2 : the set of WM slots for storing data.

element 0 : WM slot 0 is involved

element 1 : WM slot 1 is involved

. . . .

element i : WM slot i is involved

. . . .

element n-1 : WM slot n-1 is involved

(n is the size of WM)

Actually, the actions: DO NOTHING and RESET ALL needn't parameters. Only the actions SET and RESET use the dimension 4 as parameters.

1.1.5 The class RuleAttributes

Since CLARION model defines many constants and measures relevant to rule knowledge such as rule BLA, rule utility, rule positivity and negativity and their relevant constants, some constants relevant to response time. So, obviously, we need construct a class to contain all of these constants and the relevant routines to calculate these measures. In current implementation, the class *RuleAttributes* is used to achieve this. Each *RuleAttributes* object is associated with a specific rule and instantiated inside that rule.

In the class *RuleAttributes*, it declares all of constants and variables required to calculate the rule measures. Also, it declares methods to implement any possible processes of calculation since each rule measure can be calculated in different ways

depending on the configuration parameters and options. Among implementations of these measures, the calculation of BLA needs a list to record all of the times when the rule is used since as described in previous chapter, the formula for calculating BLA needs the information of its past usage. Besides the methods for calculation, it also declares routines for update of these measures at each step which is one part of rule update.

When the CLARION software is running, at each step, the associated *RuleAttributes* object of a rule is updated dynamically showing the dynamic property of the rule.

1.2 Algorithm for Overall reasoning process in CLARION

Currently in CLARION, reasoning in NACS is completely under the control of the ACS and Action-Directed Reasoning is adopted as the overall reasoning process. Following is the algorithm:

1. Observe the current state x and store it in the current state buffer.
2. Pass the state information from current state buffer to ACS.
3. Check all of the eligible/applicable networks in ACS for current state.
4. Pass the state information to all of the eligible networks.
5. In each eligible network, do the followings:
   (a) pass the state information to every component of a network: IDN, RER, IRL and FR.

(b) In IDN, compute the "value" of each of the possible actions (a_i 's) in the state x: $Q(x, a_1), Q(x, a_2), \ldots, Q(x, a_n)$.

(c) In ARS, for each component: RER, IRL and FR, find out all the recommended actions $(b_1, b_2, \ldots, b_m)$.

(d) Choose an appropriate action a through integrating the recommendations of a_i's and b_j's

(e) Perform the action a.

(f) If the action chosen involves reasoning in the NACS, reasoning takes place in the NACS, based on the information provided by that adopted action, either a specific set of dimensional values or a set of chunks. Repeat the following steps:

   i. As directed by the ACS action, perform reasoning in the GKS using associative rules (if any). A reasoning method may be specified by the action too. Reasoning continues either until there is no more new conclusion that can be drawn, or until a fix time limit is reached. The time limit may be specified by the adopted action as well.

   ii. As directed by the ACS action, perform in the AMNs associative mapping. Furthermore, optionally, the AMNs may perform further associative mapping from the result of previous associative mapping (one step at a time).

   iii. As directed by the ACS action, combine the results of associative mapping at the bottom level (which is sent up to the top level), with the results of the top-level inference.

(g) As determined by the adopted action of the ACS, output the "filtered"

results, (1) in the form of a set of chunks (that are compatible with the final result), or (2) in the form of one chunk (selected through a Boltzmann distribution based on chunk strengths).

(h) After performing the action, observe the next state y and (possibly) the reinforcement r.

i. Update the IDNs in accordance with QBP or BP algorithm.

ii. Update each component of the ARS: RER, IRL and FR.

iii. If the NACS was used at this step, update the GKS through extracting chunks and associative rules corresponding to the results from the AMNs, if any mapping occurred in the AMN and if there are no corresponding rules in the GKS.

iv. If the NACS was used at this step, optionally, update the AMNs, through presenting associations to the network (including the auto-association in the AAM, and the hetero- association in the HAM).

6. Go back to Step 1.

Chapter 2

Implementation of ACS

2.1 ACS implementation class structure

In ACS, there maybe exist three types of networks: External networks, one GS network (optional) and one WM network (optional). Network here means IDN and its corresponding rule group together that constitute a decision network. Since each network has the similar function: given current state information and select an action to perform. So we construct a generic class *AcsNet* for an ACS network and define the necessary operations on it. Each class for concrete network *EXNet, GSNet, WMNet* can extend this generic class and define new operations specific to its network type. Since both GS and WM network are special : GS network to manage GS (Goal structure) and WM network to manage WM (Working Memory), so we need embed a class *GoalStructure* and a class *WM* into the corresponding network respectively.

Within each network of ACS, there are four components representing four knowledge types: IDN, RER, IRL and FR. Since these four components have the similar function: do an action decision making given current state information. We construct a class *AcsNetComp* to represent a generic knowledge type and define a set of basic operations in the class. Based on this generic class, we further construct

two more specific classes *IDN, TLRuleSet*: one representing IDN and one representing the generic action-rule-set class and adding new operations specific on the two classes. Since each action-rule-set (RERs, IRLs, FRs) is a set of action rules (RER, IRL, FR) and each type of action rule (RER, IRL and FR) has similarity: do an action recommendation process, so we can construct a generic action-rule class *TLRule*. The generic action-rule-set class contains a set of generic action-rules.

In each IDN, since CLARION uses QBP algorithm, so in the class *IDN*, a class *QBPNet* extending the class *BPNet* (implementing backpropagation algorithm) is contained.

In each rule group corresponding to an IDN, based on the generic action-rule-set class, three concrete action-rule-set classes *RerRuleSet, IrlRuleSet, FixRuleSet* are constructed to represent the three concrete action-rule-sets: (RERs, IRLs, FRs) respectively. Based on the generic action-rule class, we can construct the three concrete action-rule classes *RerRule, IrlRule, FixRule* corresponds to RER, IRL or FR respectively.

2.2 ARS generic data structure

In the generic rule class, to represent the rule condition according to CLARION description, we use 2-dimensional array. To represent the rule conclusion (action), we still use 2-dimensional array since the ACS action is multiple dimensional and each dimension may allow multiple active values.

Since the measures defined in CLARION are relevant to rules such as rule support, rule utility, rule BLA, positivity and negativity are applied to the action rules in ACS, so we define a class *RuleAttributes* associated with a specific rule to contain these rule attributes.

2.3 implementation algorithm for eligibility

From the overall reasoning algorithm, we can see that it is important to check the eligible/applicable networks at current step. An eligibility condition may be specified for each external action network, so that not all of these networks may be applicable at a particular step. Ideally, only one external action network is applicable at each step. Inapplicable networks are simply ignored. The eligibility condition of an external action network may be specified based on goals: Each subtask may have a different goal, and the current goal indicates the eligibility of different networks. The eligibility condition may also be specified based on more elaborate information (such as the current state along with the current goal, or certain patterns in the past state sequences). In current implementation of eligibility, there is a class *EligibilityCheck* in charge of it specifically.

To implement eligibility, we assume each network has an uniform eligible condition for current input to match. Such eligible condition has the format :

- if (C_1 *or* $C_2 \ldots$ *or* C_n) then use net j,
  each C_i should be exclusive of each other and the relation between each C_i is "or".

- C_i has the format (c_{i_1} *and* $c_{i_2} \ldots$ *and* c_{i_n}).
  the relation between each c_{i_j} is "and".

- the above formats should be satisfied by user input before doing eligibility checking.

2.4 implementation algorithm for reinforcement functions

In CLARION, there are three types of reinforcement functions for external actions, goal setting actions and working memory actions. In implementation of these functions, we have to consider several cases:

- only one reinforcement function is given that encompasses all three functions. In this case, the output of this function is given not only mandatorily to the external network but all three types of networks (goal action, working memory action, and external action).

- separate reinforcement functions are given for external actions, goal actions, and working memory actions, then the corresponding networks use these reinforcement functions respectively.

- separate reinforcement functions may be provided to different networks for external actions, if multiple external action networks are used.

