KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Illustration of Artificial Intelligence Techniques in a Zoo Microworld

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-021, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/021/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/020/
Date of manuscript:	2010-12-19

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

Chapter 1

The Zoo Microworld

1.1 Introduction and Motivation

This memo describes a microworld that illustrates a number of important concepts and methods in Artificial Intelligence, and in particular in Knowledge Representation. The microworld consists of a simple zoo that is 'populated' with animals, a warden and other personnel, a number of vehicles, and physical structures that provide locations for, and resources for the people and the animals in the zoo. The Warden can perform a variety of actions, and some scenarios may allow other entities to be agents and to perform actions as well.

The Zoo Microworld System (ZMS) is organized as a basic *platform* and a number of *scenarios* that illustrate different A.I. techniques. The following are the presently implemented scenarios:

- The *Animal Care Scenario:* the Warden makes a daily tour to visit a number of animals in their stables, cages or other habitats in order to look after each of them. The tour is executed repeatedly, and on each execution there can be some unexpected events: an animal may be sick and require treatment, two animals that are in the same location may be in a fight and need to be separated, and so forth. The Warden has to decide on appropriate actions for each of the problems that he encounters.
- The *Diagnosis and Treatment Scenario:* similar to the previous one, but now a sick animal is only characterized in terms of observed symptoms, and the Warden must make a diagnosis to identify the most likely cause of the symptoms, and make a decision about appropriate treatment for that illness.
- The *Rainy Day Scenario:* this scenario actually only involves the Warden and not the animals. The road network in the Zoo has a rectangular, "Manhattan-style" structure, and the Warden can plan and carry out walks from one specific point in that road structure to another one. While walking he may get wet for a variety of reasons, including there being a rain shower, or passing by a point where a lawn sprinkler is operating. The Warden has a desire not to be wet, and if wet to be as little wet as possible, and he has means

of protecting himself (e.g. using an umbrella) and of getting dry (go inside a building, or staying the sun). The Warden also has a number of other desires for things to enjoy (eating an icecream) or to avoid (being stung by a wasp), and during the walk he will adapt his behavior in accordance with these desires.

These scenarios were first implemented as separate lab-assignment software in our course on Artificial Intelligence, where each lab built on the previous ones. They are now being integrated into one single system. The integration is not yet entirely complete, and the present document describes the Zoo Microworld Platform and the Rainy Day Scenario. The first two scenarios will be updated in due course so that they can be integrated with this system.

This Zoo Microworld System has been implemented in the Leonardo software system which is very well suited for this task. The present memo describes the Zoo Microworld Platform and the Rainy Day Scenario from several points of view, including

- The static model: animals, the places where they stay, the road structure, and so forth
- The dynamic model: actions that can be performed by the Warden and 'actions' performed by Nature
- The variant of the Belief-Desire-Intention behavior model that is used for the Rainy Day Scenario
- The reportoires defining commands for using the system, features for characterizing animals and other entities, predicates for characterizing static and dynamic aspects of the world, etc.

1.2 Static Aspects of the Zoo Microworld

The Zoo Microworld has a *static structure* that stays the same as time passes in this simulated world, and a *dynamic structure* that changes due to spontaneous changes in the world and due to actions that are performed by the Warden and by others. The static structure consists in turn of a *conceptual structure* and an *empirical structure.* The conceptual structure consists of analytical statements such as "giraffes are hoofed animals" whereas the empirical structure consists of statements such as "The chimpanzee court is located along route-4."

Some statements about class membership are clearly empirical, for example, "Rollo is in the class of animals having brown fur." For some other kinds of class membership statements it is debatable whether they are conceptual or empirical, for example "Rollo is a chimpanzee" or "Rollo is a male," but for simplicity we include all statements about classes and their members among the empirical ones.

1.2.1 Conceptual Structure: Types and Classes

The Zoo Microworld Platform defines the following *types* for entities that can be introduced in the various scenarios:

```
animal
personnel
building
route
food
medicine
tool
vehicle
```

Each scenario uses only some of these types, and some scenarios introduce additional types, but combined use of material from several scenarios is facilitated by having a standardized collection of commonly used types in the Platform.

The type `route` is used for roads and footpaths within the premises of the Zoo. The type `tool` is used for scissors, spades, scales and the like. The type `building` is used for stables, cages etc where the animals are kept, as well as for the administration building, the restaurant and other buildings for the personnel. (There is no type for visitors to the Zoo in the Platform).

In addition the following types of entities are used for classification or other static characterization of entities of the above types:

```
species
occupation
ailment
```

Entities in these types occur in attribute-values and may be associated with information in ways that are specific to each scenario. Each animal belongs to a particular species; each personnel has a particular occupation; ailments include diseases, wounds and other problems that may afflict animals and personnel.

Besides types, the model of the Zoo Microworld also contains a number of *classes.* Each class pertains to one specific type, and has a number of instances of that type as its members. For example, if `chimpanzee` is an entity of type `species,` and `Rollo` and `Lollo` are specific animals whose `type` is `animal` and whose `in-species` attribute is `chimpanzee,` then there may be a class `our-chimpanzees` that *pertains to* the type `animal` and that has `Rollo` and `Lollo` as its *members.* If `Rollo` and `Lollo` are the only chimpanzees in the microworld then this class can also be written using the following expression in Description Logic [1]

```
(those animal that in-species all {chimpanzee})
```

1.2.2 Physical Structures

Scenarios typically include the possibility of movement by personnel or by animals, which means that there must be some physical structure for the Zoo that defines the available locations, their properties and relationships. This subsection describes the currently used physical structures.

[1] This uses the CLE variant of the Description-Logic notation, with the additional proviso that the first argument of the `those` operation shall be a type and not a class.

The Locations of Animals

The simplest physical structure introduces a set of distinct *locations* and makes it possible to assign a location to each animal and to each personnel. These location assignments may change over time within each simulation of the Zoo microworld. Locations for animals may be specified as stables, cages, or other similar habitats; locations for personnel may in addition include buildings of various kinds and rooms within those buildings.

