DESS: Demonstration
Expert System Shell

User Manual

Chapter 1

Introduction

DESS is an expert system shell written to demonstrate some of the basic concepts of expert system technology, including:

- forward chaining
- backward chaining
- tracing information
- explanations
- frames, with subclass hierarchies
- reasoning with uncertainty, using a certainty factor model.

Getting started

Ensure that the DESS program is executable, e.g.

```
% chmod 755 dess
```

To start DESS, type:

```
% dess
```

To leave DESS and return to UNIX, type:

```
|: exit;
```

(Note that all DESS commands are terminated with a semi-colon.)

 8 November 1997

Chapter 2

Reasoning with Facts

Facts

Facts (or ‘‘assertions’’) in DESS are enclosed in quotes.[1]

Facts indicating that the barometric pressure is rising and the western sky is cloudy can be introduced as follows:

```
|: 'the barometric pressure is rising';
|: 'the western sky is cloudy';
```

To find out what facts are known to the system, type:

```
|: show facts;
```

Rules

Rules in DESS have the form:

[**rule** <*RuleName*>] **if** <*Conditions*> **then** <*Consequences*>

For now, we will consider rules which contain facts. Facts on the left hand side of a rule can be separated by ‘and’ or ‘or’. When a rule is processed, ‘and’ takes precedence over ‘or’, unless disjunctions are enclosed in parentheses. The right hand side of a rule can be a single fact, or a conjunction of facts.

It is not *necessary* to give a rule name — the system will generate a unique rule name automatically if this is omitted from the rule definition. However, it is strongly recommended that the user should give each rule a name since this will make tracing information and explanations more meaningful.

Here are two simple rules about the weather:

```
|: rule r1
|: if      'the barometric pressure is rising' and
|:         'the western sky is cloudy'
|: then    'it is going to rain today';

|: rule r2
|: if      'it is going to rain today'
|: then    'I am not going out today';
```

[1] The ‘quote’ character, ‘double quote’ or ‘backquote’ can all be used, but matching quotes must be of the same type e.g. 'this is a fact', "It is John's birthday" and 'He said "Hello"' are all valid facts.

The `list rules` command displays the names of all rules known to the system.

```
|: list rules;
r1
r2
```

To inspect a particular rule, e.g. r1, type:

```
|: show rule r1;
```

To look at all of the rules, type:

```
|: show rules;
```

Loading from a File

While facts and rules can be typed in at the DESS prompt, this can become tedious. If you make a typing error when entering a rule, then you have to type the whole rule again. When you exit from DESS, all of your rules and facts will be lost so if you want to use them later you will have to type them all in again. Therefore, I recommend that you always define your rules in a file, and then load the file into DESS.

The `load` command is used to load facts and rules from a file. Suppose that the facts and rules described above are stored in a file called ‘weather’. They can be loaded by typing:

```
|: load "weather";
```

A ‘%’ (percent) sign in a knowledge base file indicates the start of a comment. Everything from a percent sign to the end of the line is ignored.

White space (<space>, <newline>, <tab>) can be used freely within a knowledge base file.

Working Memory

When a fact is read, it is added to working memory so that it can be used if forward chaining is invoked. To inspect the current contents of working memory, type:

```
|: wm;
2 elements in working memory.
      1 : 'the barometric pressure is rising'
      2 : 'the western sky is cloudy'
|:
```

DESS will first print the total number of items in working memory, and then list these, printing an identifying number beside each one (later we will see how these numbers provide the user with a convenient shorthand for referring to particular working memory elements).

Forward Chaining

Forward chaining is invoked using the command `fc`:

```
|: fc;
..
All applicable rules fired
|:
```

A dot is printed each time a rule fires. Two dots here mean that two rules were fired. Once all applicable rules have been fired, a message is displayed and the DESS prompt returns. Facts inferred by forward chaining are added to working memory. If we look at how the forward chaining sequence has affected working memory, we see that two new facts have been added:

```
|: wm;
4 elements in working memory.
      1 : 'the barometric pressure is rising'
      2 : 'the western sky is cloudy'
      3 : 'it is going to rain today'
      4 : 'I am not going out today'
|:
```

Typing `show facts` will also show that four facts are now known to the system.

"How" Explanations

The `how` command is used to inform the user how a particular working memory element has been derived. The keyword `how` is followed by the working memory element about which we want more information , e.g.

```
|: how 'I am not going out today';

By using rule r2 and knowing:
    'it is going to rain today'

|:
```

Alternatively, we can follow the keyword `how` with the number of one of the working memory elements:

```
|: how 3;

By using rule r1 and knowing:
    'the barometric pressure is rising' and
    'the western sky is cloudy'

|:
```

If we ask about a fact which was entered directly by the user, or read in from a file, the system replies: 'You told me'.

Clearing the Knowledge Base

The `clear` command removes all facts and rules, and clears working memory.