Obviously, the reinforcement functions is task-specific, CLARION knows nothing about reinforcement measure in a specific task and can only offer a default function. If reinforcement is important to a task, user should give concrete reinforcement functions to override the default.

2.5 IDN implementation algorithm

In IDN, a class *QBPNet* is contained to implement the QBP algorithm. See section ACS in chapter 2 on the details of QBP algorithm. In simulating real tasks, when simplified Q-Learning is applied, there exists a special case of it: step-wise predic-

tion. In the case of step-wise prediction, we implement it as following: $r = 1$ if the chosen a is the correct prediction; $r = 0$ if the chosen a is not the correct prediction.

2.6 Implementation of RER

2.6.1 RER data structures

Different from other types of action rules, RER rule is more complex since its condition is changed dynamically. The condition change depends on a generalization process, a specialization process or a rule merging process (merge the most similar rules) happens at current step.

There are two difficulties in implementing RER: one is how to maintain the dynamic RER rule structure correctly and the other is how to record the PM (Positive Match) and NM (Negative Match) of a RER rule correctly. To overcome the first difficulty, we use a special class named *HashSet* in java class package to maintain the dynamic RER rule structure since *HashSet* satisfies the property RER rule structure has: no duplicate rules are allowed in the active rules and in the inactive rules. To overcome the second, we use a 4-dimensional array $A(i, j, k, m)$ to record the statistics of PM and NM. $A(i, j)$ indicates a condition: jth allowable value in the ith dimension. k indicates with or without the condition value $A(i, j)$ and m indicates the statistics for PM or for NM. This array can dynamically record the statistics of PM and NM correctly since this data structure considers every possibility: recording statistics for the rule condition plus/minus every possible value in every input dimension and at each step, this array will be updated.

2.6.2 RER implementation algorithm

1. Update the rule statistics (to be explained later).

2. Check the current criterion for rule extraction, generalization, and specialization:

   (a) If the result is successful according to the current rule extraction criterion, and there is no rule matching the current state and action, then perform extraction of a new rule: *condition* $\rightarrow$ *action*. Add the extracted rule to the ARS.

   (b) If the result is unsuccessful according to the current specialization criterion, revise all the rules matching the current state and action through specialization:

      i. Remove the rules from the rule store.

      ii. Add the revised (specialized) versions of these rules into the rule store.

   (c) If the result is successful according to the current generalization criterion, then generalize the rules matching the current state and action through generalization:

      i. Remove these rules from the rule store.

      ii. Add the generalized versions of these rules to the rule store.

3. Merge all of the possible rules if necessary.

Some definitions and measures for RER

When implementing RER, following definitions and measures are necessary:

1. $PM_a(C)$ (i.e.,Positive Match) equals the number of times that a state matches condition C, action a is performed, and the result is positive;

2. $NM_a(C)$ (i.e.,Negative Match) equals the number of times that a state matches condition C, action a is performed, and the result is negative.

3. The default criterion of positivity or negativity for updating PM and NM is:

$$\gamma \max_b Q(y, b) + r - Q(x, a) > threshold_{RER} \tag{2.1}$$

which indicates whether or not action a (chosen according to a rule) is reasonably good [**?**]. Alternative criteria are also possible. For example, when immediate feedback is given, positivity may be determined by the immediate feedback: If $r > threshold_{RER}$, then it is positive; otherwise, it is negative.

4. The default measure for Information gain measure (IG) may be calculated:

$$IG(A, B) = \log_2 \frac{PM_a(A) + c_1}{PM_a(A) + NM_a(A) + c_2} - \frac{PM_a(B) + c_1}{PM_a(B) + NM_a(B) + c_2} \tag{2.2}$$

where A and B are two different rule conditions that lead to the same action a, and c_1 and c_2 are two constants representing the prior (the default values are $c_1 = 1$; $c_2 = 2$). Essentially, the measure compares the percentages of positive matches under different conditions A and B (with the Laplace estimator). If A can improve the percentage to a certain degree over B, then A is considered better than B.

Algorithms for the three processes in RER

1. Extraction: If the current step is positive (according to the current positivity criterion) and if there is no rule that covers this step in the top level (matching both the state and the action), set up a rule $C \rightarrow a$, where C specifies the values of all the dimensions exactly as in the current state x and a is the action performed at the current step.

2. Generalization: If $IG(C, all) > threshold_1$ and $\max_{C'} IG(C', C) \geq 0$, where C is the current condition of a rule (matching the current state and action), *all* refers to the corresponding match-all rule (with the same action as specified by the original rule but with a condition that matches any state), and C' is a modified condition such that $C' = C$ plus one value (i.e., C' has one more value in one of the input dimensions) [that is, if the current rule is successful and a generalized condition is potentially better], then set $C" = argmax_{C'} IG(C', C)$ as the new (generalized) condition of the rule. Reset all the rule statistics. Any rule covered by the generalized rule will be placed in its children list.

3. Specialization: $IG(C, all) < threshold_2$ and $\max_{C'} IG(C', C) > 0$, where C is the current condition of a rule (matching the current state and the current action), *all* refers to the same as in generalization, and C' is a modified condition such that $C' = C$ minus one value (i.e., C' has one less value in one of the input dimensions) [that is, if the current rule is unsuccessful, but a specialized condition is better], then set $C" = \arg\max_{C'} IG(C', C)$ as the new (specialized) condition of the rule. Reset all the rule statistics. Restore those rules in the children list of the original rule that are not covered by the specialized rule and the other existing rules.

Let us discuss the details of the operations used in the above algorithm and criteria measuring whether a result is successful or not. At each step, we examine the following information: (x, y, r, a), where x is the state before action a is performed, y is the new state after an action a is performed, and r is the reinforcement received after action a. Based on that, we update (in Step 1 of the above algorithm) $PM_a(C)$ and $NM_a(C)$ (see section ACS chapter 2 on their definitions) for each rule condition and each of its variations (e.g., resulting from the rule condition plus/minus one

possible value in one of the input dimensions), denoted as C, with regard to the action a performed. Positivity or negativity (for updating PM and NM) may be determined based on a certain criterion. See section ACS, chapter 2 on the default criterion.

Each statistic is updated with the following formulas:

- $PM := PM + 1$ when the positivity criterion is met;
- $NM := NM + 1$ when the positivity criterion is not met.
- At the end of each episode, they are discounted: $PM := PM * 0.90$ and $NM := NM * 0 : 90$. The results are time-weighted statistics, which are useful in non-stationary situations.

Current CLARION software has implemented several versions of rule generalization/ specialization. The default version is that any value from any dimension may be chosen to add or delete. It is also the only way when these dimensions are nominal. On the other hand, if a dimension is ordinal, there are the following options: we can either add/delete an arbitrary value, which results in non-contiguous value ranges for some input dimensions in a rule condition, or we make sure that we have contiguous value ranges by adding/deleting values only at the two ends of the current ranges.

To implement generalization process, there exists another option: when we generalize, we can either add one value at a time to a dimension, or add all the values of that dimension. In case the "all values" option is adopted, the IG calculation as specified before needs to be altered: It should be done with respect to $C' = C$ plus all values. Similarly, when we specialize, we can remove one value at a time from a dimension, or we can remove all but the last value (if there are two or more

values) from a dimension. In case we use the "all values" option, the IG calculation specified earlier can be similarly altered, with respect to $C' = C$ minus all values except one (if there are two or more values). Beside the default IG measure, one may specify alternative IG measures. Similarly, one may also specify alternative positivity criteria used there.

In the current implementation of Clarion, there are two versions of extraction process. As evident in the discussion above, this (default) extraction method is suitable for specific-to-general rule learning ($[\mathbf{?}, \mathbf{?}]$). That is, the agent extracts a most specific rule and then tries to generalize it later. An alternative, for general-to-specific rule learning, is to extract a rule with a small condition involving, for example, only one input dimension (or only a few), and then to try to specialize it. In this case, to extract a rule, we do the following:

> Extraction: If the current step is positive (according to the current extraction criterion), and if there is no rule that covers this step in the top level (matching the state and the action), set up a rule $C \rightarrow a$, where C specifies a (randomly) chosen set of values of a (randomly) chosen input dimension consistent with the current state x and a is the action performed at the current step.

