Chapter 6

The Instruction

6.1 Introduction

As discussed in Chapter 5, the *instruction* is a conceptual model that implements the action to be performed by the animated agents. It emulates the development of the actions in terms of well-defined motional activities through the planning process. The possibility of extending the level of abstraction of the actions makes the agents exhibit more intelligent behaviour. Consequently, the level of detail involved in planning in instruction becomes significantly more complex.

The natural strategy for coping with varying levels of abstractions is to use hierarchies where actions are described by a number of simpler actions, thereby associating the major instruction with a number of sub-instructions. In Chapter 3 we examined some problem-solving strategies used in many planning and control systems. We have identified that the use of hierarchy is an essential part of the solution, however there are problems in implementing this. In this chapter we explore features of the hierarchical structure that control an agent's behaviour. These features concern the co-ordination of the abstraction levels which involve the internal planning of the actions, the association of one action with another, the identification of the environment context to be handled, etc. We also examine the structure of the Instruction KS (IKS) which is the engine that handles the instruction process.

6.2 The Instruction Concept

A typical example of an *action* in the animation is "John picks up the glass!". This means the goal of the agent *John* is to perform a *pick up* action of the object *glass*. The goal state associated with the action obviously is to have *John holding the glass*. However, in order to explain the nomenclature adopted, we argue that it is much easier to reason in terms of *goal actions* rather than in terms of *goal states*. On the one hand, the *goal state* is a state or a set of states that can be achieved by different actions or plans of actions (Figure 6-1a). For example, John could do many irrelevant actions and end up in the goal state *holding the glass*. On the other hand, the *goal action* can be seen as a bridge that evolves from the current state to the goal state and the action follows a plan (the most appropriate one) that achieves its goal. Therefore, for our purposes the *goal* to be pursued is understood to be the *goal action* and the *goal state* is the outcome of this.

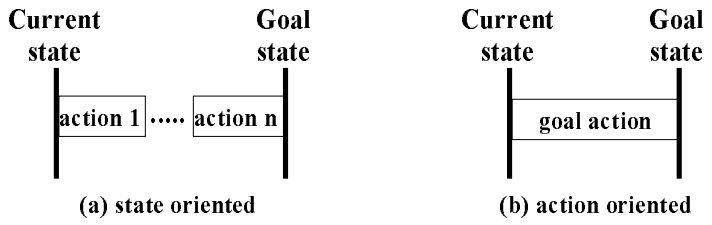

Figure 6-1: Contrast of goals: state oriented and action oriented.

In the present work, the denomination of i*nstruction* is given to a computational entity that implements the concept of *action* discussed in the Chapter 3. It is a control structure which will eventually be transformed into sequences of motor or motional actions. An instruction can be interpreted as a command given by the animator to a robotic agent to carry it out in the animation environment. An instruction can also be interpreted as a machine instruction that is generated consequent upon events in the environment and carried out by the robot itself. In any case such instructions are at a very high-level, the performance of which will depend on the capability of the robot and the circumstances of his surrounding. An animation instruction has the general structure:

[<*subject*> <*verb*> <*object*>]

where <*subject*> is the agent performing the action described by the <*verb*> using the parameter list specified by the <*object*>. The <*verb*> is the focus of the planning activity

which gives the name to the instruction. The <*subject*> and <*object*> are the parameters operated by the instruction.

6.3 Planning in the Instruction

For the purpose of behavioural animation, planning an action can be understood as an explanation of how an activity is to be achieved according to the expectations of the animator or the system developer. The planning of this activity reflects the knowledge and the understanding that the animator has about it and what combined behaviours can accomplish it. Despite the term *instruction*, as far as animation is concerned, nothing is accomplished in the instruction level except the generation of plans. The plan generation is in fact pure symbolic manipulation. A fully developed plan is composed of simple actions that are processed by the Task KS.

This kind of planning intended for the behavioural animation purpose fits the *script-based planning* where plans are formed by *a priori* knowledge rather than by the use of algorithms or heuristics. For example, the high-level action for *picking up a glass from a counter* can be described by either following sequences:

detailed plan	higher-level plan
1) look at the glass 2) walk to the counter 3) approach the left-front side of the counter, 4) extend the right arm 5) pick up the glass from the counter	1) approach the counter, 2) pick up the glass from the counter

Although the specific planning of an action may vary considerably from one person to another, common sense is the best rule to follow. However, a more pragmatic rule to observe is to build plans that do not exceed four sub-actions in length. The longer the plan the shallower it is and part of that plan is likely to occur elsewhere. In such a case it is interesting to create an (intermediate) instruction that implements that part of the plan and to include the intermediate instruction wherever it is required. For example, Figure 6-2b and Figure 6-3b make use of an intermediate instruction *b1*, thus a plan for the instruction *a* is comprised by the sequence [*c1,b1,b2*] and the instruction *k* by [*d1,b1*]. As new skills are incorporated to the animation repertoire, the vocabulary of motions

evolves and the rearrangement of plans naturally becomes apparent. If, however, a plan re-structuration is not sought the repetition is likely to occur and the advantage of abstraction is lost as both Figure 6-2a and Figure 6-3a show.

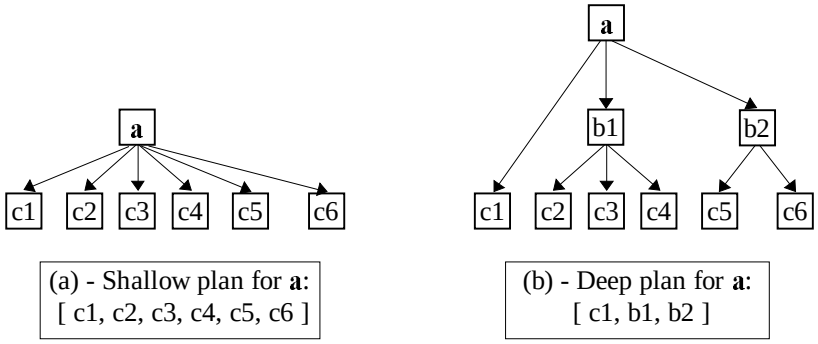

Figure 6-2: Two abstractions for the same plan.

Figure 6-3: Planning with existing actions.

The key idea of planning with a minimum of goals is to stress the objective embedded in the plan rather than details, that is, a plan for an action is expected to specify an overall and approximate solution which will probably lead to the final solution. As the simplest and the basic instructions are created, more elaborate or abstract instructions can be built from existing ones. Therefore, the level of abstraction raised in the animation is reflected in the depth of the planning tree which is also a reflection of the augmentation of the animation database. The most extreme case is also acceptable but it may be rather useless. For example, a new plan specifying a complete sequence of simple operations is designed for every slightly different situation.