The definition of connections and distances between locations is specific for each scenario definition.

The Road Network in the Zoo

A slightly more complex physical structure defines a rectangular road network, containing roads in the east-west direction called *paths* and in the north-south direction called *trails.* Each path and trail has an integer number between 1 and some upper limit, presently set to be 6. Each intersection between path and trail is a *crossing* and is described in the obvious cartesian way as (trailnr, pathnr), so that the first component represents the x axis.

This physical structure is used for the Rainy Day Scenario. The following are some additional details in this model.

Each crossing is a roadpoint. In addition there are roadpoints halfway between the crossings but on the paths and trails, for example (3+, 2) and (3, 2+), but not (3+,2+) since that would not be on a road. These are all the roadpoints there are. The following diagram of roadpoints illustrates this structure:

```
    |             |
-- 3,4 -- 3+,4 -- 4,4 -- 4+,4 --  etc
    |             |
-- 3,3+          4,3+
    |             |
-- 3,3 -- 3+,3 -- 4,3 -- 4+,3 -- etc
    |             |
```

It follows that when the warden is at a crossing he has four possible moves, namely north, south, east and west, and each of these will take him to a roadpoint that is not a crossing. Non-crossing roadpoints only allow two possible moves, in the obvious way.

The Platform contains a simple route planner that can construct a sequence of moves that will take the warden from his present position to a given new position. The route plan is a sequence where the elements are north, south, east, or west.

1.3 Dynamical Aspects of the Zoo Microworld

The dynamical aspects of the Zoo Microworld use the common notation for actions and change using first-order logic with explicit discrete time, including the use of features, the `H` and `D` predicates, and so forth. This formalism has been introduced in the lecture-note on actions and change in our set of

lecture notes. The present chapter adds notation and conventions that are specific to the Zoo Microworld, and also repeats some of the material from the earlier lecture note.

1.3.1 Time, Episodes and Situations

The basic facility in the Zoo Microworld Platform is a simulator that makes it possible to obtain simulated *episodes* in the Zoo Microworld. Each episode is organized as a sequence of *timepoints* which are represented as integers from 0 and up. Each timepoint in the episode is associated with information about the values of certain *features* at that time. Each episode also contains information about the *actions* that have occurred or are presently occurring in the episode. Actions have a *starting time* and an *ending time* which must be timepoints, and the ending time must be strictly larger than the starting time.

At each point in the simulation of an episode there is a *current time* for the simulation. The state of the simulated world is saved at each timepoint (or more precisely, the changes from the previous timepoint are recorded) so that it is possible to inquire about the state of the simulation at earlier timepoints.

Within a computational *session,* i.e. a run of the system in a computer, it is possible to initiate and perform several episodes. This is useful e.g. for demonstration purposes: the recorded history of an individual episode can be extracted and saved on a file for later inspection and publication.

At a given point of a simulation where the current time is `.t` it is possible to construct *successor situations.* These are hypothetical future states of the world that will be obtained, according to the agent's prediction, if particular actions are being performed. If `a1, a2,` and `a3` are actions then one can construct, for example,

```
(succ .t a1)
(succ .t a2)
(succ (succ .t a1) a2)
```

and so forth. Thus, the current time is a situation, and for every combination of a situation `.s` and an action `.a` there is a specific situation which is written as (succ .s .a). The Platform contains the facility for constructing a sequence of situations corresponding to a given sequence of actions, and to calculate the changes of feature values in the successive situations in that sequence.

1.3.2 Features

The dynamic structure is represented using *features* that are assigned a value at each point in time using the `H` predicate. The Platform provides only one way of constructing features, namely using the `the` function, like in

```
(the fur-color of Rollo)
```

for example. This function has two arguments, namely `fur-color` and `Rollo` in the example; the word `of` is only inserted for readability. The

first argument shall be an attribute whose value can change over time; the second argument shall be an entity whose type is such that it may have that attribute.

The following are features that are defined in the Platform for possible use in several scenarios, with an indication of the required type for the carrier of the attribute in each case:

```
(the location of /personnel)
(the location of /animal)
(the hunger of /animal)
(the thirst of /animal)
(the ailments of /animal)
```

This is merely the beginning of the list; additions to be made.

1.3.3 Actions

Each scenario contains definitions of a set of verbs that are used in that scenario. An action is an expression consisting of a verb and appropriate arguments and/or parameters for the verb, using the syntax of the CEL notation. Most actions have a parameter with the tag `by` that indicates the *subject* of the action, as in the following examples:

```
[eat :obj banana-4 :by Rollo]
[pass-overhead :by airplane-4]
```

or, using the slash notation for specifying the required type of the parameter:

```
[move-to :by /personnel :obj /animal :to /location]
```

There are a few examples of actions that do not have any `by` parameter ("natural actions"), such as

```
[startrain]
```

Actions that are performed by the Warden of the Zoo (or the primary warden if there are several of them) have the entity `TheWarden` as the value of the `by` parameter.

It may be argued that natural actions and actions having an inanimate subject (for example "the stone falls") ought to be called events rather than actions. However we stay with the term "action" since the term "events" has another meaning in the BDI terminology, and since that meaning is also widely used.

1.3.4 Duration of Actions and Features

Long and Short Actions

Some of the scenarios use a distinction between *long* and *short* actions. A long action consists of several *steps* that are normally performed in sequence, but it is possible to intersperse one or more short actions between successive steps in a long action, or even perform them concurrently with long-action steps. The most important type of long action is the *promenade* where the Warden goes from one waypoint to another, or where he makes visits to

a number of animals in their different locations. Short actions during a promenade may include curing an animal, taking out an umbrella, scaring away a wasp, or answering a question by a customer.

The distinction between long and short actions is useful since it allows one to define an agent behavior where the agent acts towards an overall goal, namely, the goal of the current long action, and where anyway it is able to adapt his behavior to the situations that are encountered. It is a simple and not very flexible solution to that problem, and more sophisticated approaches are certainly possible, for example using an HTN architecture (Hierarchical Task Networks), but it is sufficient for illustration purposes. In particular, the Rainy Day Scenario uses long and short actions and uses a BDI behavior model (Belief-Desire-Intention) for the choice of short actions between successive steps in a long action.