Algorithm for RER Rule Merge process

We may merge rules: If either rule extraction, generalization, or specialization has been performed, check to see if the conditions of any two rules are close enough and thus if the two rules may be combined: If one rule is covered completely by another, put it on the children list of the other. If one rule is covered by another except for one dimension, produce a new rule that covers both. One may choose

how frequently the merge operation is performed.

2.7 Implementation of IRL

In the current implementation of Clarion, one may specify several subsets of rules. Each subset is specified through a rule template, and ranges of its parameter values. To implement this, we need construct a class *IrlRuleForm* for IRL rule template and define operations specific to IRL rule. These rule subsets will be tested in the order specified. That is, if one subset of rules (generated from one template) fails the test and consequently all the rules in the subset are deleted (one by one), then the next subset of the rule (i.e., the next template) will be tested, and so on.

criterions used in IRL

When implementing IRL, the following criterions are used:

1. one possible positivity measure is:

$$\gamma \max_b Q(y,b) + r - Q(x,a) > threshold_{IRL} \tag{2.3}$$

2. one possible IG measure for IRL rule testing is (with respect to any particular domain): "If $IG(C) = \log_2 \frac{PM(C)+c_5}{PM(C)+NM(C)+c_6} < threshold_4$, we delete the rule C. "

Some processes in IRL

When appropriate, generalization and specialization may also be performed based on the IG measure as before. Specialization consists of removing allowable values from the condition of a rule (the same way as with RER rules). Generalization then consists of adding allowable values to the condition of a rule (again, the same way as with RER rules).

Often, an initial IRL rule (specified a priori to be tested through experience) does not involve values of input dimensions in its condition (see, e.g., the rules used in simulating process control tasks). In that case, we consider the rule condition consists of all the values in all the input dimensions (at its most general form), thus, no generalization may be performed on such a rule initially, but specialization can be performed. Deletion can also occur when specialization leads to an overly specialized condition that has no possibility of matching any input.

2.8 Implementation of FR

Actually, FR has two versions: one is fixed condition and conclusion, the other is a fixed process to deal with current state. For the first version, we just do the normal action decision making: compare current state with the rule condition and get the fixed action as recommended action from this rule if the rule condition is matched, otherwise get empty action. For the second version, we need construct a generic process for dealing with current state and then in a specific task, user can set up their FRs by extending this generic process if there are FRs of this version.

2.9 Implementation of Goal structure

Since there are two types of Goal Structure defined in CLARION: goal stack and goal list. So, first we construct a generic class *GoalStructure* for goal structure and define some operations on it. The two concrete classes of goal structure: *GoalStack* and *GoalList* can extend the generic class and define new operations. The goal item is represented by the class *GoalChunk* which extends from the more general class *Chunk*, that means, goal item is a special type of chunk.

2.10 Implementation of Working Memory

In implementation of Working memory which consists of a specified number of working memory items, we use the class *WMItem* to represent the items. Also, the class extends from the class *Chunk*. So, essentially, each item is represented a special type of chunk. For working memory part of current state, only items with BLAs above a threshold ($threshold_{WM}$) can be used as part of the current state with strength of 1.

To simulate some real tasks, there needs a few special working memory items, namely flags that do not correspond to any sensory input dimensions and can be either on or off. So, we need implement these special items. We add two additional working memory actions for each used: set $flag_i$ and reset $flag_i$, where i is a flag number and setting a flag turns it on and resetting a flag turns it off. In addition, there is $reset - all - flag$. Each action is represented by an individual node and together they constitute a flag action dimension.

2.11 Coordinations

The coordination algorithm is as follows:

- First select one knowledge type (by using variable or fixed stochastic selection), or combine the two levels (by using a more complex method, such as weighted-sums).
- Using the selected knowledge type or using the combined values, if there are multiple possible external actions, choose one (or choose all, as described before).
- Then (with the selected knowledge type or the combined values), select among

the chosen external action, the chosen goal action, or the chosen WM action: Use one of the following two methods:

1. Select one type of action, according to a pre-specified probability distribution. (This method includes "external action first" as a special case.) If the selected action type is not available or indicates do-nothing, select one of the remaining two action types stochastically. If one of the two remaining actions is not available or indicates do-nothing, then select the sole available action type. If none of the action type is available or indicates anything other than do-nothing, do nothing.
2. Perform simultaneously all the chosen actions of each of the three action types. (If one action type is not available or is do-nothing, ignore it.)

Chapter 3

Implementation of NACS

In CLARION model, the overall reasoning process is under the control of ACS. For example, how the reasoning process goes on in NACS is actually decided by ACS actions. So, besides the external, goal structure and working memory actions, the another action type: NACS control action is designed for ACS to control NACS learning and process process. In essence, the NACS control action is a special part of external action.

3.1 NACS control action format

The format is the following:

1. dimension for action type

   dim_0: action type

   set the element indicating NACS control action.

2. subaction dimensions

   In this part, each dimension is a subaction. So, a control action actually is composed of a series of subactions.

(a) dim_1: activate NACS or report

element 0 - not activate NACS and not report

element 1 - activate NACS

element 2 - report to external destination from the retrieval buffer of NACS

(b) dim_2: choose level

element 0 - choose GKS

element 1 - choose AMNs

element 2 - choose both

(c) dim_3: encode externally given knowledge

element 0 - do nothing

element 1 - encodes as associative rules

(d) dim_4: assimilate explicit knowledge

element 0 - do nothing

element 1 - assimilate

(e) dim_5: do inference and retrieval

element 0 - do nothing

element 1 - retrieve one chunk from NACS above a threshold by Boltzmann distribution

element 2 - retrieve all chunks from NACS above a threshold

(f) dim_6: set chunk strength

element 0 - do nothing

element 1 - set strength

3. parameter dimensions.

   This part is relevant to subaction part since some subactions need parameters to help the control action work properly.

   (a) dim_7: format of input to NACS (using input to ACS)

   the length of this dimension = number of dimensions of input to ACS. if dim7(i) ¿ 0.5, then the ith dimension of input to ACS is input to NACS

   (b) dim_8: format of input to NACS (using output from ACS)

   the length of this dimension = number of dimensions of output from ACS. if dim8(i) ¿ 0.5, then the ith dimension of output from ACS is input to NACS

   (c) dim_9: format of output from NACS (using input to ACS)

   the length of this dimension = number of dimensions of input to ACS. if dim9(i) ¿ 0.5, then the ith dimension of input to ACS is output from NACS.

   (d) dim_{10}: format of output from NACS (using output from ACS)

   the length of this dimension = number of dimensions of output from ACS. if dim10(i) ¿ 0.5, then the ith dimension of output from ACS is output from NACS.

   (e) dim_{11}: GKS reasoning method

   element 0 - forward reasoning

   element 1 - forward chaining with similarity-based reasoning

   (f) dim_{12}: number of iterations of GKS reasoning

   (let maximum number of iterations be 10)

element 0 - number of iterations is 1.

element 1 - number of iterations is 2.

. . .

element 9 - number of iterations is 10.

element 10 - number of iterations is unlimited.

(g) dim_{13}: number of AMN passes for each iteration

(let maximum number of passes be 10)

element 0 - number of passes is 1.

element 1 - number of passes is 2.

. . .

element 9 - number of passes is 10.

(h) dim_{14}: involved AMNs

element 0 - AMN 0 is involved

element 1 - AMN 1 is involved

. . .

element n-1 - AMN n-1 is involved

(n is the number of AMN)

(i) dim_{15}: type of chunk to store into retrieval buffer

element 0 - all chunk types

element 1 - state related chunks

element 2 - state related chunks not involved in the input to NACS.

element 3 - action related chunks

element 4 - action related chunks not involved in the input to NACS

(j) dim_{16}: the chunk to set strength of

element 0 - chunk 0 in GKS

element 1 - chunk 1 in GKS

. . .

element n-1 ?chunk n-1 in GKS

(n is the chunk number in GKS)

(k) dim_{17}: strength level to set at

element 0 - 0.0

element 1 - 0.1

. .

element 10 - 1.0

We note that in the above action format, there exist two kind of inter-dependency : one within subation part and one between subaction and parameter part. For the first case, for example, if we choose dim_1 = "report to external destination from the retrieval buffer of NACS", all of the other dimensions will be disabled since they are irrelevant to this subaction. For the second case, for example, if dim_6 = "do nothing", its relevant parameters dim_{16} and dim_{17} will be disabled. Actually, all of the above inter-dependencies are implemented as routines in current CLARION software.