6.4 Instruction Operation

The organisation of an instruction as an entity is very simple. It is comprised of the instruction frame, a set of rules called *ruleset*, and a set of procedures. Such simplicity is

necessary for the user to compose new instructions. The activity of an instruction is initiated by the Instruction KS with the creation of an instance from the specific instruction frame and a number of links to other instructions and entities are developed by invoking its ruleset. Figure 6-4 presents a fragment of an instruction in an activity during the development of a plan. A general view of the issues are discussed in the next sections. The implementation of instructions involves a number of issues such as: the ability to recognise the context of the instruction to perform context identification, the problem of expressing the goal as a series of sub-goals (planning), and data representation (frame and rules), linking goals (parameter passing), etc.

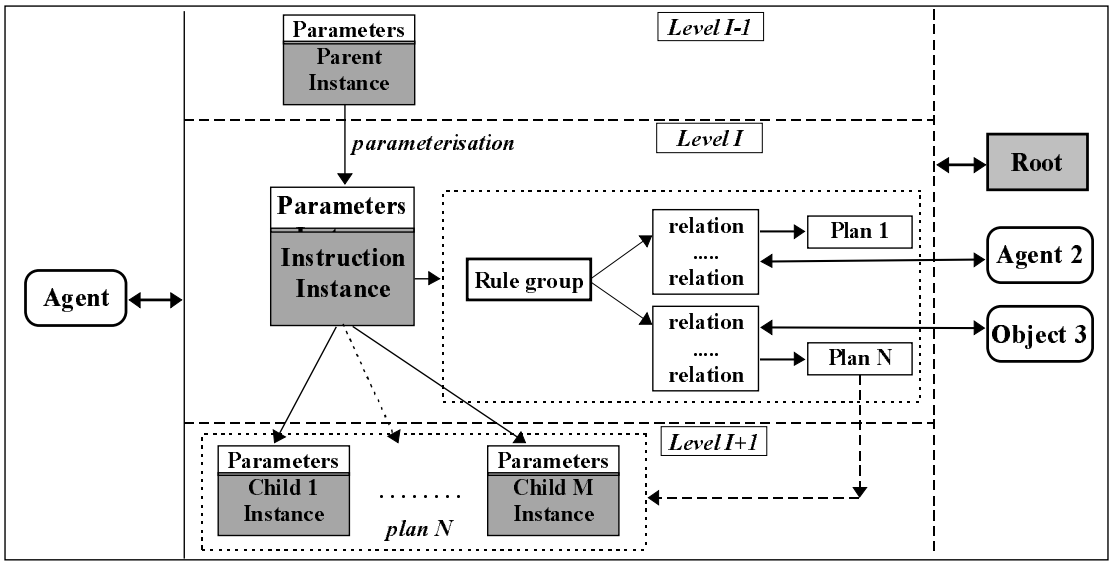

Figure 6-4: Activity of an instruction instance.

6.5 The Instruction Frame

All information associated with an instruction is stored in a frame structure. An instruction is characterised by a frame as it organises all the information related to it such as the goal, parameterisation, ruleset, internal data, etc. There is a generic instruction frame which provides the basis for deriving most of the animation instructions, as shown in the upper part of Figure 6-5. A specific instruction can be derived from the generic form with the addition of customised slots initialised to default values, as shown in the lower part of Figure 6-5. During run-time the instructions frame serves as a template for generating instances in the building of plans. The instances hold run-time data and at the conclusion of a planning process they are removed from the working memory. In order

to differentiate one instance from another a number is appended to the name. For example, the instructions *go_to_i* and *sit_i* have instances starting from *go_to_i1* and *sit_i1*. It is interesting to observe that operations are always performed on instances. Sometimes when referring to an instruction instance (also to a task instance and a message instance) the term instance is simply dropped because the attention is on their conceptual aspects rather than on programming uses.

```
frame instruction;
   default state is plan_selection
   default type is instruction
   default root is nothing and
   default history is none
   default template_ parameter is []
   default parameter_list is []
   default plan_name is nothing
   default plan_sequence is []
   default child is nothing
   default quality is primary          // or secondary
   default owner is system             // or user script
......

frame pick_up is an instruction ;
   default name is pick_up
   default param_list is [place,object,condition,arm]
   default place is table
   default object is glass
   default condition is clean          // objects's condition
   default arm is r_arm
   default closeness is nothing
   default approach_side is front
....
```

Figure 6-5: Derivation of an instruction frame from the generic form.

6.6 The Root Frame Representing Process

During run-time every major goal pursued by an agent is developed into a planning tree. The instances of this tree have some information in common that are stored in a special instance called the *root instance*. As the instances of the tree are independent chunks of information, the use of a root instance as a unique dataset that identifies the tree and helps to avoid repetition of information and permits the IKS control to deal with the tree as a whole process when necessary. For example, the interruption of a planning process at any point of the tree is detected by examining the root. The root frame is presented in Figure 6-6 and the operation of a root instance during run-time is depicted in Figure 6-7.

```
frame root;
  default source     is system and
  default agent      is nobody and
  default top_instance is nothing and
  default interruption is off and
  default current_instance is nothing and
  default focus      is on and
  default time_out is 0 and
  default callback is nothing .
```

Figure 6-6: Root frame.

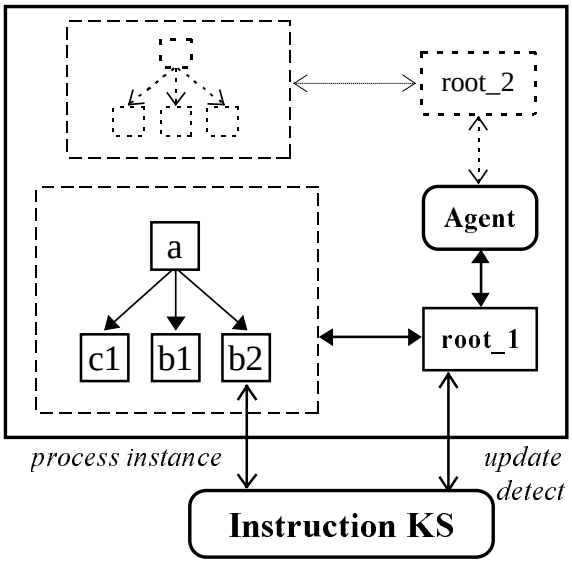

Figure 6-7: Linking agent, root, and plan tree.

6.7 The Instruction Parameter Template

Part of the data operated on by instructions in general are external inputs which enter through parameter passing. Every instruction is templated with a list of parameters given in the *template* slot of the instruction frame and the parameters of this list are also slots of the instruction frame. The parameters are thus initialised with default values that may be used in the absence of external data. This implies that all parameters in the template are not always provided when the instruction is being used. In fact, the template of an instruction tends to become of general use and the explicit specification of all parameters is required only in few cases.