Persistent and Transient Features

Some features are *persistent* in the sense that they retain their value until the value is changed for some specific reason, which is usually due to the occurrence of an action; most actions have the effect of changing the value(s) of some feature(s). Other features are *transient* in the sense that they have a normal value, and if they are changed to another value at some point in time then they return spontaneously to the normal value, usually already in the next timepoint. We refer to *transient values* and *stable values* for transient features. For persistent features all values are stable.

Persistent attributes can be used for representing phenomena that we think of as actions but where there are no distinguishable "steps" like for long actions as described above. For example, the statement that the Warden carries a particular bag for a period of time may be represented by using an attributes `carries` for the entity `TheWarden` where the value is an entity such as `TheBag`, or a set of entities if the Warden can carry more than one thing. There will then be short actions for representing that the Warden picks up the bag and puts down the bag, that is, for the beginning and ending of the natural-language action of the Warden carrying the bag.

1.3.5 Basic Verb Vocabulary

Although each scenario uses its own collection of verbs, there is a definition of basic verb names in the Zoo Microworld Platform, in order to facilitate the interoperability between different scenarios.

We distinguish between actions that are performed by Zoo personnel (and therefore by people), actions that are performed by animals, and a few other groups. The following is a very preliminary list of such verbs, due to be discussed with several users.

The following verbs are tentatively proposed for animal actions:

```
eat
drink
bark/ cackle/ ...
sleep
bathe
```

```
swim
point at
give birth
breastfeed
begin or end doing some of the above (including wake up, fall asleep)
```

The following verbs are tentatively proposed for personnel actions with an animal as the object:

```
cause animal to do (some of the above)
assist animal in doing (some of the above)
move
wash
delice        (= remove lice from)
wrap in sheets   (for warmth)


separate      (two or more animals, in case of fight)

take temperature
clean wound
remove nails/...
inoculate     (= vaccinate)
chain

give name
take picture of

kill
```

The following are tentatively proposed verbs for actions done by personnel to the physical premises of animals

```
clean up
wash floor
turn on/ turn off light
open/close door
```

The following are tentatively proposed verbs for personnel that they can perform while moving around, using the basic `goto` or `do-promenade` long actions:

```
put-down bag
pick-up bag
enter building
leave building
take out object from bag  (e.g. umbrella)
put back object into bag
open/close umbrella
chase away insect
take on/ take off raincoat/ shoes/ etc.
```

The following are tentatively proposed operations done to a third party

```
post picture of ... on zoo website
include picture/name/description of ... in zoo brochure
send picture of ... to newspaper/ other interested party
```

```
put nameplate of ... at yard
remove the above
```

1.4 Predicates

The standard `H, Hc` and `D` predicates have already been mentioned; these are used for characterizing the dynamic aspects of a scenario in the microworld, including for preconditions and effect rules for verbs, and for the conditions that occur in behavior rules.

The static aspects require a number of additional predicates, and there is also a need for certain functions for use together with `H` and `D` in the dynamic aspects. These predicates and functions are as follows.

1.4.1 Static Predicates

`[equal .x .y]` for equality

`[attrib-is .e .a .v]` is true iff the entity `.a` has the value `.v` for the attribute `.a`

1.4.2 Logic Operators for the Dynamic Aspects

`(the .a of .e)` constructs a feature, where `.a` is an attribute and `.e` is an arbitrary entity.

`(succ .s .a)` constructs the situation that is the successor of the situation `.s` after having executed the action `.a`

`(val .s .f)` obtains the value of the feature `.f` in the situation or timepoint `.s`

`(curval .f)` obtains the value of the feature `.f` in the current timepoint

`[sameval .f .g]` is true iff the features `.f` and `.g` have the same value in the current timepoint

The predicates and functions of the Common Expression Language may also be used.

Chapter 2

Behaviors in an Intelligent Agent

The Leonardo software system contains a facility for defining preconditions of actions. The Zoo Microworld System extends this with a facility for defining nontrivial responses to precondition failures. [1]

In addition the Zoo Microworld System contains a facility for defining goal-directed agent behavior using a variety of the Belief-Desire-Intention (BDI) behavior model which has already been described in the lecture note on intelligent agents.

The purpose of the present chapter is to define these facilities in a general form. Chapter 3 will then describe how the BDI facility is used for one particular scenario, namely, the Rainy Day Scenario.

2.1 Response to Precondition Failure

Each action in the Zoo Microworld is defined in terms of its *formal precondition* and its *performance script.* The precondition specifies conditions that must be satisfied in order for the action to be performable; the performance script specifies how to execute an action when its formal precondition is satisfied.

The formal precondition need not be exhaustive or, more precisely, it is not always possible to make it exhaustive, so it may happen that the execution of the performance script *fails* even though the formal precondition was satisfied. However, formal preconditions are anyway useful for rational behavior since they can be used for planning and for prediction. One is entitled to consider it as a default that the action succeeds if its formal precondition is satisfied.

If an agent has the intention to perform a particular action, for example due to a command by the user, or because it is part of a plan that the agent has decided to perform, and if the formal precondition of that action is

[1] This facility will be included in forthcoming versions of the general Leonardo system.

not satisfied, then there are two possible courses of action. One possibility is for the agent to first make some other actions that *achieve* the formal precondition, that is, they make it come true, and then to perform the intended action. The other possibility is to retract the intention to perform the action in question and to consider some *substitute* action. Doing nothing i.e. dropping the intention altogether is one possible substitute action.

In particular, if the action in question has been selected as a step towards a larger goal, then there may be some other way of achieving that goal. In this case the remaining sequence of actions towards that goal can be replaced by another such sequence, i.e. by another plan.

Preconditions are expressed as a proposition (i.e. as a logic formula) that is associated with each verb. The violation of a precondition is typically identified as a ground literal, that is, as a predicate symbol with its arguments, not using any variables, and possibly with a negation. Each predicate symbol is associated with one or more *methods* that may be used for *achieving* such a literal, that is, for making it come true.