The above generic NACS control action format contains all of the major stuff described in NACS subsystem of CLARION model. The action-directed reasoning process in which NACS is under the complete control of the ACS can be implemented completely by configuring the above NACS control action. For example, an action command by the ACS may specify performing reasoning in the NACS and

obtaining reasoning results from the NACS retrieval buffer before taking another action. Actually, to configure this action using the above action format, this action is achieved by two steps. First is the inference step and the second is report step.

For the inference step:

- dim_0 : action type

  set the element indicating NACS control action.

- dim_1: activate NACS or report

  set element 1 - activate NACS

- dim_2: choose level

  set element 2 - choose both

- dim_3: encode externally given knowledge

  set element 0 - do nothing

- dim_4: assimilate explicit knowledge

  set element 1 - assimilate

- dim_5: do inference and retrieval

  set element 1 - retrieve one chunk from NACS above a threshold by Boltzmann distribution

- dim_6: set chunk strength

  set element 0 - do nothing

- dim_{11}: GKS reasoning method

  set element 0 - forward reasoning

- dim_{12}: number of iterations of GKS reasoning

  set element 10 - number of iterations is unlimited.

- dim_{13}: number of AMN passes for each iteration

  set element 0 - number of passes is 1.

- dim_{14}: involved AMNs

  set element 0 - AMN 0 is involved

- dim_{15}: type of chunk to store into retrieval buffer

  set element 0 - all chunk types

For the report step:

- dim_0 : action type

  set the element indicating NACS control action.

- dim_1: activate NACS or report

  set element 2 - report to external destination from the retrieval buffer of NACS

- dim_2 to dim_{17}

3.2 General data structures in NACS

3.2.1 The base classes *Feature* and *Chunk*

Dimension and chunk are basic concepts in CLARION, so the class *Feature* is to represent the term dimension. Since many different kinds of chunks described in CLARION, so we need construct a common base class *Chunk* for concrete chunk

classes to extend. The class *Chunk* is to represent the concept Chunk in CLARION. These two classes are basic classes in the CLARION software.

In the common base chunk class, as described in CLARION model, chunk actually is a set of dimension-value pairs, so, in class *Chunk*, the most important variable (named *featList*) is to represent a set of features (dimensions) represented by a set of *Feature* objects.

Also, in the base chunk class, we define some variables used to represent the commonality of all kinds of chunks : BLA(base-level activation). Based on the generic class *Chunk*, there are several types of specific chunks, such as goal chunks, WM items, GKS chunks, all those can extend the basic class *Chunk* and define their own attributes within their own class.

3.2.2 The concrete class *GKSChunk*

The class *GKSChunk* is used to represent the chunks in GKS. This class extends the base class *Chunk*. And within this class, it declares an associative rule group pointing to this *GKSChunk* object. And it declares some variables relevant to chunk strength. Besides the variables, it also implements the methods how to activate the chunk using the given reasoning method, how to calculate the chunk strength using the measure described in CLARION model, and how to update the attributes of the chunk such as BLA and the associated rule group.

3.2.3 The classes *AssocRuleGrp* and *AssocRule*

Both the class *AssocRule* and the class *AssocRule* are used to implement the concept of associative rules in GKS. In GKS, the associative rules are described as the associations between chunks. To implement the associative rules in an easier way, we let the associative rules as attributes of the associated chunks the rules point

to. In this way, we needn't maintain two kinds of data structures for Chunks and associative rules independently and makes update of chunks and their relevant associative rules much easier. That is, it makes implementation structure simple and makes maintenance easier.

The class *AssocRule* is to represent an associative rule. In the class, it declares the condition of this rule, represented by a one-dimensional array, which is a set of chunks. And it declares the conclusion of the rule which actually is the pointing-to chunk. As similar to the classes of ACS rules, it also declares the relevant variables for rule support and one *RuleAttributes* object to represent the rule attributes such as BLA and rule utility;

the class *AssocRuleGrp* is to represent a group of associative rules the pointing-to chunk has. In this class, the associative rule group is implemented by a *LinkedList* object, a dynamic data structure, which is easy to insert and delete a rule quickly and also space efficient.

3.2.4 The class *NACS*

The class *NACS* is to implement the functionalities of NACS described in CLARION model. In this class, it contains two basic components of NACS subsystem described in CLARION model: one *GKS* object implementing the NACS top level: General Knowledge Store and an array of *AMNet* objects implementing the NACS bottom level: Associative Memory Networks. The details of these two classes will be discussed later. Besides these two basic components, it also declares some variables to represent the parameters and options for configuring NACS subsystem such as the option of Episodic Memory (on or off), the option of Abstract Episodic Memory (on or off), the option of experience-specific chunks (on or off) and so on. The

values of these configuration parameters will determine the instantiations of some components inside GKS or AMNs such as the *EM* object implementing the Episodic Memory and the *AbsEM* object implementing the Abstract Episodic Memory. The details of these two classes will be discussed later.

As described in CLARION model, the main functionality of NACS is to control and coordinate the reasoning processes in NACS such as integrating the outputs from GKS and AMNs using the process of bottom-up activation and top-down activation. The method called *run* in the class is used to achieve it and the procedure is the followings:

- Do reasoning in GKS.

  By calling the method *reasoning* in class *GKS*.

- Do reasoning in a specified AMN network.

  By calling the method *retrieval* in a specified *AMNet* object. Which AMN network will be involved in current reasoning process is determined by command of the NACS control action.

- Get the retrieval result from AMN and send it up.

  This step is to implement the process of bottom-up activation described detailedly in chapter 2.

- Calculate the final result.

  By calling the method *formResults* in the class *GKS*. In this step, to calculate the final result is depended on the retrieval mode : retrieving one activated chunk above the threshold or retrieving all activated chunks above the threshold. Also, the retrieval mode is determined by the command of NACS control action. The final retrieval result is based on the strengths of the activated

chunks. The high strength an activated chunk has, the more opportunity it can become the final result. Basically, we use a maximum function when retrieving only one chunk or using a chunk decider implementing the Boltznann distribution when retrieving multiple chunks to make a final decision on retrieval result. The details of how to calculate the chunk strength is described in the part of specification of NACS in chapter 2.

- Get final result.

Since many parameters relevant to NACS reasoning are determined by the command of NACS control action. To run the NACS reasoning process properly, it is very important to configure the NACS control action and initialize the relevant variables properly. Another method *init* in class *NACS* is to do the action configuration and initialization of the variables declared in class *NACS*.

3.2.5 The class *GKS*

The class *GKS* is to implement the NACS upper level : General Knowledge Store. As described in CLARION model, GKS encodes explicit, non-action-centered knowledge, that is, chunks and associative rules, in the class *GKS*, it declares a vector of *GKSChunk* objects with dynamic size to represent the chunks since the size of GKS is dynamic during the learning process. To implement of associative rules, all of the rules with the same pointing-to chunk are declared inside the pointing-to *GKSChunk* object, details is discussed in the the class *AssocRuleGrp* and *AssocRule*.

Besides the vector of *GKSChunk* in *GKS* class, as described in CLARION model, another important component inside GKS is EM (Episodic Memory) which helps the learning both in ACS and NACS, so, to implement it, the class *EM* is represent it and the details will be discussed later.

Besides the representations of the two basic components of GKS in the class, it also declares some other variables for GKS reasoning method, reasoning parameters, reasoning process, cue to activate the reasoning process, GKS output decision, calculation of strengths of the activated chunks, calculation of response time and so on.