Figure 6-8 shows the *pick up* instruction as a "black box" where only the templated parameters are relevant for the planning of an instruction. However, if an instruction has

a null template list this does not imply that the instruction has a very restricted role. As mentioned in section 6.2 the agent name is included as a component of the main instruction which is stored in the root. Therefore, the instruction can still fetch vital information on which to operate.

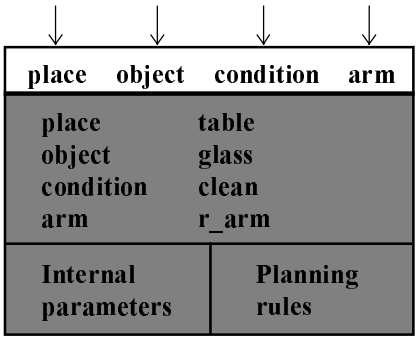

Figure 6-8: Instruction as a "black box".

6.8 The Plan Representation

A sequence of actions is coordinated in the form of a plan. The general format of a plan is given by:

***plan_is*(<*plan_name*>, [<*instr1*>, <*instr2*>, ..., <*instrN*>])**

< *instrI*> => [<*instruction*>, <*param1*>...<*paramI*>] or <*instruction*>

where the clause *plan_is* is associated with the *plan_name*, which identifies an instruction plan, with a sequence of instructions, or tasks, each one with its relevant parameters. A comparison can be made with a conventional procedure, where the plan name is equivalent to the name of the procedure and the sequence of instructions in the plan is analogous to a sequence of procedure calls accompanied by the relevant parameters. These parameters are bound to the parameters indicated by the templates of the corresponding instructions.

In order to allow agents to achieve goals in the face of a diversity of conditions, a number of alternative plans are provided. The system is then able to select the appropriate course of actions according to the current context. For example, some of the available plans provided for the *pick_up* instruction are defined by:

plan_is(pick_1, [pick_up_t]) .

plan_is(pick_2, [[face_i, object], pick_up_t]) .

plan_is(pick_3, [[approach_i, place, object], pick_up]) .

In order to differentiate one node from another it is convenient to adopt a naming convention. Task nodes such as the *pick_up_t* bears the *_t* suffix denoting that it is a *task* data type. Similarly, *_i* denotes an *instruction* node. In the case of the first plan, *pick_1*, the agent might be in the correct position, ready to extend its arm and reach the object, so a simple task is enough to accomplish the goal. Plan *pick_2* is chosen when the agent is not facing the target object, but when the agent is close enough. Parameters in plans such as *object* and *place* are discussed later. Finally, *plan_3* is a more general one that is used when the agent is reasonably far from the object. It should be observed that the second node in *plan_3*, *pick_up*, which is an instruction, is a recursive form which will be discussed later.

6.9 Plan Selection

Each instruction has a number of rules for selecting the corresponding plans. The rules are organised in groups of rulesets and invoked in sequential order by default. Unless another selection criteria is in place, these rules are usually considered in the fixed order in which they are listed because they incorporate a recursive strategy. An attempt is made to identify the simplest context first. The recursive feature is discussed in a later section. For the sake of simplicity, the name of the ruleset is the same as that of the instruction:

group <*instruction_name*> <*plan_name1*>, < *plan_name2*>,..., <*plan_nameN*> **.**

The mechanism for the selection of an appropriate plan is decided by rules. Each rule identifies one class of situation that is accomplished by a plan and sets up some of the internal parameters. One collection of the parameters in the plan comes from the instructions internal parameters and the other part from the templated parameters. The inference rule for plan selection is typically of the form:

rule <*plan_name*>
  if plan(<*instruction_name*>**, Instance, Agent) and**
    <*conditions*>

then
choose_plan(<*plan_name*>, **Instance) .**

which means that, if the <*conditions*> are satisfied, then the <*plan_name*> is chosen to implement the instruction. However, <*conditions*> have a more important role in the instruction process because, very often it is not a matter of choosing one plan or another to make the instruction goal achievable, but of identifying the context with which the instruction is dealing and processing this information. The proper identification of the context leads to the correct choice of a plan and the consequent passing of the appropriate parameters to the plan. The presence of the variable *Instance* is mandatory in the *plan* clause, in the *choose_plan* clause, and very often in the <*conditions*> tests because it links and stores information related to both the plan selection and context identification stages. The variable *Agent* also provides important information in context identification.

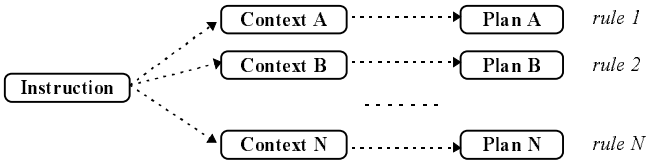

Figure 6-9: Ruleset for plan selection associated to an instruction.

For our purpose, *context* are the surroundings which relate an agent with objects that potentially can be involved in an action. The particular situation of the agent is also part of the context and it is also considered in the planning process. For example, suppose that John intends to pick a glass, some of the elements of the context considered for a plan selection are the posture of John and the distance between John and the glass. If John is standing up and the glass is out of the reach of his arm, he will have to approach the glass first. However, if John is standing up with the glass within his reach, he can simply pick up the glass. Although the difference between the former and latter cases is only the distance, the situations have two distinct contexts requiring different analyses.

6.10 Context Identification

The <*condition*> component referred to in the previous section has the main role in the selection of the appropriate plan, so that they can be called as *plan selector*. In fact plan

selection is an activity based on context identification. It is composed of relations which can be facts or procedures (e.g., the procedure *can_pick_up* in Figure 6-11). Some of the purposes of these relations are:

- to verify key constraints required for carrying out the associated plan;
- to validate parameters. Some parameters may have or may not have been provided with adequate information for the context considered;
- to provide the complementary parameters for the instantiation of the plan;
- to identify which context it is to be applied. Some parameters might be missing or not evident but the general context is identified. The missing parameter may possibly be derived from the environment.

The creation of a large number of rules to implement an instruction can be unmanageable. Usually an instruction might have a limited number (about up to five) of stereotyped contexts with the same number of relevant plans. In fact the combination of a number of parameters helps to deal with several similar (or repetitive) contexts where just one parameter or another is missing, but the context is ultimately the same. Thus, the presence of one parameter compensates for the absence of another by, whenever possible, retrieving or deducing the missing data consistently. At the same time it avoids the repetition of plans for dealing with similar contexts. For example, consider the context of an agent standing up in front of a *bar_counter* and having to pick up a glass from a counter, if the glass is specified (e.g., *glass1*, which happens to be on the *bar_counter*) then he knows which counter the glass is on. However, if the agent is a *barman* who usually works on that specific counter then he knows that he should pick up a glass from the *bar_counter* (which is supposed to have at least one glass). Therefore, both cases fit the context and similar plans are applied to pick the glass.