The considerations whether to try to achieve a failed precondition or to select a substitute action or plan are included in the overall behavior definition. In principle it would be possible to use the BDI behavior model for this purpose, but this has not been realized or tested yet.

2.2 BDI Behavior Model

The BDI behavior model is described in the Compendium on Programming Techniques for Intelligent Autonomous Agents. The BDI implementation in the Zoo Microworld Platform follows the general BDI design that was described there, but some additional aspects must be added in order to make it viable, and they will be described here.

The BDI behavior model in the Platform is based on the distinction between long and short actions. Long actions are initiated by commands from the user, and are not obtained from the BDI model; this model is only used for selecting short actions that can be inserted in the sequence of steps that constitute a long action.

In the design of using a distinction between long and short actions it is natural to introduce a facility whereby a short action may change some aspects of the long action within which it is being executed, for example, by changing the route of walking. Such a possibility has however not (yet) been included in our system.

2.2.1 Desirability Rules and Opportunity Rules

The concept of a *desire* is fundamental in the BDI Behavior Model. Desires can be thought of as policies: they are continously monitored while the agent operates, and instances of these desires guide the selection of intentions and goals. In the present variant of the model, desires are defined in terms of the values of features: desires specify which feature values are desirable and which are non-desirable.

We have defined above how feature values may be persistent or transient, and this influences how desires relate to them. The following cases are of interest:

- *Feel good momentarily:* the agent has a desire that a particular, transient feature value or transient combination of feature values shall occur.
- *Feel bad momentarily:* the agent has a desire that a particular, transient feature value or transient combination of feature values shall not occur.
- *Feel good persistently:* the agent has a desire to establish and maintain a particular, persistent feature value, or the default value of a transient feature.
- *Feel bad persistently:* the agent has a desire to avoid, or to minimize the extent of a particular, persistent feature value, or the default value of a transient feature.

These are purely qualitative considerations. In addition there may be desires to maximize or to minimize the values of certain numerical-valued features.

In line with these distinctions the following are the desired functionalities in the agent's behavior system:

- *Detect transient opportunity:* React to observations that suggest that using some short action the agent will be able to achieve an instance of *Feel good momentarily.*
- *Detect transient problem:* React to observations suggesting that without additional action by the agent there will be some forthcoming instance of *Feel bad momentarily.* (Notice that if the agent does not do any prediction of the near future so that the feel bad momentarily has already occurred, then there is nothing the agent can do about it).
- *Detect feel good persistently:* React to observations that suggest that using some short action the agent will be able to achieve an instance of *Feel good persistently.*
- *Retain feel good persistently:* React to observations that suggest that without additional action the current instance of feel good persistently is going to end. (This may require prediction).
- *Detect feel bad persistently:* React to observations that suggest that without additional action there will be an instance of feel bad persistently. (This case also requires prediction).
- *Discontinue feel bad persistently:* React to observations that suggest that using some short action the agent will be able to cause a current instance of *Feel bad persistently* to end.

In addition there are quantitative counterparts of some of these:

- *Maximize feel good persistently:* During a 'feel good' period of time, perform additional action so as to increase the value of the attributes representing the comfort level.

- *Minimize feel bad persistently:* During a 'feel bad' period, perform additional action so as to decrease the value of the attributes representing the level of discomfort, even if it is not possible to reduce it to zero.

These functionalities can in fact be implemented using one single technique with relatively minor variations. The basic idea is to use conditions that are evaluated repeatedly during the execution of the system, and typically between successive steps of the current long action. This is immediately useful for dealing with cases of *feel bad persistently.* In this case there is a *desirable condition* that is written so that it is true if the agent "feels good" in this respect and false if it "feels bad." When the desirable condition is false then some action must be taken if possible, in order to discontinue the persistent "feel bad" condition.

For example, the agent being thirsty is a persistent "feel bad" condition, the desirable condition is written as an expression that is true if the agent is not thirsty, and false if it is thirsty. If the desirable condition becomes false during the simulation of an episode then the agent shall seek a corrective action, such as drinking a glass of water.

There are two other negative functionalities, namely *detect transient problem* and *detect feel bad persistently.* These can also be handled using desirable conditions that are used to indicate a problem that needs to be acted on, but in these cases the desirable conditions must be applied to a *predicted future situation.* If the momentary "feel bad" is already occurring then there is no reason to do anything, since it is transient anyway, and if it is persistent and has already started then the need is to discontinue it since it is too late to prevent it from starting.

The case of *detect transient opportunity* is the simplest one of the positive functionalities. A very general way of implementing it would be to always (i.e. at each timepoint, or always between two successive steps of a long action) consider all possible short sequences of short actions, to predict their effects, and to check whether any of them lead to achieving an instance of *feel good momentarily.* This is however not a realistic method, and it is more reasonable to use *opportunity conditions* that indicate that a transient opportunity is within reach. For example, there could be an opportunity condition that evaluates to true if the agent finds itself beside an icecream stand, and that is associated with a method for obtaining an icecream, such as purchasing one.

Opportunity conditions are similar to desire conditions but there are two differences. First, desire conditions trigger an action if they evaluate to false whereas with the natural way of writing opportunity conditions they will trigger an action when they evaluate to true. However for uniformity of processing it is convenient to reverse the sign for opportunity conditions, so that they evaluate to true when there is an absence of opportunity, and to false when there is an opportunity.

Secondly, when a desire condition evaluates to false then the associated methods are methods that will make the same condition come true again, but the methods for opportunity conditions are designed so as to make some other condition come true, namely, an instance of a desire. For example, the persistent condition of being beside an icecream stand is not a goal in itself, but it may bring to mind a method that has the enjoyable although transient

effect of eating an icecream bar. A general design that covers both cases is therefore one where a *behavior rule* has three components: a *rule condition,* a method or a set of methods, and a *purpose condition* that the method is supposed to achieve. The first and the third component are equal for *desirability rules* and different for *opportunity rules.* It remains to consider the two other kinds of positive functionalities. The case of *detect feel good persistently* can be treated in the same way as *detect transient opportunity,* with or without prediction of future states. Finally, the case of *retain feel good persistently* is concerned with identifying situations where an ongoing feel-good state will be discontinued unless an action is taken. Here again we will have a behavior rule where the first and the third condition is equal: if this condition is false in a predicted future state then corrective action is needed in order to make sure that it actually retains the value true.