In the method part of this class, the most important is a method declared as *public boolean reasoning()* to implement the reasoning process in GKS. The basic process of the implementation is that:

- check current size of GKS.

  if the the size of GKS = 0, that is GKS is empty, then return *false* to indicate no reasoning at all.

- use the current given cue to activate the chunks in GKS and return the number of activated chunks.

  this is a complicated routine described in CLARION model.

- check the number of activated chunks.

  if the number is zero, that is, no chunks are activated given current cue, so return *false* to indicate no activated chunks at all.

- use the activated chunks to draw the conclusions.

  this is another complicated routine described in CLARION model. In this routine, the basic reasoning methods such as forward chain reasoning, similarity-based forward chain reasoning are implemented. Also, the method of calculation of chunk strength from the source of GKS inference described in CLARION model is implemented.

As described in CLARION model, one iteration of GKS reasoning starting from all the currently activated chunks in GKS and all of the applicable associative rules in GKS fire simultaneously. New chunks are inferred in GKS as intermediate result and plus previously activated chunks both for next iteration of reasoning. To implement this reasoning process. We use a Vector with dynamic size to store all of these intermediate reasoning results. At the beginning of reasoning, it only store the chunks activated by cue. After one iteration of reasoning, the newly derived chunks will be added into this vector and all of the chunks so far stored in the vector will be used to trigger next iteration of reasoning until The termination of one round of reasoning.

The termination of one round of reasoning is depended on one parameter specifying the number of iterations or no limitation on the number of iterations. The parameter is specified by the command of NACS control action. In implementation, if current number of reasoning iterations is equal to the specified limit or no newly chunks is derived, that is, the size of the chunk vector is unchanged in two consecutive iterations of reasoning.

- make a final decision on the reasoning results.

  The final decision actually is based on the specified reasoning method, the reasoning parameters, the strength of the retrieved chunks those are pre-specified in the NACS control action.

- return the final reasoning result.

3.2.6 The class *AMNet*

As described in NACS part of CLARION, at the bottom level, "associative memory" networks (or AMNs) encode non-action-centered implicit knowledge, that is, the implicit associations. To implement the AMNs, a backpropagation network (or one of its many variants) can be used. The class *AMNet* is to represent a AMN network. This class extends the base class *BPNet* implementing the regular backpropagation learning algorithm which can be used to implement the learning process of the implicit associations between input and output. Associations are formed by mapping an input to an output. An input or an output is in the form of a set of dimension-value pairs, in implementation, they are an array of *Feature* objects.

In the bottom level, there are maybe multiple AMN networks, so, at each step, an eligible network will be selected to involve in the reasoning process in bottom level. Actually, how to select an eligible AMN network is decided by the current NACS control action which specifies this information.

We may help speed up the learning process in bottom level, this is another option specified by the NACS control action. If this option is specified, the method *offlineTrain(EMNacsSample[] samples)* will be called to do the offline training. The samples represented by the class *EMNacsSample* for training come from the Episodic Memory (EM) which stores the previously experienced input/ouput associations.

3.2.7 The class *EM*

Episodic memory (or EM for short) is a special part of the GKS. It stores recent experiences in various forms. Part of this memory may be used for helping learning, because it provides additional opportunities for practicing.

The EM stores not only action-oriented experiences involving stimulus, response,

and consequence, along with time stamps of when those occurred but also non-action-oriented experience, in the form of chunks and associative rules (from the GKS), along with time stamps. Two classes are used to represent the two experience types: the class *EMAcsSample* for action-oriented experiences and the class *EMNacsSample* for non-action-oriented experiences. In these two classes, a sample is represented by its different composing parts which makes later offline training much easier to distinguish them and map them into the corresponding input/output dimensions properly.

In the class *EM*, it declares two sample lists with dynamic size, one is for ACS samples and the other is for NACS samples. These two lists are in charge of storing the experienced samples.

As described in CLARION model, information in the EM is recency-filtered. A base-level activation is associated with each item in the EM. The BLA decays gradually and eventually, when the BLA of an item falls below $threshold_{EM}$, the item is removed from the EM. A method called *checkValid()* is used to calculate the BLA of each item in the EM and check their validity (BLA great or equal to $threshold_{EM}$) at each step.

In EM, there are many types of episodic chunks: (1) each state (at each step), as a whole, as observed by the ACS. (2) each step performed by the ACS, as a whole (including the state, the action in that state, the next state following the action, and the immediate reinforcement following the action). In addition, (1) each action rule, (2) each associative rule, (3) each action chunk, (4) each NACS chunk, (5) each association given by the ACS to the NACS, and (6) each association inferred by the NACS.

According to the format of the classes *EMAcsSample* and *EMNacsSample*, we

need a routine in EM to separate a EM chunk into different parts to construct *EMAcsSample* or *EMNacsSample* objects for later offline training on ACS or NACS, for example, separating the "step" chunk into current state, current action, the next state and the immediate reinforcement. Two methods called *fetchAcsChunk()* and *fetchNacsChunk()* are called to do the process of separations on ACS and NACS samples respectively.

3.2.8 The class *AbsEM*

According to the description of CLARION model, the abstract episodic memory (the AEM) summarizes information regarding past episodes experienced by the ACS. The summary information can be useful (1) in helping learning and (2) in extracting explicit knowledge from the IDNs ([**?**]).

We use the class *AbsEM* to represent it. In this class, we declare two *ProbBPNet* objects to represent the action frequency network (AFN) and the state frequency network (SFN) those described as components of AEM.

The goal of abstract episodic memory is that given current state, action (to be performed or performed), as input to AFN or SFN, hopefully, the memory can learn to predict the proper frequency distributions of action, state and reinforcement at next step.

Here, in order to make easier the implementation of the learning process of the frequency distributions of state, action and reinforcement, we use different representations for state, action and reinforcement. Different from the previous encoding with dimension-value pairs, here, states are coded in this case using localist (unitary) encoding in order for the two networks to output state transition frequencies: each state is represented by one individual node (at the input or the output layer

of a network). Thus, A maximum number of states experienced is declared as a constant in the class *AbsEM*. States are identified one by one, as these states are experienced, and coded localistically (in a unitary manner) by the two networks. Actions are also coded using localist (unitary) encoding (for the same reason as that for localist state encoding): each action is represented by one node (at the input or the output layer of a network). Actions are identified one by one as they are experienced (i.e., performed by the agent). Reinforcement is quantized into a certain number of intervals and each of these intervals is coded localistically. In this way, a frequency distribution of reinforcement can be output. In the class *AbsEM*, three lists with dynamic size are used to store the experienced states, actions and reinforcements.

Both AFN and SFN are backpropagation networks, but trained with an alternative algorithm: the probabilistic backpropagation learning algorithm, which is more suitable for learning frequency distributions. The class *ProbBPNet* is to implement this learning algorithm. In the learning algorithm, we assume a two-layer network is used (no hidden layer). The off-line learning rule is:

$$\delta w_{jk} = \alpha \sum_i (d_j(x_i) - h_j(x_i) x_{jk}(x_i) \tag{3.1}$$

where x_i is the input to the network, $d_j(x_i)$ is the given target output for x_i, $h_j(x_i)$ is the actual output representing the frequency, and $x_{jk}(x_i)$ is the kth input (from the kth input unit) to the jth output unit given input x_i. An on-line version of the learning rule is:

$$\delta w_{jk} = \alpha (d_j(x_i) - h_j(x_i) x_{jk}(x_i) \tag{3.2}$$

The training of SFN and AFN is based on episodic data (4-tuples of state, action, new state and reinforcement) from the (regular) EM.

Since the AFN maps the state to the frequency distribution of actions and the SFN maps the state/action pair to the frequency distribution of succeeding states, as well as the frequency distribution of (immediate) reinforcement. So, in AFN, the *state* part of the 4-tuples is provided as input x_i and the *action* part as desired output $d_j(x_i)$. In SFN, both the *state* and *action* are provided as input x_i and the *newstate* and $reinforcement$ as desired output $d_j(x_i)$.