In one way or another parameters can be derived and employed in the plan. Thus, in order to avoid such problems and keep one plan selector for a truly distinct context, the evaluation of the <*conditions*> for a plan is done separately from the rule. For example, the use of rules in Figure 6-10a is cumbersome. It is much better and maintainable to have instead one rule with a rational condition testing the different variations of a context as shown in Figure 6-10b. In this framework this condition testing is implemented in *flex*/Prolog procedures called *relation*.

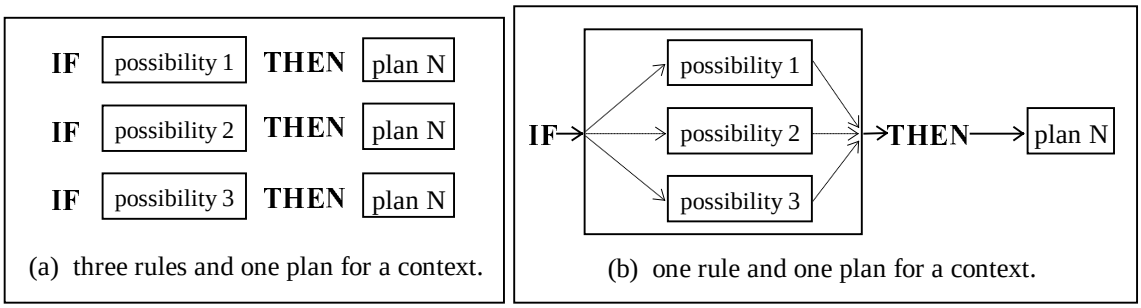

Figure 6-10: Use of relations in a context identification.

Figure 6-11 presents an example of condition testing, *can_pick_up*, which is implemented as a Prolog procedure. The first alternative of the *can_pick_up* relation verifies if the target object has been passed to the instruction instance. That is, the relation *has_selected_object* could have the simple task of confirming that an instance of a glass object has been selected by the agent, this information having been passed down from a parent instance. In the second alternative, *can_pick_up*, could have the more elaborate task of requiring the object to be picked up from the selected place. For example, the information to be considered could be (*counter* = *bar_counter*), (*object* = *glass*), and (*condition* = *used*).

There are, however, other considerations that are important in both alternatives of the relation entries but not apparent in the *can_pick_up* level. For example, the selection of the arm the agent is going to use. At this level there is also a consideration that determines which is the preferred arm if both arms were available. If neither of the arms were available the instruction would fail completely.

```
rule pick_up_3
if  plan( pick_up, Instance, Agent ) and
    can_pick_up( Agent, Instance ) and
    the proximity of the Instance is too_far
then choose_plan( pick_3, Instance ) .

a_plan( pick_3, [ [approach, place, object], pick_up] )

relation can_pick_up(Agent, Instance ) if    // Alternative 1
   has_selected_object( Instance )  and
   find_distance(Agent, Instance ) .

relation can_pick_up(Agent, Instance ) if    // Alternative 2
   has_selected_place( Agent, Instance, Place ) and
   can_select_object_from( Instance, Place, Object ) and
   the object of Instance becomes Object .

relation has_selected_object( Instance ) if
   the object of the Instance is an instance of SomeObject
   and SomeObject is not nothing.
 .....
```

Figure 6-11: An example of a plan selector, for the *pick_up* instruction.

6.11 Plan Instantiation

The instantiation of a plan involves the creation of new instances for each of the instruction nodes of the plan. As each node in the plan is accompanied by a parameter list (PL), plan instantiation is a process that also involves the binding of these parameters to the corresponding newly created instances. Each instruction has sufficient generality to deal with a specific class of problems through the use of parameters. Thus, each instruction has a template list (TL) through which communication is established at run-time between the instance (derived from the instruction) and its immediate parent instance by the way of the parent's plan as shown in Figure 6-12. The second and last activity after the instantiation process is the context identification stage. The job of the context identification can be trivial if actual or specific data are passed into the instance; or more elaborated if the data passed is not specific, that is, it represents categories, types, or generic class of objects. In the latter case, the "generic" parameters behave as variables and at some stage of the overall planning process, a context identification will determine the specific objects referred to by these variables for a given context. That means that parameters are not always initialised and they may be passed from parent to

children as they are. The initialisation of parameters in a plan by the context identification will become clear in the next sections. Once the context identification activity in the rule has been successfully completed then the associated plan is selected and the cycle of plan selection and plan instantiation are applied to the children nodes and so on. In section 6.16 a complete overview of the instruction control process is given. The overall plan instantiation is achieved by forward and backwards binding.

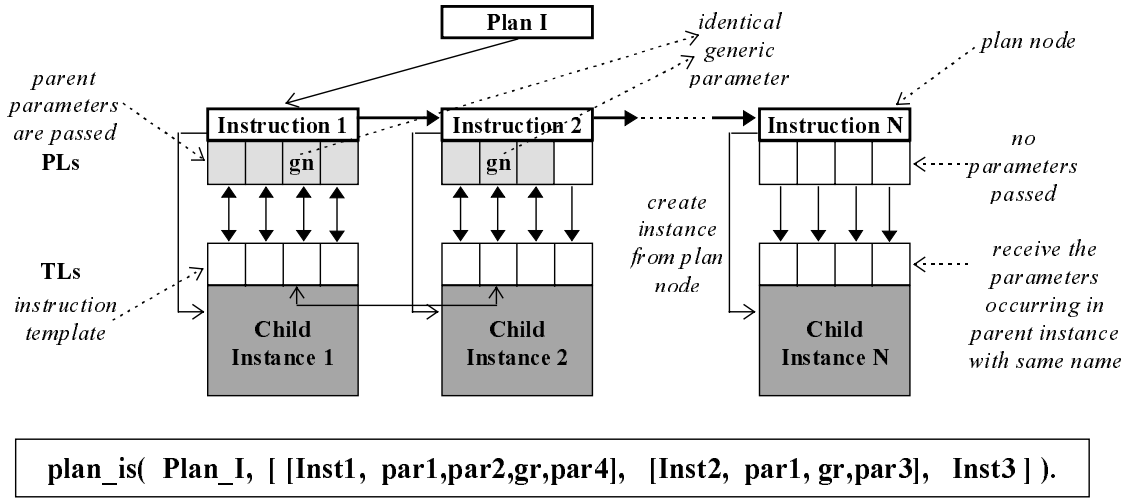

Figure 6-12: Plan instantiation and parameterisation.