The quantitative-oriented behavior rules for *maximize feel good* and *minimize feel bad* can likewise be represented using these three components, and also in this case the purpose condition is different from the rule condition, since it characterizes the expected level of comfort or discomfort as a result of applying the method.

Sometimes, but not always one can use the same method for several of the cases described above. For example, becoming wet as a result of being out in the rain is a continous-valued discomfort, and opening an umbrella is useful both for avoiding becoming wet, and for not becoming more wet when one has already become somewhat wet. As a contrary example, however, the methods for avoiding being stung by poisonous jellyfish are different from the methods for reducing the pain when one has already been stung.

2.2.2 Applicability Tests and Choice of Method

Both kinds of behavior rules specify a rule condition that is evaluated regularly, and one or more methods that it may be appropriate to apply if the rule condition is violated. Usually, each method consists of both an applicability condition and a sequence of actions, and the applicability condition must be satisfied before one can consider using the method. The applicability condition is similar to the preconditions of ordinary actions. If the behavior rule contains several methods then it may be that only some of them pass their applicability tests.

There are also other ways of defining the choice of method, for example by merely specifying the purpose condition explicitly and leaving it as a planning task to find a sequence of actions that will achieve the purpose condition in a particular situation.

If several methods pass their applicability tests then the system must choose which of them to use. Simple choice criteria include making a random choice, or associating a goodness value with each method and picking the method with the best goodness value among those that pass their applicability tests.

A more refined way of making the choice is to predict the result of performing each method separately, and then evaluating the outcomes so as to pick the one with the preferred result. This can be done by assigning a numerical merit to each of them, using a combination of merit components that capture different preferences by the agent. The system's body of 'desires' shall

be used as a basis for assigning the merit numbers, and for quantitatively expressed desires the contribution to the overall score may be obtained from the level of comfort or discomfort in the attributes that are used.

Since both transient and persistent feature-values must be accounted for, it is necessary to evaluate the successive states in a predicted sequence of future states, and not merely the last one of them.

Another possibility is to use a *comparison predicate* that will indicate which of two alternatives one shall prefer, using other methods than assigning a numerical score to each of them. A comparison predicate may for example be defined using a decision tree.

2.2.3 Follow-up of Behavior Rules

Once a method has been selected and applied, including the "method" of doing nothing as one of the alternatives, it is also important to perform *follow-up,* that is, to check whether the expected outcome has in fact been obtained. This follow-up operation uses the third component of the behavior rule. For simple desirability rules the follow-up consists of verifying that the intended feel-good effect has been achieved; for opportunity rules it consists of verifying whether the target opportunity did in fact materialize.

The outcome of the follow-up can be used for both short-term and long-term purposes by the system. The short-term use is to control whether to try again or to resign. If the desired feel-good condition is not achieved then the system may decide to try again using the same method, or using another method, but it may also decide that it is not worth trying and that it will simply accept this particular not-feel-good situation until it encounters an opportunity for discontinuing it, that is, a method for *discontinue feel bad persistently* whose applicability condition is satisfied.

The potential long-term use of follow-up is to modify the entire structure of behavior rules and method evaluation rules. Techniques for doing this belong to the area of *case-based reasoning* and are beyond the scope of the present note.

2.2.4 Revised Top-Level BDI Loop

The standard definition of the generic BDI Main Cycle is as follows:

```
initialize-state
repeat
   options := option-generator(event-queue)
   selected-options := deliberate(options)
   update-intentions(selected-options)
   execute()
   get-new-external-events()
   drop-unsuccessful-attitudes()
   drop-impossible-attitudes()
end repeat
```

This definition is realized as follows in the case of the Zoo Microworld System (ZMS).

The BDI operation `initialize-state` is performed by the ZMS operation `startlab.` Since ZMS uses the BDI machinery for deciding which short actions, if any, are to be done between successive steps in a long action such as `goto` , the `repeat` loop in the BDI definition is implemented as a ZMS function `decide-short-actions` that is invoked in the definition of `goto` and any other long action. Before the invocation of `decide-short-actions` the long action shall perform one step in its own operation, including the side-effects of that step. For example, the definition of `goto` in the Rainy Day Scenario will move the Warden from one roadpoint to the next according to the chosen itinerary; it will also change the location of anything that the Warden is carrying to the Warden's new location, and it will update the Warden's level of wetness according to whether it is raining, whether the Warden is carrying an umbrella, and other relevant circumstances.

Deciding on Short Actions

After having thus updated the state of the world, the long action invokes the function `decide-short-actions` that performs exactly once the body of the loop shown above. It executes the following steps in sequence.

1. Make a loop over the list of available behavior rules and evaluate their rule conditions. If the condition is false then this returns an instance of the condition that falsifies it. If this is the case then add the pair of the identifier for the behavior rule and the falsifying instance to the set `options.` Moreover, if the condition is true (that is, the desire is satisfied) then add the identifier for the behavior rule to the set `autoachieved-options` (for use in a later step).

2. Make a loop over the set `options` . For each option, if the option is a member of the set `*resigned-intentions*` then do nothing. Otherwise, make an inner loop over the set of known methods for the behavior rule in the option and evaluate the method's applicability condition. If this condition is satisfied then add the method to the set `methods-here` that becomes the result of the inner loop. If this set is empty at the end of the inner loop then add the falsifying instance to the set `*resigned-intentions*` - this has the effect that this falsifying instance will be ignored in subsequent invocations of `decide-short-actions,` i.e. the agent has given up on it.