In class *AbsEM*, the method called $offlineTrain$ using a *EMAcsSample* object as argument to implement the probabilistic backpropagation learning algorithm. The procedure is the following:

- Decompose the argument : a *EMAcsSample* object into four parts: state, action, next state and reinforcement.
- Store the components: state, action, next state and reinforcement into corresponding lists if they are new.
- For each network: AFN and SFN, do the followings:
  - fill in the input using the decomposed components.
  - fill in the desired output using the decomposed components.
  - training the network implemented by two processes: $forward$ and *backward* those are two basic routines declared in *ProbBPNet*.

The AEM, the same as the (regular) EM, may be used to help speed up the learning of the ACS. The process of helping learning is the followings:

- generate synthetic "experience" for training the ACS
- through iteratively doing the following:

- Select a state randomly.
- Sample the action distribution, the succeeding state distribution, and the reinforcement distribution based on the selected state.
- The generated synthetic experience (a 4-tuple of state, action, new state, and immediate reinforcement) can then be used to train the ACS, as if it is real experience.

3.3 Implementation of Learning

As described in CLARION model, there are two basic learning processes in NACS : top-down learning and bottom-up learning. And also, learning occurs within each level separately.

3.3.1 Learning Explicit Knowledge

Externally Given Explicit Knowledge

Explicit knowledge can be given, from external sources, in the form of chunks and associative rules connecting chunks. Then, it can be encoded, in a straightforward way, in the GKS.

Usually, the encoding of externally given knowledge is under the control of the ACS, that is, determined by NACS control action. The specific NACS control action for this learning process is the followings:

- dim_1: activate NACS or report

  set element 1 - activate NACS

- dim_2: choose level

  set element 0 - choose GKS

- dim_3: encode externally given knowledge

  set element 1 - encodes as associative rules

- dim_7, dim_8 will carry the original external given explicit knowledge in format of dimension-value pairs.

- other dimensions are disabled.

The implementation of this learning process is the followings:

- Analyze the NACS control action sent from ACS.

- If the action is to do encoding externally given explicit knowledge, process the knowledge (in the format of a set of dimension-value pairs) carried by the NACS control action (in dim_7, dim_8) in different ways:

  if the original knowledge is an associative rule,

  - decompose it into two parts : the condition part and the conclusion part,
  - encode the decomposed parts into chunks by filling in the feature list inside a chunk with original dimension-value pairs information.
  - In the pointing-to (conclusion) chunk, instantiate an *AssocRuleGrp* object (construct the associative rule group)
  - In the *AssocRuleGrp* object, instantiate an array of *AssocRule* objects. (construct each associative rule those can be extracted from the original knowledge.)

  otherwise, encode the original knowledge as a chunk by filling in the feature list inside a chunk with original dimension-value pairs information.

Extraction of Explicit Knowledge

Extraction from the IDNs in the ACS. In the ACS, a chunk may be learned as a result of extracting an action rule: When the condition of an action rule is established, a localist (unitary) encoding of that condition is also established, and thus a new chunk is formed. Actually, the implementation of this learning process in the part of implementation of ACS.

Extraction from the AMNs. To implement this learning process, a method in the class *GKS* named *extractExplicitKnowledge* with the retrieval result from AMN as its argument is declared to do it. The basic procedures are the followings:

- Calculate the strength of AMN retrieval result.
- If the strength is greater than $threshold_ce$, chunk extraction will be considered. With probability of p_{ce} do the followings:
  - insert cue as a new chunk into GKS if new for GKS.
  - insert AMN result as a new chunk into GKS if new for GKS.
  - insert cue and AMN result together as a new chunk into GKS if new for GKS.
- check if allows associative extraction, if *no*, terminate, otherwise do the following:
- If the strength is greater than $threshold_ae$, associative rule extraction will be considered.
  With probability of p_{ae} do the followings:
  - insert cue as a new chunk into GKS if new for GKS.

- insert AMN result as a new chunk into GKS if new for GKS.
- insert an associative rule between cue and AMN result inside the AMN result chunk if new for GKS.

In setting the strength level of an extracted chunk based on the activations of its features, the bottom-up activation process as discussed before is used.

3.3.2 Learning Implicit Knowledge

Assimilating explicit knowledge is accomplished through training an AMN using associations stored in the episodic memory.

In class *AMNet*, the method named *offlineTrain* with an array of *EMNacsSample* objects (samples from EM) as its argument to implement this learning process. The basic procedure is the followings:

- Get the number of training iterations: $trainIterNum$ for the chosen training set at each step.

  The number is a configuration parameter and get its value from the associated *Global* object.

- Get the number of items from EM : $sampleNum$ to use for training this AMN at each step.

  The number is obtained from the argument associated with this method.

- construct randomly a list of samples for training.

  In the this list, the total length is $sampleNum \times trainIterNum$ and each sample should appear $trainIterNum$ times.

- While the list is not empty, do the following:

- Remove one sample randomly from the list for training.
- Decompose the sample into input and (desired) output parts for the network (BackPropagation network).
- set input to the network.
- calculate the outputs of the network.
  using the routine *forwardPass* declared in the base class *BPNet*.
- set desired output to the network.
- update the network. using the routine *backwardPass* declared in the base class *BPNet*.

An AMN may be trained under the direct control of the ACS: The ACS may specifically provide an association to train a particular AMN. So to implement this case of learning:

- Configure the NACS control action as the following:
  - dim_1: activate NACS or report
    set element 1 - activate NACS
  - dim_2: choose level
    set element 1 - choose AMN
  - dim_4: assimilate explicit knowledge
    set element 1 - assimilate
  - dim_7, dim_8 will carry the original association in format of dimension-value pairs.
  - dim_{14}: involved AMNs
    set element 0 - AMN 0 is involved

  - other dimensions are disabled.

- Use the above implementation in which NACS control action is not involved.

3.4 Implementation of Reasoning Methods

3.4.1 Forward chaining reasoning

Drawing all possible conclusions in a forward direction - from known conditions to new conclusions. The basic procedure of its implementation is (at each iteration of reasoning):

For each chunk in GKS, do the followings:

- For every associative rule in the associative rule group associated with the chunk do the following:
  - check the condition and find how many chunks in the condition are activated so far in order to calculate the rule support.
  - calculate the rule support based on specified rule support measure.
- calculate the strength of the chunk based on the maximal rule support among its associative rules.
- A threshold ($threshold_r$) is used to determine whether the chunk is acceptable or not as conclusion.

3.4.2 Similarity-based forward chaining reasoning

Drawing all possible conclusions, using rules as well as similarity-based inference. The basic procedure of its implementation is (at each iteration of reasoning):

For each chunk in GKS, do the followings:

- use the above procedure to calculate the strength of the chunk based on the maximal rule support among its associative rules.
- calculate the strength of the chunk based on the specified similarity measure.
- calculate the final strength of the chunk based on the maximal value of rule support and similarity.
- A threshold ($threshold_r$) is used to determine whether the chunk is acceptable or not as conclusion.

3.5 Implementation of Coordination of the NACS and the ACS

3.5.1 Action-Directed Reasoning

In this coordination, the NACS is under the complete control of the ACS. Current NACS control action format is for this purpose. In current NACS control action format, it may specify performing reasoning in the NACS and obtaining reasoning results from it before taking another action.

Current NACS control action format can dictate the type of reasoning to be performed by the NACS. Reasoning in the GKS may be carried out using a variety of reasoning methods. Usually, forward chaining or similarity-based forward chaining is used.

Current NACS control action format may dictate other reasoning parameters for the NACS as well, including number of reasoning iterations, number of AMN passes for an iteration, and so on.

Current NACS control action format may decide how outcomes from the NACS are to be used. As dictated by an action of the ACS, "filtered" results of the NACS

may be retrieved, either

- in the form of all the "activated" chunks,

  In this case, all the "activated" chunks (with a strength level $> threshold_r$) are reported back to the ACS.

- in the form of a single "activated" chunk.

  competition occurs among all the chunks "activated" (with a strength level $> threshold_r$). The winner is reported back to the ACS.

In either case, we may focus on only a certain type of results, for example, only on chunks that are not contained in the original input to the NACS at the beginning of reasoning. These choices are also determined by NACS control action. When such "filtered" results are sent back to the ACS, they may be stored into the working memory.