6.11.1 Forward binding

In forward binding the data from a parent instance is copied to the child instances created from its plan. The process of matching a plan node's parameter list with the child instance's template list observes the following rules:

- if a parameter from PL is an instance of a generic object such as a glass or a robot then the parameter is called *grounded* and the corresponding parameter in TL is assigned to this instance as a new value. In Figure 6-13 the parameter passed, *bar_counter*, is an instance of counter;

- if the parameter is a slot name that occurs in the parent instance then the value of that slot is assigned to the corresponding parameter indicated in the TL as a new value. Similar to procedure calls in a conventional program, the corresponding parameter names need not be the same. In Figure 6-13 the parent's parameter *state* is a slot that does not occur in the child instance. However, there are two further cases to be

considered in this situation. The parameter might be a "dummy" one and is used as a pad to keep the matching order and allow the access to the following parameters. The other case might occur when the same dummy parameter occurs in two or more PLs as it is the case of the *gn* parameter in the instructions 1 and 2 of Figure 6-12. This case has the purpose to connect two or more actions. It works in combination with backwards binding and thus will be discussed next;

- if the number of parameters in PL passed to a child instance has fewer parameters than that in the child instruction's TL, then the excess parameters of the TL cannot be matched and, consequently, not evaluated. Thus, the binding process checks whether the omission of parameters was deliberate. That is, if a parameter name in the instance's TL also occurs in the parent instance then it will be considered as deliberately omitted and the parameter's value will be copied from the parent node to the child node. This is just a matter of convenience for the animator. However, if the parameter does not occur in the parent node then nothing is done and the problem with the non-initialised parameter is left to context identification to find a proper initialisation if needed. For example, in Figure 6-12, the child instance 2 receives one parameter short from PL than that indicated by its TL. While the child instance N receives an empty PL. For example, in Figure 6-13 *arm* is a parameter of the TL that remains unaffected by the binding process but it has a default value which has to be tested for consistency by the context identification.

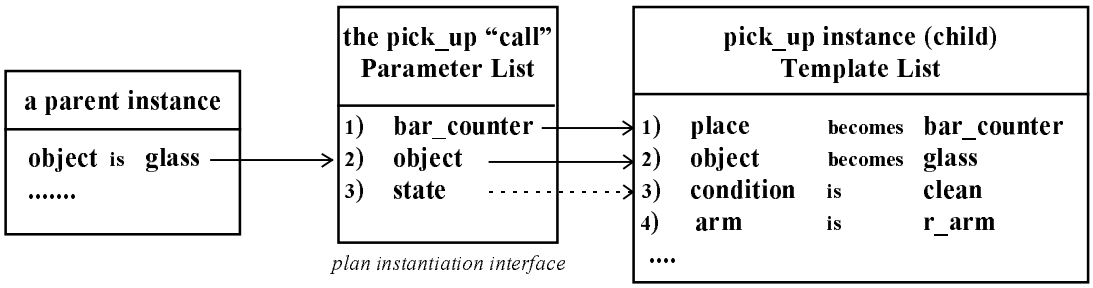

Figure 6-13: Example of parameter binding.

6.11.2 Backwards binding.

Whenever possible at the completion of its activities, a child instruction transfers data, which has been obtained from the current context, back to its parent instance by using backwards binding. The process is very similar to forward binding but in the opposite direction. However, if some parameters in PL are partially or totally omitted then these are not passed back. Special attention is given to the dummy parameters in the PL with slots that do not belong to the parent instance. In this case the backward binding process creates new slots, as indicated in PL, in the parent instance. If the dummy parameter occurs again in the subsequent nodes' PLs of the same plan level then they establish a connection. That is, the second occurrence of the dummy parameter is no longer a dummy because a slot has been created by backwards binding at the first occurrence. Thus the parent instance behaves as temporary storage. The important outcome is that data determined by one instruction is made available to others. Such a feature allows plan nodes to share the data and to make subsequent actions consistent.

6.12 Grouping Instructions

Grouping of instructions is a simple but special case of the grouping plans discussed in section 6.9. There are occasions that a collection of different activities specified by instructions are related to achieving a main goal. For example, each agent in an animation environment has an associated behaviour such as *barman*, *waiter*, or *customer*. When such an agent becomes idle, having no specific instruction to perform, then the activity characteristic to his behaviour is performed, as is the case of the instruction *manage_bar* for the *barman* agent. The instruction *manage_bar* refers to a goal which can be satisfied by one of the self-contained alternative activities such as, *clear_counter*, *fill_up_stock*, or *observe_people*. *manage_bar* is effectively an abstraction, or alias, for a number of more specific activities which can individually achieve that goal in different contexts. Such instructions are called "self-contained" because they are goals that normally take no part in a plan and therefore they have null template lists. However these instructions can be grouped as alternative plans that achieve a common goal without receiving specific directions through parameterisation. Therefore, the instruction that groups others acts as an instruction selector (Figure 6-14). One of the tests in such

a plan selection process is to verify whether an instruction has been selected recently. This is useful because in most of the cases more than one alternative activity is acceptable. Such a resolution avoids always selecting the same alternative consecutively. For example, a customer agent when in an idle condition (e.g., waiting the *barman* services) can *observe_people* or "flex the arms" leisurely.

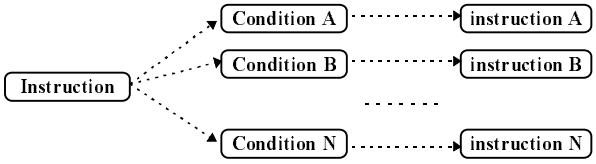

Figure 6-14: Instruction as a collection of activities.

6.13 Recursion in Instruction Planning

Complex goals can be satisfied using a recursive process. The solution of a problem is a self reference of a simpler case until the recursion is replaced by non-recursive instructions, and ultimately by tasks.

A similar approach is employed in the process of planning an action in the situation that the goal can be achieved for different contexts, each one with a varying level of complexity. If the goal is to be achieved in a complex context, the solution in the recursive approach is to realise a preliminary action in such a way that a new context obtained for solving the original goal is a simpler context. This can also be understood as a succession of changes of contexts, each one requiring a simpler plan for achieving the goal. An instruction can be employed in a number of contexts with solutions that could be classified as *trivial*, *simple*, or *elaborate*. In the particular case of recursive instructions, at least one of its plans is of *elaborate* type and at least one the other plans is a *trivial* or *simple* type. The *trivial* case occurs when the goal of an instruction is already satisfied and no further action is needed (Figure 6-15a). In the *simple* case the solution comprises a sequence of actions as shown in Figure 6-15[b,c], where self reference does not occur in the plan. In the *elaborate* case, the context for the instruction might be too far from the "ideal conditions" as those tackled by the *trivial* or *simple* types of plans, so a recursive plan is preferred. That is, the recursive plan includes a preparative action, or a sequence of them, that brings the action to a simpler

context and then attempts the original action again (Figure 6-15[d,e]). The BNF, or production rule, equivalent to Figure 6-15 is given in Figure 6-16.