If the set `methods-here` is not empty, on the other hand, then invoke the function `evaluate-methods` on it. This function shall return one of the methods which is then selected for further use, or the symbol `nil` if it is considered that none of the methods is appropriate. If the returned value is not `nil` then add the triple consisting of the identifier for the behavior rule, the falsifying instance, and the identifier of the method to the set `selected-options.`

3. For each member of the set `autoachieved-options` that is updated in step 1, if this member is represented in the set `*resigned-intentions*` then remove it from there. The set `autoachieved-options` is initialized to the empty set at the beginning of each invocation of `decide-short-actions.`

4. At this point it would be appropriate for the procedure to check that the elements in `selected-options` are consistent and can function well

together. However, at present the implementation does not do this, and the set `selected-options` proceeds directly to the next step.

5. For each element in `selected-options,` execute the script that is associated with the identifier for the method, together with the variable bindings that are obtained from the falsifying instance of the condition in the behavior rule. After each script execution, evaluate the purpose condition of the behavior rule. (Please recall that for desirability rules this is the same as the rule condition, and for opportunity rules it is a separate condition). If the purpose condition is satisfied then the use of the rule was successful, otherwise not. Report this to the user of the system using the command-line dialogue. Also, if the condition was not satisfied in the case of a desirability rule, then add the falsifying instance to the set `*resigned-intentions*`

It is clear how these steps correspond to the steps in the idealized top-level loop. Notice that the BDI step `get-new-external-events()` does not have any counterpart since the ZMS does not make any distinction between the simulated world and the agent's knowledge of the world. Therefore, the update of the world state that is done by the long action is immediately available to the procedure defined here.

Evaluation and Selection of Methods

The function `evaluate-methods` does the following. For the given set of methods, it first of all adds the null method consisting of doing nothing at all. It then identifies a sequence of short actions consisting of the next few intended steps in the long action within which the function is being called. For each of the methods, it constructs a sequence of actions consisting of the action sequence of the method itself, followed by the identified forthcoming long-action steps. The effects of performing this sequence of actions are calculated by constructing the successive situations (using the `succ` function) and deducing the feature values that apply to each of them.

The merit value is calculated for each of these situation sequences, and the method (including the empty method) has obtains the highest score is selected for use; the identifier for that method is returned.

Several methods are possible for calculating the feature values in constructed future situations. The present implementation uses a simple (incomplete) resolution theorem prover together with effect laws for each of the verbs. Indirect effects are calculated using the same procedure as is used by the long verbs.

The merit value for a situation sequence is calculated as the sum of contributions from a number of specific *merit components,* each of which is related to some of the desirability rules.

2.2.5 Urgency Levels

The implementation defines a *level of urgency* for each behavior rule. Rules with *medium urgency* are used for identifying short actions at successive steps of a long action, and this is the only level that is in active use at present. The level of *low urgency* is intended to be used for behavior rules that allow the system to decide autonomously about its next long action.

Likewise, the level of *high urgency* is intended for stimulus-response rules where the failure of the rule condition results in immediate execution of a method without first considering alternative methods or the effects of the designated method.

Chapter 3

The Rainy Day Scenario

The general design in the previous chapter can be better understood by also considering a concrete example. We shall do so using the Rainy Day Scenario which has been the first use of BDI facility in the Zoo Microworld Platform.

3.1 Actions and Laws of Change

This Rainy Day Scenario uses the road network model of the Zoo that was described above. The sole agent in this scenario is a warden which is represented by the entity `TheWarden` and which may move from one roadpoint to an adjacent one in one timestep. A sequence of such moves constitutes a promenade, which is a long action and which can be requested from the command loop using the verb `goto`.

3.1.1 Simple Short Actions

A few short verbs are introduced in order to obtain a basis for behavior rules in the BDI machinery. They refer to actions that are performed by the Warden, and are defined in terms of appropriate features for each verb. For example, there is the `pickup-coin` action which is possible if the Warden is at a location, i.e. at a roadpoint where there is a coin on the ground. The effect of performing the action is that there is no longer a coin on the ground there, and the Warden has a transient positive value for a "feel-good" feature.

The presence of a coin on the ground at a particular roadpoint p is represented by the attribute `has-on-ground` whose value is a set that may or may not contain the entity `coin`. The transient attribute for the Warden is called `just-received` and it may have the value `coin` or `none`, or some other alternative values that are obtained from other short actions.

Two things should be noted in this example. First, the behavior rule that suggests that the Warden may use the `pickup-coin` action in order to feel good is triggered by the Warden passing by a particular roadpoint. Both for this example and in general, the way to prepare for an episode in this scenario is to equip the road network with entities in suitable locations, such

as the coin on the ground in this example. After the road network has been suitably equipped one can initiate a `goto` action where the agent passes by anticipated locations and takes appropriate action at some of them. On the other hand there is no command for command-line use that will cause specific changes to be made at specific timepoints in a forthcoming episode.

Secondly, the reward in this example is represented as a transient feature value. It may be objected that when the Warden has picked up the coin then he will keep it until something happens with it, and he will have an advantage of that all the time. However, given that there are no mechanisms whereby the agent may lose the coin, there is not going to be any need for rules of the form *retain feel good persistently* in this respect, so the effect may just as well be represented as transient.

The use of the `just-received` feature suggests an opportunity rule of the type *detect transient opportunity.* A second short verb in this scenario is `buy-icecream` which is possible when the agent is in a location where there is a building of type `icecream-parlor` . It is represented similarly using a transient feature and an opportunity rule for *detect transient opportunity.*

3.1.2 Rain-Related Short Actions

Besides these simple short actions, there is also a group of actions that are introduced in order to have somewhat more complex developments, and that contribute to a desirability rule that works for both *detect* and *discontiue feel bad persistently.* These actions are based on the notion that the warden can get wet, namely, if it is raining, or if he passes by a roadpoint where a grass sprinkler is running. Raining is a persistent feature of the Nature object, so it may start and stop raining at specific timepoints. The `startrain` and `stoprain` actions are requested in the command-line dialog and can not be controlled or even predicted by the Warden. Notice that these actions can only be executed between promenades, and not within them, since they can only be invoked from the command-line dialog.