Chapter 4

Implementation of MS/MCS

Now, let us discuss the implementation of Motivational and Meta-cognitive subsystem, the more complex kinds of agent/environment interaction in which motivational and meta-cognitive processes are involved.

4.1 Important data structures in MS

4.1.1 The classes *MS* and *Drives*

In our view, the motivational subsystem (the MS) is concerned with drives and their interactions ([**?**]) and why an agent does what it does - how an agent chooses the actions it takes. In current implementation, the *MS* class is used to implement the subsystem.

Similar to implementation of other subsystem, in order for MS to access the globally used parameters and options, a *Global* object is declared inside the *MS* class to refer to the *Global* object associated with the task agent.

As we know in chapter 2, basically, MS subsystem has two parts: the goal structure and the drive states. Also we know that the goal structure is an integral part of both ACS and MS subsystem and in the center of the CLARION model, so inside the class, we declare a *GoalStructure* object to refer to the one used in ACS. Also,

we declare some variables relevant to the input (goal action from MCS and ACS to operate on the goal structure) and output (a selected goal to ACS and MCS) of the goal structure. To implement the part of drive states, the *Drives* class is used to represent the set of drives so far defined in CLARION Inside the class, we declare parameters for calculating the strength of each drive such as the food deficit, food stimulus; water deficit, water stimulus; danger stimulus, danger certainty; night proximity, exhaustion; mate stimulus; belonging deficit, esteem deficit, self actualization deficit. And also declare the routines to access the strength of a specific drive. The default formulas for calculating the drive strengths are implemented in the class. As described in the part of MS/MCS in chapter 2, alternatively, the drive strength can be calculated by a drive network (backpropagation network) using sensory input (including the above drive parameters) as input and drive strengths to MCS as output. We declare a *BPNet* object implementing the backpropagation algorithm to represent the drive strength decision network.

Since in advance of cognitive modelling of specific tasks, the drive network may be trained offline. A routine called *offTrainDriveNet* is used to do the offline training as the following:

- set input (the raw sensory input) to the network.
- calculate the actual output from the network. using the routine *forwardPass* declared in *BPNet* class.
- set desired output (proper drive strengths based on the above mentioned formulas) to network.
- update the network by using the routine *backardPass* declared in *BPNet* class.

4.2 Important data structures in MCS

4.2.1 The class *MCS*

In Clarion, meta-cognitive control regulates not only goal structures but also cognitive processes per se, for the sake of facilitating the achievement of the goals. As we know in chapter 2, the meta-cognitive subsystem (the MCS) is comprised of two levels of implicit and explicit processes, the same as the overall Clarion architecture. In this subsystem, the two levels are both action-centered, and are very similar to the ACS. The bottom level consists of implicit decision networks. The top level consists of groups of rules. To implement MCS subsystem, the class *MCS* is to achieve it.

Similar to implementation of other subsystem, in order for MCS to access the globally used parameters and options, a *Global* object is declared inside the *MCS* class to refer to the *Global* object associated with the task agent.

Inside the *MCS* class, many variables are declared to represent its components:

- a *GeneralNet* object for evaluation of reinforcement;
- a *GeneralNet* object for goal action decision making;
- an array of *GeneralNet* objects for ACS input dimension filtering;
- an array of *GeneralNet* objects for ACS output dimension filtering;
- an array of *GeneralNet* objects for NACS input dimension filtering;
- an array of *GeneralNet* objects for NACS output dimension filtering;
- an array of *GeneralNet* objects for selection of the reasoning methods in ACS.
- an array of *GeneralNet* objects for selection of the reasoning methods in NACS.

- an array of *GeneralNet* objects for selection of the learning methods in ACS.
- an array of *GeneralNet* objects for selection of the learning methods in NACS.
- an *MonitorBuf* object for monitor buffer;

Since the structure of MCS is similar to that of ACS: both have two levels: the bottom level consists of implicit decision network and the top level consists of groups of rules, So, the general *GeneralNet* class is used to represent the base class for a knowledge network which contains an array of *GeneralNetComp* objects representing the knowledge components in the knowledge network, usually, a *BPNet* object representing the bottom level and optional *RerRuleSet*, *IRLRuleSet* or *FixRuleSet* representing the rule types in the top level. It can be extended by the concrete networks of ACS or MCS for specific purpose.

Besides the variables for its components, we also declare all of the relevant variables for the inputs and outputs of the components such as the state, the goal, the drives as input to the component networks and the reinforcement evaluation, the selected goal action, the output from the filtering, selection and regulation networks as output from the component networks.

Besides the variable declaration, we also declare the routines to access to the components and modifying the components.

4.2.2 The class *MonitorBuf*

As described in chapter 2, the monitoring buffer may be subdivided into several sections: the ACS performance section, the NACS performance section, the ACS learning section, the NACS learning section, and other sections. Each section contains information about the bottom level and the top level of a subsystem.

In each performance section, the information about each level of a subsystem includes the relative strength of the top few conclusions, which concerns how distinguished or certain the top conclusions are in relation to other competing ones.

The class *MonitorBuf* is used to implement the monitoring buffer. Inside the class, the variable *acsRelStrengths* and *nacsRelStrengths* (in format of array) to represent the relative strengths, *preAcsChunks*, *preNacsChunks*, *curAcsChunks* and *curNacsChunks* (in format of Vector) to represent the top conclusions.

In each learning section, the performance of each subsystem is tracked for up to a certain number of episodes backwards. This information shows the improvement of performance, namely learning, over time. The variables *acsProgress* and *nacsProgress* (in the format of array) are to represent the progresses in ACS and NACS. The *acsImprove* and *nacsImprove* (in the format of array) are to represent the improvements in ACS and NACS.

4.2.3 Action precedence, priority or overriding

In CLARION model, so far we know that there are ACS action, MCS subsystem and the original configuration of options and parameters have influence on the values of options and parameters. So, there is a problem of action precedence. Now, we set the action precedence (priority, overriding) as ACS action with highest priority, MCS with middle priority and original configuration with lowest priority. The one with higher priority can override the one with lower priority in changing the values of options and parameters.

4.2.4 Goal action decision making

Theoretically, there are two possible ways in which goal setting may be carried out ([**?**]: (1) Balance-of-interests: Each drive votes for multiple goals and the goal that

receives the highest total score becomes the winning new goal. This is preferable to a single-vote approach. (2) Winner-take-all: In this case, the drive that has the highest strength wins the competition. The new goal is chosen for the sole purpose of reducing the winning drive.

The multi-vote approach is implemented in current clarion software system because it allows multiple considerations and different degrees of preferences to be taken into consideration. More importantly, the approach satisfies the requirement of "combination of preferences" as stated earlier.

4.2.5 Reinforcement Evaluation

Generally, the world in which an agent lives does not readily provide a simple, scalar reinforcement signal, as usually assumed in the reinforcement learning literature ([**?**, **?**]). In such a real world, an appropriate reinforcement signal has to be determined internally within the agent, through synthesizing various kinds of information. We may posit that such a signal is internally determined from the drives and the goals of the agent. More specifically, reinforcement signals are derived from measuring the degree of satisfaction of the drives and the goals of the agent.

In the class *MCS*, the above *GeneralNet* object for evaluation of reinforcement is to implement the reinforcement evaluation based on the current state, current active goal and current drive strengths.

4.2.6 Filtering, Selection, and Regulation

There are several aspects include: focusing of input and output dimensions, selection of reasoning methods, and selection of learning methods.

First of all, focusing either on input or output (either specific dimensions or values), is carried out by the MCS, through meta-cognitive actions that suppress

certain dimensions and/or values. The filtering is mainly based on the current sensory input, the current goal, the drives, the working memory, and the on-going performance of the ACS and the NACS (registered in the monitoring buffer of the MCS).

Similarly, selection of reasoning methods is carried out by the MCS, using its meta-cognitive actions that enable certain methods and disable some others. The basis for this type of decision, again, lies in the current goal, the drives, the working memory, the sensory information, and the ACS/NACS performance information (registered in the monitoring buffer of the MCS). The selection can be made separately for the top levels of the ACS and the NACS.