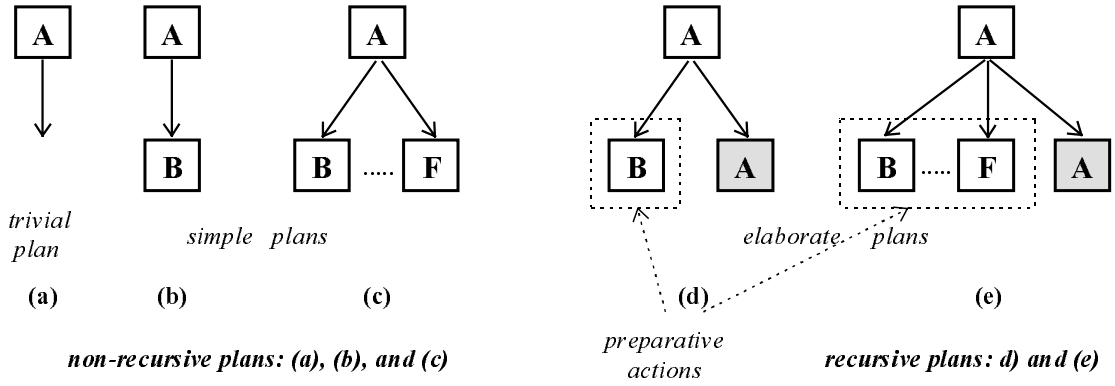

Figure 6-15: Two typical types of instruction plans.

A -> λ
A -> B
A -> B .. F
A -> B A
A -> B .. F A

Figure 6-16: Instruction plan as production rules.

For example, the *pick_up* instruction is a typical example of a recursive instruction. If an agent had to *pick_up* a glass on a counter across the room, he would have to be there before actually trying to pick up the glass. Thus the plan would comprise *approach the counter* and *pick_up the glass*, which follows the shape of the plan given in Figure 6-15d. The *pick_up* node in the plan, unlike the parent, will probably be able to achieve the goal.

6.14 Plan Execution

The execution of a plan is interleaved with the building of the plan. Because the execution of a plan may take a variable length of time to accomplish, it is unreasonable to plan a complete sequence of actions in advance. Depending on the dynamics of the environment, the state of the environment at various stages of the operation may be different from that at the beginning therefore planning should be a continuous process.

The use of a hierarchy as the planning structure is very well suited to the requirement for "planning on demand". Higher level instructions will obviously have a higher level plan. Such a plan reflects less knowledge about the environment and more about the goal as shown schematically in Figure 6-17. These goals form a main sketch of a plan with "landmarks" which guide the development at the particular level of abstraction. The successive refinement of plans into lower levels incorporates more knowledge about the environment for increasingly specialised goals. Eventually an unexplored node of a plan is expanded into a branch where the leading node is a simple action. This action is executed at the first opportunity that it is considered in the tree traversal. If the next step of a plan (in breadth, obviously) is an instruction node it will be expanded, otherwise it will be executed. The traversal of the planning tree in this way, executing actions and expanding instruction nodes as they occur, allows the Instruction KS to select plans that are compatible to the occasion. In other words, premature commitment to information that might not be valid at a later time is avoided. Thus, such an operation exhibits adaptiveness and also there is no loss of "sense of direction" which is guaranteed by the "skeletal" structure of the plan as a goal specification.

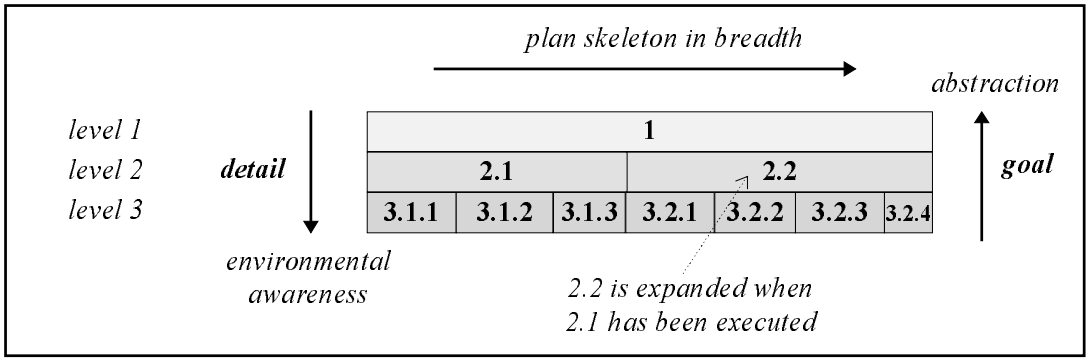

Figure 6-17: Contrasting knowledge with goal.

6.14.1 The Instruction Knowledge Source

The handling of instruction instances in the framework is done by the instruction knowledge source (IKS). The IKS is implemented as a state machine which monitors the progress of the instances by applying procedures corresponding to their current state. Each state is shown in Figure 6-18 as a rounded box. The *ascend_tree* boxes in particular do not represent states, it is used to highlight the shift of focus of attention

from a child node back to the parent node. Most of the operations indicated by the states have been discussed and comprise the following steps:

Initialise and refine instruction states. An instance of an instruction is created and the parameters in the template list are bound to the parameter list. *Initialise* is concerned with the topmost (major) instruction that gives rise to the planning process. The *refine instruction node* is concerned with the initialisation and binding of instructions with exception of the topmost instruction.

Select plan. A plan is selected for an instruction according to the context identification.

Follow plan. The next node of a plan is examined to determine whether is an instruction, a task, or a message type. If it is an instruction node then refinement in depth order will occur, otherwise, the focus of attention is shifted to the corresponding KS that operates on that type of node.

End plan. The conclusion of a local plan causes the Instruction KS to shift its focus of attention up to the next step higher in the parent's plan. If the overall plan is finished, a major update is effected in the performing agent's database prompting the agent to undertake new activities.

Interrupt instruction. An instruction is interrupted before it can select a plan and it resumes plan selection when a new opportunity arises.

Callback instruction. The instruction, which is awaiting for an opportunity to execute, is called back to activity by another process that has been interrupted, stopped, failed, or ended.

Re-assess instruction. The select plan state can be done at most twice for a given instruction.

Failure control. Failures in general are handled in this state. This is discussed in the next section.