The sprinklers also do not turn on or turn off during a promenade, so at some roadpoints there is a sprinkler and it runs all the time.

While performing his moves the Warden may or may not carry a particular bag, called `TheBag.` There are actions whereby he puts the bag on the ground beside him, or picks it up. If he has picked up the bag then it comes with him as he moves to a new roadpoint, but if it is on the ground then it stays there.

One more artifact is included in the scenario, namely an umbrella that may be either in the bag, or held in the warden's hand. If the warden gets in the way of a sprinkler or of rain then he will get wet, except if he has taken out his umbrella. The warden does not like to get wet, so he may choose to take out his umbrella when rain occurs or he gets in the way of sprinkling.

However, to complicate matters, in order to take out the umbrella from the bag, he must first put down the bag on the ground, then take out the umbrella, and then pick up the bag again if he wishes to continue carrying it. While performing these operations he will get increasingly wet. Therefore, if he is already in the way of a sprinkler, it is best to continue walking through it and not stop to take out the umbrella. If it is raining then the reverse is true, unless the agent is already soaked.

Another way for the warden to avoid getting wet or to dry up is to go inside a building. However, then he must be at, or go to a roadpoint where there is a building. Well inside a building he may wait a while so as to become dry again.

Being wet is not a binary condition, therefore; the warden's wetness is defined on a scale from 0 to 10, where 0 means entirely dry. Each time unit where the warden is exposed to rain or sprinkling his wetness increases by 2; every time unit where it is not raining or he is inside a building the wetness decreases by 1, all within the interval from 0 to 10. For these purposes, a time unit is when the warden moves from one roadpoint to the next. The actions where the agent puts down the bag, picks up the bag, gets out the umbrella, or puts it back count as half a time unit for the purpose of getting more wet, i.e. wetness increases with 1 if it increases, but it does not decrease in any case. Moreover there is a 'wait' action where nothing happens except that the warden's wetness changes according to the rules (increases by 2 or decreases by 1).

The obvious condition that the Warden does not want to get wet is represented formally by a desire of having wetness level 0.

3.2 Implementation of Behaviors

In this section we shall describe the formal definitions that are used in order to implement the motivational structure described above. The purpose of this section is not to be like a software documentation, but merely to give the reader an idea of the size, complexity and character of the definitions that are needed.

3.2.1 Attributes

Attributes are persistent unless otherwise noted. The following attributes are used for `TheWarden`

`location`	`Indicates the location of the warden as a composite entity, e.g. (crossing: 2 3)`
`howmuch-wet`	`Number indicating the wetness of the warden on a scale from 0 to 10`
`open-umbrella`	`Indicates whether the warden has opened up his umbrella, value yes or no`
`carries-bag`	`Indicates whether the warden carries the designated bag, value TheBag or no`
`is-inside`	`indicates whether the warden is inside a building, value yes or no`
`just-received`	`As described above, value coin or no. This is a transient attribute whose stable value is nil`
`just-enjoyed`	`Value icecream if TheWarden just had one. Transient attribute, value icecream or no`

The following attributes are used for `Nature`

`raining`	`Value yes or no`

The following attributes are used for `TheBag`

`onground`	`Value yes or no, indicates whether the bag is standing on the ground or being carried`

The following attributes are used for roadpoints

`has-building`	`There is a building at the roadpoint. The value is the type of building, e.g. icecream-parlor, and otherwise nil`
`has-sprinkler`	`There is a lawn sprinkler at the roadpoint. Value yes or no`
`has-on-ground`	`The value is a set of entities which may include e.g. the entity coin. The empty set may be represented as nil`

3.2.2 Verbs

The following short verbs are defined in the Rainy Day Scenario and for use by the Warden.

`buy-icecream`	`the Warden buys an icecream which leads to transient gratification. It can only be done if there is an icecream parlor at the Warden's current location`
`pickup-coin`	`the Warden picks up a coin that is on the ground in the Warden's current location`
`pickup-bag`	`the Warden picks up the Bag that must then be standing on the ground at the same location as the Warden`
`putdown-bag`	`the Warden puts the Bag on the ground`
`takeout`	`the Warden takes out the umbrella from the bag that must be standing on the ground`
`putback`	`the Warden puts back the umbrella into the bag that must be standing on the ground`
`go-in`	`the Warden enters a building that is at the same roadpoint as where he is located`
`go-out`	`the Warden leaves a building where he is at the present time`
`wait`	`the Warden does nothing during one timestep`
`wait-until-dry`	`the Warden stays inside a building until his wetness has been reduced to 0`

3.2.3 Desires and Methods

The following are a few examples of how behavior rules and methods are written. First, a trivially simple example of a an opportunity-rule saying that if the Warden is at a location where there is a building of type `icecream-parlor` then he will consider buying an ice-cream.

```
------------------------------------------------------------
-- opporule-1

[: type opportunity-rule]
```

```
[: has-methods <get-icecream>]

@Rulecond
[-Hc (the has-building of (get TheWarden location)) icecream-parlor]

@Purpose
[true]

-----------------------------------------------------------
-- get-icecream

[: type behavior-method]

@Requires
[true]

@Has-script
[buy-icecream]

-----------------------------------------------------------
```

The `Rulecond` property contains the rule condition. At a step during a promenade where it is false, the system will take this as an option, in the sense of the BDI behavior model, and consider each of the methods in the sequence (or set) that is the value of the `has-methods` attribute as a way of dealing with this option. In this simple example there is only one such method. For each method it will check whether the expression in the method's `Requires` property is true. If it is then it will consider this method as a candidate for the dealing with the option. In this case there is only one method that passes this test, so there is no need to choose between methods. The method's script is simply to perform the action `[buy-icecream]` which is therefore performed. After performing it, the agent evaluates the expression in the rule's `Purpose` property in order to check whether the intended purpose of performing the method has been met. The value of the `Purpose` expression is not used for anything in particular in the present system, but in a future development it may be used for revising the agent's behavior rules and the priorities of these rules.