The selection of learning methods is carried out by the MCS in the similar way. The basis for the decision consists of the monitoring of on-going learning performance (performance improvement or the lack of it) as registered in the monitoring buffer of the MCS, in addition to the sensory information, the current goal, the working memory, the drives and so on.

The selection of which level to carry out the processing of a task is a key aspect of Clarion. It may be decided in a variety of ways in Clarion. Normally, the selection is determined based on a (fixed or variable) probability distribution (when stochastic selection is used). The MCS may override the default selection, through designating the top level or the bottom level specifically. Generally, the relative reliance on either level can be altered by changing the probability of selecting a level by the MCS. The setting and changing of other parameters involved in the ACS and the NACS can also be carried out by the MCS. These parameters include, for example, bottom-level learning rate, temperature in stochastic decision making, rule learning thresholds for RER or IRL, and many others.

Chapter 5

Implementation of GUI

5.1 The GUI architecture

To configure the options and parameters described in CLARION model, the more easier and direct way for users without background on computer programming is to configure CLARION by GUI (Graphical User Interface). Although GUI is not a part of CLARION model itself, it is essential especially for the users without background on computer programming to use CLARION system to simulate different human learning tasks.

Since the CLARION model is a comprehensive cognitive model which involves many kinds of options and parameters most of those are inter-dependent, so its implementation architecture is very complicated: the whole architecture is integrative and the internal components are inter-dependent. Current GUI implementation architecture is hierarchical according to the hierarchy inside CLARION model. On the top of GUI implementation architecture, the *GuiClarion* class is in charge of the overall user configuration process. Inside this class, it contains all of the entries to the GUIs of CLARION subsystem configurations such as Action-Centered Subsystem, Non-Action-Centered Subsystem, Motivational/Meta-Cognitive Subsystem and the GUIs for input/output dimension specification, Response Time configura-

tion and Task-Specific CLARION configuration. Besides these entries, there are some other essential routines, for example, loading a specific task to be simulated with an existing configuration, saving the new configuration to hard disk as a new configuration for later use or just to the associated *Global* object to initialize it when leaving the GUI and so on.

Since the GUI architecture is hierarchical, current GUI is implemented in a hierarchical way that when a new configuration is loaded to replace the old one, the GUI system can assign all of the options and parameters of CLARION with the new values properly and automatically with a top-down process of re-configuration. Since two situations are often occurred: one is that some new components are required to be configured by user's selection, for example, in new configuration, user may require configure two ACS external networks in stead of one ACS external network, and the other is that within a component, more parameters are required to be initialized by user's selection, for example, user may specify more input/output dimensions. So, there is a problem: how to configure these new components or parameters those didn't exist in the previous configuration ? To solve this problem, we take the following strategy - "best match first" strategy : 1) for configuring a new component, if the same kind of component has already existed in the configuration, use it to configure the new one, or if the same kind of component exists in another existing non-default configuration, use that component to configure the new one, otherwise use the same kind of component in the default configuration to configure it, (So, every default configuration has all of the components described in CLARION), 2) for configuring new parameter, use the similar process of configuration as described 1).

Within the GUI of each subsystem of CLARION, it contains all of the entries

to the GUIs of its component configurations and the routine for returning to the higher level. Within the GUI of each component of a CLARION subsystem, the major routines are for configuring the associated CLARION options and parameters and the routine for returning to the higher level. Whenever returning to a higher level, the system will check two kinds of interdependencies is still correct: one is the interdependency between the different options and the parameters within or out of a component and the other is the interdependency between different components. These two kinds of interdependency are described implicitly or explicitly in CLARION model. If the interdependency is broken, the values of the relevant options or parameters are unchanged, otherwise the new values are assigned. In the hierarchical architecture, the number of levels within a subsystem may be different depending on the descriptions of CLARION model.

Since in current implementation of GUI, we take "best match first" configuration strategy for different situations and consider all of the interdependencies described in CLARION model, the GUI becomes more intelligent and more efficient.

The following figure 5.1 shows one two sample screens for configuring CLARION model by GUI.

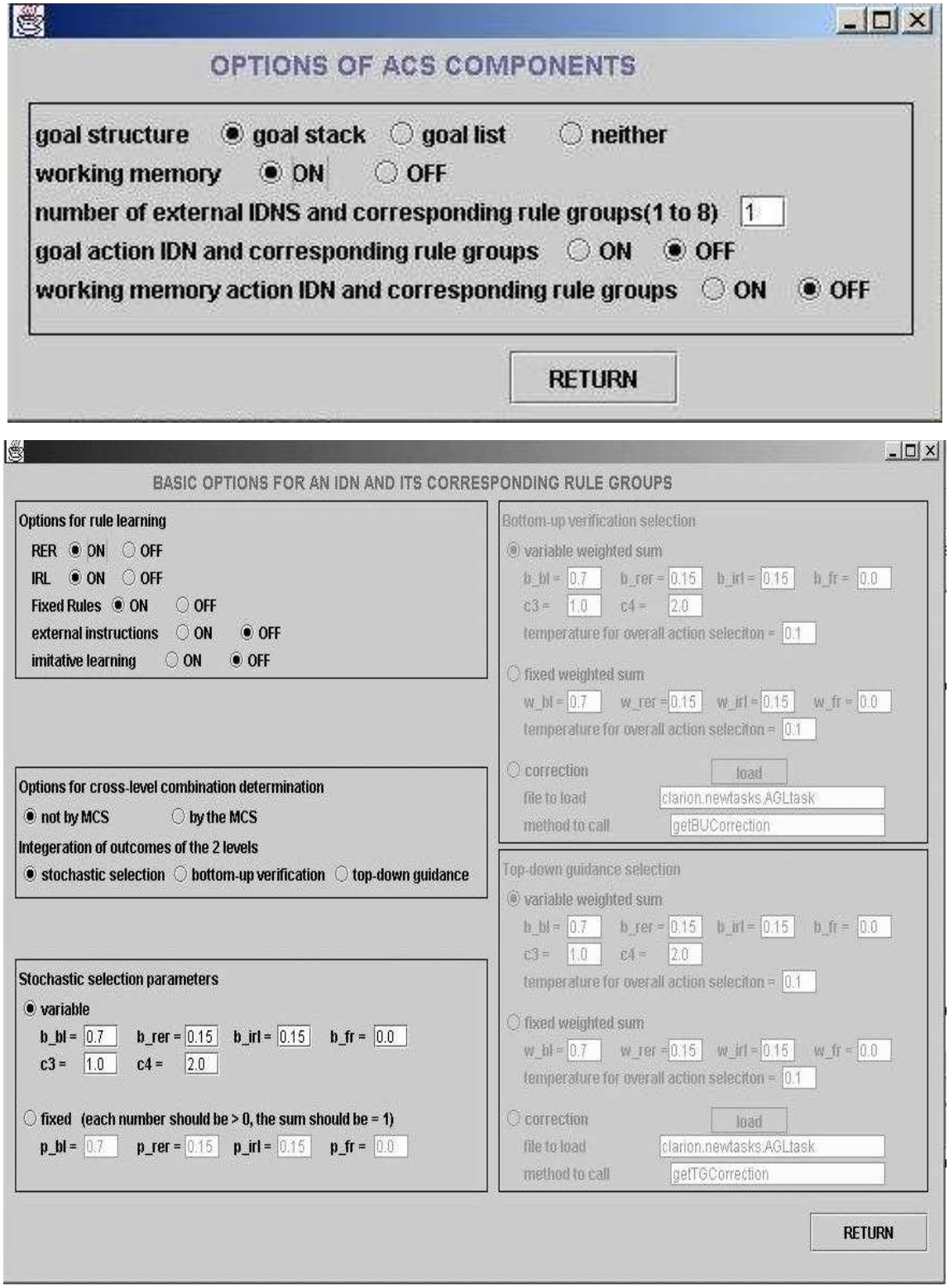

Figure 5.1: The sample configuration by GUI.