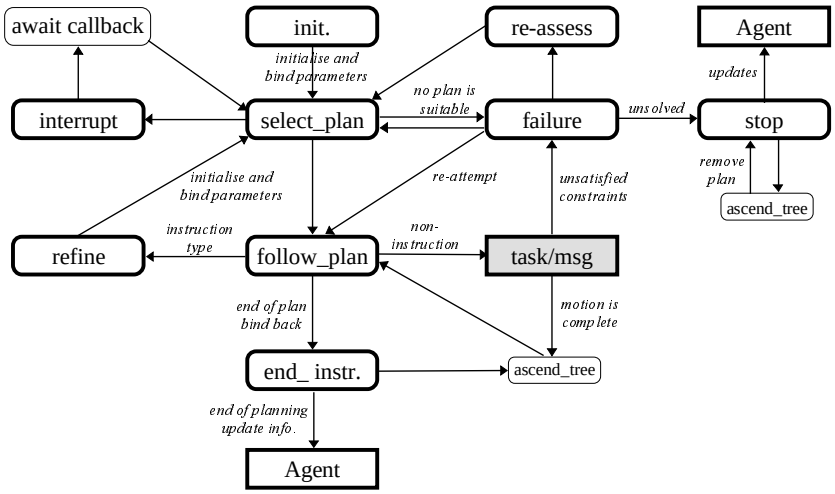

Figure 6-18: Instruction Control Scheme.

6.14.2 Failure Control State

The failure control state is a single rule in the Instruction KS that centralises a number of recovery alternatives for overcoming failures which may arise in the execution of plans. A number of relations, that is, Prolog procedures, are employed in this rule to deal with different failure conditions in a similar way to the context identification process (Figure 6-19). Each relation has some facts that identity the failing condition, which if matched, result in an attempted recovery process. Failures may occur in situations where two processes are competing for the same resource (an object), for example, the *barman* and the *waiter* are about to pick up the same glass from the counter, the waiter who has started moving earlier gets hold of the glass first and makes it unavailable. As the targeted glass "vanishes" from the counter the *barman* tries to pick up another glass if any, otherwise he will order more glasses from the *supplier*. That is, the instruction instance is set to the *re-select* state which will re-plan with another object as a target. Failure may also arise when a plan is not sufficiently well structured, that is, when there are situations in which one action does not properly follow another. In such a case it is necessary for the animator to re-examine his repertoire of instructions.

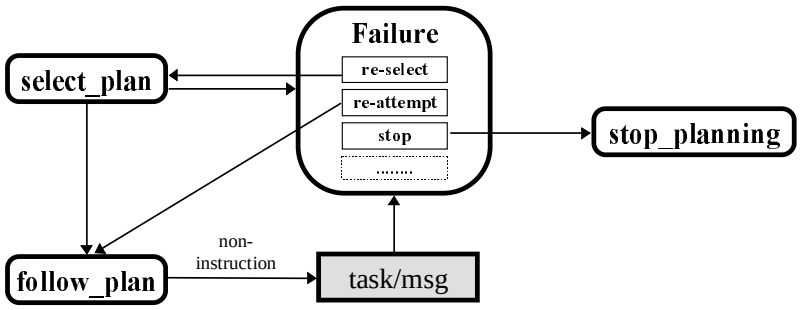

Figure 6-19: Structure of the failure control.

Three cases of failure handling are presented here. However, if specific error handling is required then it is just the case of adding a new relation to deal with that particular feature and optionally add a new state in the form of a rule into the Instruction KS

Figure 6-20a presents the failure handling state as a rule with a powerful relation, *eval_failure*. *eval_failure* has multiple entries that identify different failing conditions. The rule deals with two main groups of failures. The first group comprises those failures identified as unsuitable for proceeding and some of the entries that are used to identify such patterns of failures. For example, Figure 6-20b recognises in the entry *n* that an attempt has been made to overcome the problem and it has failed again. Thus, *eval_failure* has the condition in the rule FAILED_STATE which sets the instance to the *stop_planning* state. The other group consists of those failures that are individually identified with prescribed associated solutions. For example, the instance is set with a suggested state (e.g., *select_plan*) alongside other information. At the end of the entry there is a *fail* statement which "forces" an exit *eval_failure.* This also leads to the exit of the rule itself and, therefore, the evaluation of the instance to proceed in the suggested state which is specified as a rule in the IKS.

```
RULE  FAILED_STATE
IF plan( Instruction, Instance, Agent ) and
   Instance's state is failure and
   eval_failure( Instance )
then
   Instance's state becomes stop_planning .
```

(a)

Entry points for eval_failure(Instance).

1) if Instance's drop_on_failure is true and
   .....
   inform_parent(Instance) and
   loose_focus_of_attention(Instance) and
   ! and fail .

2) if Instance's history is none and
   Instance alternative is Alternative_rule and
   ....
   Instance's state becomes select_plan and
   ! and fail .

.....

n) if Instance's history is re_attempted
   ! and true . // No alternative, stop planning.

(b)

Figure 6-20: Partial coding for failure handling.

An ingredient in the handling of failure in *eval_failure* is the use of *history* along with other relevant parameters related to the specific failure in the instruction instance. The history parameter provides an indication of whether the failure in the instance has been handled previously and what solution was attempted. This mechanism avoids indefinite backtracking in the planning process. The use of customised parameters in instructions helps to supply adequate decision making for certain kinds of instructions. For example, in the entry 1) of *eval_failure,* if the *drop_on_failure* parameter is true then the control is shifted to the parent instance because it makes no sense for certain instructions to re-attempt the plan or to re-evaluate a new plan because of the nature of the failing conditions. The entry 2) verifies if the specific instruction has an alternative instruction to handle the "unusual" situation. The entry n) advises that planning of that instruction should be abandoned and that the evaluation of the failing condition should be shifted back to the parent instance.

The occurrence of a failure in a planning tree node causes the IKS to attempt to perform failure handling. If it succeeds, obviously the planning proceeds normally. Otherwise, the parent node is set to the failure state and the focus of attention is shifted further back to the parent node. The IKS applies the same evaluation to the parent node to fix the problem and keeps a note in the *history* slot. If, eventually, the control returns to that node in the failure state and discovers that an attempt has been made to solve problems in an earlier stage by looking at the history then a final decision might be to abandon the process.

6.15 Examples of Instruction Parameters

The features discussed above can be clarified by examining some examples. First we examine some variations of the *pick_up* instruction:

1 - John pick_up the tall_glass from the bar_counter.
2 - John pick_up a used glass from a counter.
3 - John pick_up a glass from a counter.
4 - John pick_up (an object) from a counter.
5 - John pick_up a glass (from a place).

Some of these five alternative commands have parameters which are lacking in degree of definition or completeness. Those with missing parameters such as which *arm* is to be used, may resort to default values or determine them from the context. In the case of the first command, the parameter communicated to the instruction is quite specific as the target object is explicitly stated to be an instance of *glass*.