We proceed to a slightly more complex example, namely, the definition of the desirability rule for not getting wet and its associated methods.

```
-----------------------------------------------------------
-- desirule-1

[: type desirability-rule]
[: has-urgency medium]
[: has-methods <open-umbrella dry-in-building>]

@Rulecond
[Hc (the howmuch-wet of TheWarden) 0]

-----------------------------------------------------------
-- open-umbrella

[: type behavior-method]
```

```
@Requires
(and [Hc (the open-umbrella of TheWarden) no]
     [Hc (the carries-bag of TheWarden) TheBag] )

@Has-script
[soact [putdown-bag][takeout][pickup-bag] ]

-----------------------------------------------------------
-- dry-in-building

[: type behavior-method]

@Requires
(and [Hc (the is-inside of TheWarden) no]
     [-attrib-is (get TheWarden location) has-building nil] )

@Has-script
[soact [go-in][wait-until-dry][go-out]]
```

In these definitions there is a desirability rule, called `desirule-1` which represents the desire for the warden not to get wet. It is specified by a logical expression for the desire, in the `Rulecond` property, and this behavior rule refers to two possible methods. Each method is specified by a `Requires` formula and the `Has-script` formula like in the previous example.

This is basically what is needed in order to define such desires and methods for achieving the instantiated goals. The following other things are also needed:

- Definitions of the verbs that are used for performing actions and achieving goals.
- Definitions for the machinery for instantiating desires and administrating the resulting goals. This is in the entityfile `motivsys` and requires around 420 lines of code in the present lab materials, which means it can be done quite compactly.
- Definitions for how to do prediction of the results of proposed scripts. This uses the implementation of a simplified resolution theorem-prover.

The following are the important points that can be seen in this simple example and that we wish to illustrate with it:

- The simple desire-driven and goal-directed behavior can be implemented using a very moderate amount of code, combined with a straightforward use of standard first-order logic and of the formalism for reasoning about actions.
- The behavior that results from the definitions in the example is quite simple, but considerably more sophisticated behavior can be implemented by adding more 'knowledge' in the form of logic formulas, and without the need for a lot of additional programming.

Chapter 4

Operating Instructions

The following are the commands that are needed for experimenting with the Zoo Microworld System (ZMS) and the Rainy Day Scenario (RDS).

At present this is organized as lab5b in the software for the course TDDC65, "Artificial Intelligence and Lisp." This is likely to change in the near future so that this example becomes a free-standing application where several of the lab assignments for the course have been included and integrated. However the following refers to the presently available organization of the system.

4.1 Startup

The following commands are used for starting the ZMS/RDS in the Leonardo system.

```
loadk indivmap-kb
loadk indiv-kb          Only needed if lab results are to be
                        uploaded
loadk servdefs-kb
loadk lab5b-kb
startlab lab5b
```

The `startlab` command can be executed repeatedly; it starts a new episode each time it is called. The `steim` command (from earlier labs) shall not be used for this purpose.

4.2 Road Network

Entities for roadpoints are formed as in the following examples:

```
(crossing: 3 3)        The roadpoint (3, 3)
(pathpos: 3 3)         The roadpoint (3+, 3)
(trailpos: 3 3)        The roadpoint (3, 3+)
```

In all cases each of the arguments must be between 1 and 5.

4.3 Commands

Commands are specified with typical examples of arguments.

4.3.1 Configuration and Administration of Episodes

```
initpos 3 4
```

Sets the Warden's location to `(crossing: 3 4)`

```
neto (crossing: 3 4)
```

Calculates and displays the route that the Warden will use if given the `goto` command with the same argument, but does not actually move the Warden.

```
ephas
```

Shows the current state and the past history of the current episode.

```
put (crossing: 2 3) has-on-ground coin
```

This is an example of how the general-purpose `put` command can be used for assigning an attribute value to an entity, in this case, a roadpoint. The following attributes are used for roadpoints, with their possible values:

```
has-on-ground       coin, no
has-building        icecream-parlor, no
has-sprinkler       t, nil
```

Notice that a coin on the ground will be removed if it is picked up, but an icecream parlor at a location will not go away if the Warden buys an icecream there.

```
rembu
```

Removes buildings and sprinklers at all roadpoints.

4.3.2 Long Actions

The following is the main command for requesting a promenade by the Warden, with an example of the argument.

```
goto (crossing: 4 5)
```

This has the effect of calculating a sequence of steps for going from the Warden's present location to the indicated destination, while checking in each step for appropriate short actions that may be performed in-between according to the Warden's desire structure.

Notice that if one is interested in going repeatedly to the same location during a session, for example for having the Warden going back and forth between two roadpoints, then it is convenient to make assignments of roadpoints to session variables and use them like in the following example.

```
ssv .c (crossing: 4 5)
goto .c
```

4.3.3 Stepwise Operation of Short Actions

It is possible to request short actions directly from the command-line dialog, without having them embedded in a long action. The respective verbs are given directly as commands.

```
mp dir
```

Move the agent to the location indicated by the direction `dir` which may be either of the symbols `N, S, E, W, north, south, east, west.`

4.3.4 Prediction

The following commands are used for generating and viewing a successor situation or a sequence of successor situations at a given timepoint in an episode.

```
nxseq <.a1 .a2 ... .ak>
```

Creates the suite of successor situations

```
(succ .t .a1)
(succ (succ .t .a1) .a2)
   ...
```

where `.t` is the current time in the current episode, derives the changes of feature values from the current time to that situation, and displays the changed feature values in these. For example:

```
nxseq <[putdown-bag][takeout][pickup-bag]>
```

Finally,

```
val .s .f
```

Displays the value of the feature `.f` in the situation `.s` which may be either an integer (e.g. for the current timepoint) or a situation formed using the function `succ.`

4.4 Miscellaneous

When the entityfile `motivsys` is loaded it resets the global variable for the list of behavior rules to the empty list. This list obtains its members when entityfiles containing behavior rules are loaded, in particular the file `motivdefs` in the distribution of the RDS. Therefore, if `motivsys` is reloaded during a session then one must take care to also reload `motivdefs` and any other file containing behavior rules.