6.16 Example of an Instruction - *clear_counter*

The instruction *clear_counter* is used by the *barman* with the intention of moving the glasses in the *used* condition from the *bar_counter* to the *exchange_counter*. This instruction receives no parameters and its only plan specifies three major goals given by:

```
plan_is( clear_counter,
        [[check_i,  counter, objects_list, glass, used],
         [pick_up_i,  counter, glass, used],
         [put_i,  shelf, glass]]  )
```

There are similar instructions for *fill_up_stock,* used by the *supplier* to bring new glasses from his counter to the *exchange_counter*. If the *supplier* finds *clean* glasses on the *exchange_counter*, he exits from the plan because a clean glass has been found and one need not be supplied, otherwise he will go to the next step in the plan.

```
plan_is( fill_up_stock,
          [[check_i,  shelf, objects_list, glass, clean,return],
           [pick_up_i,  counter, glass],
           [put_i,  shelf, glass]]  )
```

In the *fill_up_stock* plan, described above, the *check_i* instruction is applied to the members of the *objects_list* until the first occurrence of a *glass* is found with the *condition* is *clean*. The actions undertaken by *check_i* are described by the *check_1* plan given below.

```
plan_is( check_1,
        [[goto_i,  place],
         [look_at_i,  object],
         [select_i,  object, condition, action]] ) .
```

In this plan, one *object* at a time is approached and examined by the barman. Literally, he goes to the *place* where the *object* is, looks at the *object* and if the *condition* on the *object* is met then the control action specified by *action* is taken.

```
frame person is an entity ;
  default type is person
  default posture is standing_up
  default support_leg is both
  default l_r_handed is right
  default r_hold is object
  default r_arm is st( straight, free )
  default r_leg is st( straight, free )
  ....

frame barman is a person ;
  default activity is barman
  default behaviour is bar_man_job
  default colour is gray
            name is john
            shelf is exchange_counter
            counter is bar_counter
            r_arm is st( hold, pick_up_1 )
            r_leg is st( straight, walk_1 )
....

frame glass is an object ;
  default type is glass
  default user is nobody
  default place is table
  default condition is clean
  ....
```

Figure 6-21: Partial description of the *barman* and *glass* frames. (Slots without "default" are examples of instantiated values.)

Figure 6-21 presents examples of frames of the animated objects which will be used in Figure 6-22. In Figure 6-22 each box node of the *clear_counter* instruction tree has information in the following order: name of the instruction; parameters specified by the parent plan, indicated by *p*; and the full list of the parameters for the instruction template, followed by some of the instruction slots with instantiated values.

In Figure 6-22 an example of the development of a plan for the *clear_counter* instruction is given. The internal instruction slot, *obj_list*, is initialised with a list of objects currently found on the *bar_counter*. The list is passed to *check_i* which examines each

of the items of the list. The *condition* required for *glass* in this case is "*used"* and, if successful in meeting it, the plan will proceed because the parameter *action* implicitly specified in the TL of *check_i* has a default value *proceed.* The first object of the list selected by *check_i* is *glass1* which condition is *used.* Once *check_i* is successful in finding the object (*glass1)* in the specified condition, *glass1* is passed back to the *clear_counter* instruction and bound to the *pick_up* and *put_i* instructions through the common parameter *glass* in their *p* slots.

Some instructions such as *pick_up* and *put_i* behave as data providers for their related tasks. In the case of the *pick_up* instruction the parameters are bound directly to the *pick_up_t* task. This is not the case for *put_i* because the place, *exc_counter*, is out of reach of the arm holding *glass1*. In this case, a plan that includes *approach_i* is chosen. Note that the second goal of the plan is another instance of *put_i*. This time, the evaluation of *put*_i concludes that the *arm* is within reach. The successful termination of *put_t* finishes with the *clear_counter* instruction.

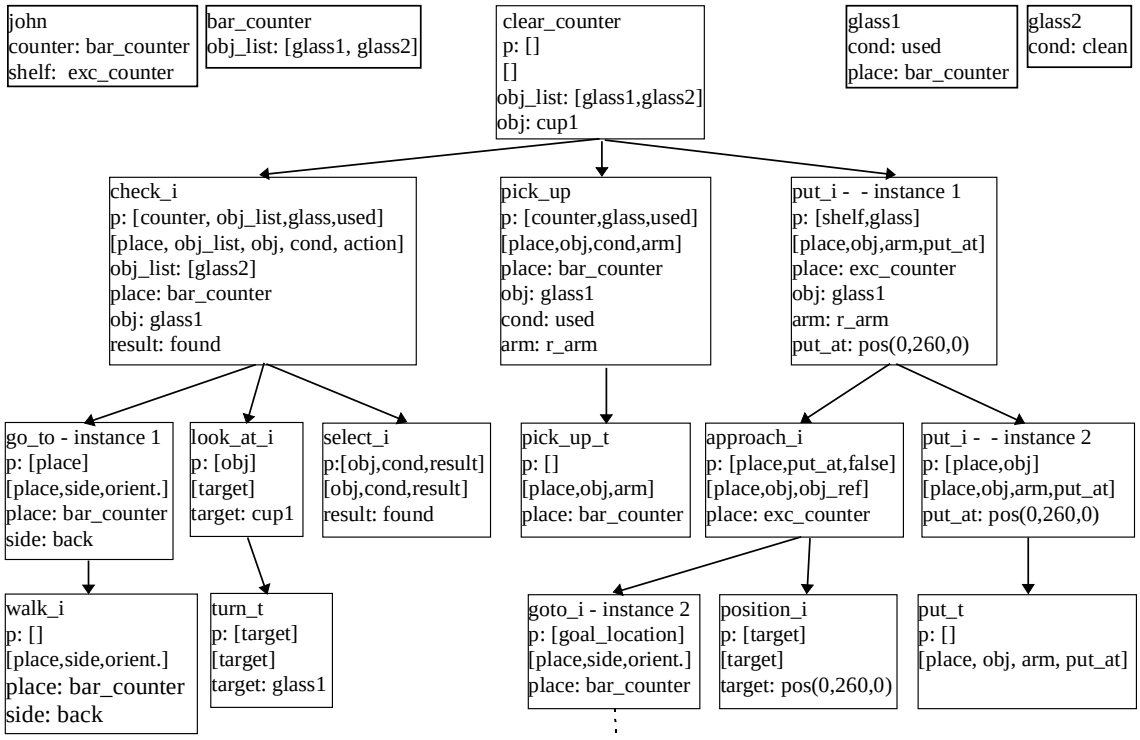

Figure 6-22: Developing an instance of *clear_counter* instruction.

6.17 Summary

The concept of the *instruction* is flexible since it permits features to be embedded which are found in conventional programming such as the case command (rule selection), the sequence of commands (plan of actions), procedure call (instruction instantiation and context identification), loop and recursive call (planning), etc. These simple constructs provide the elements for building an increasingly elaborate vocabulary specifying actions. Existing high-level languages such as Prolog provide the necessary tools for representing knowledge along with mechanisms of inference, symbolic manipulation, and the capability of interpreting the state of an environment.