Tracing Forward Chaining Inference

The `trace` command lists the tracing options available, indicating whether each is currently on or off.

```
|: trace;

Trace options
-------------
      1 : Show the name of the chosen rule [off]
      2 : Show new working memory elements [off]

|:
```

The keyword `trace` can be followed by a list of integers, referring to option numbers. The state of each option selected will be switched and the list of options will be displayed again, showing new states, e.g. the following command will switch on tracing options 1 and 2 (the comma is optional):

```
|: trace 1, 2;

Trace options
-------------
      1 : Show the name of the chosen rule [on]
      2 : Show new working memory elements [on]

|:
```

(Repeating this command would switch them both off again.)

Now when we invoke forward chaining, the name of each selected rule is printed as that rule fires and each fact added to working memory is printed.

```
|: fc;
. Firing rule: r1
    Adding working memory element: 'it is going to rain today'
. Firing rule: r2
    Adding working memory element: 'I am not going out today'

All applicable rules fired
|:
```

The following sequence shows the knowledge base being cleared and initialised with facts and rules loaded from the file 'weather'. Since the trace option 'Show new working memory elements' is still on, the facts read from the file are printed to the screen. This option is switched off and forward chaining is invoked. This time, the name of rules being fired is the only tracing information displayed.

```
|: clear;
|: load "weather";
    Adding working memory element: 'the barometric pressure is rising'
    Adding working memory element: 'the western sky is cloudy'
|: trace 2;

Trace options
-------------
      1 : Show the name of the chosen rule [on]
      2 : Show new working memory elements [off]

|: fc;
. Firing rule: r1
. Firing rule: r2

All applicable rules fired
|:
```

Removing Items from Working Memory

The `remove` command is used to remove items from working memory. The keyword `remove` is followed by the working memory elements which we want to remove, or a list of integers corresponding to working memory elements, e.g.

```
|: wm;
4 elements in working memory.
      1 : 'the barometric pressure is rising'
      2 : 'the western sky is cloudy'
      3 : 'it is going to rain today'
      4 : 'I am not going out today'
|: remove 'I am not going out today';
|: remove 3;
|: wm;
2 elements in working memory.
      1 : 'the barometric pressure is rising'
      2 : 'the western sky is cloudy'
|: fc;
..
All applicable rules fired
|: wm;
4 elements in working memory.
      1 : 'the barometric pressure is rising'
      2 : 'the western sky is cloudy'
      5 : 'it is going to rain today'
      6 : 'I am not going out today'
```

Deleting Rules from the Knowledge Base

The `delete rule` command is used to delete rules from the knowledge base, e.g.

```
|: clear;
|: load "weather";
|: list rules;
rule_1
rule_2
|: delete rule rule_1;
|: list rules;
rule_2
|:
```

Backward Chaining

The `deduce` command[2] invokes backward chaining. DESS replies 'YES' if we try to deduce a fact which is known to the system or can be derived using the rules.

```
|: deduce 'the barometric pressure is rising';
YES
|: deduce 'I am not going out today';
YES
|:
```

If we try to deduce a fact which is not known to the system and cannot be derived using the rules, DESS replies 'NO', e.g.

```
|: deduce 'it is sunny';
NO
|:
```

New facts established during backward chaining are not added to working memory and are not stored explicitly. That is, typing '`wm`' or '`show facts`' will still show only two facts.

Tracing Backward Chaining Inference

If the trace option 'Show the name of the chosen rule' is on then the name of each rule tried is displayed. For each rule listed, there will be a matching line which stares whether that rule was successful or whether it failed.

[2] The command `bc` (''backward chain'') can be used as a synonym for `deduce` in DESS.

```
|: clear;
|: load "weather";
|: trace 1,2;

Trace options
-------------
      1 : Show the name of the chosen rule [on]
      2 : Show new working memory elements [on]

|: deduce 'I am not going out today';
 Trying rule: rule_2
 Trying rule: rule_1
 Rule successful: rule_1
 Rule successful: rule_2
YES
|:
```

Extended Example

A version of the creatures knowledge base that uses facts is given on another handout. This is also available on-line, in:

```
creatures.facts
```

Chapter 3

Object-Attribute-Value Triples

The Problem with Facts

Using simple facts in DESS we can only write very limited, specialised rules. For example, using facts we can define a rule which establishes John's nationality, if it is known that John was born in Sydney:

```
if      'John was born in Sydney'
then    'John is Australian';
```

The first problem is that this rule only enables us to establish John's nationality. What about other people born in Sydney? Next, what about people born in other Australian towns and cities? To define rules for each combination of person and town would be tedious, if not impossible.

To deal with more general rules, we need to use richer knowledge representations than simple facts. Specifically, we need to use variables and a means for specifying rules which model the relationships between variables.

Object-Attribute-Value Triples

Objects are often used in expert systems to represent real-world entities. Each object has a unique identifier. In DESS, this is just the object's name. Each object may have several *attributes*, representing that object's properties. Each attribute may have one or more *values*.

In DESS, object-attribute-value triples (OAVs) have the following syntax:

the <*Attribute*> **of** <*Object*> **is** <*Value*>

OAVs about where John and Mary were born, and the location of Sydney are introduced as follows:

```
|: the place_of_birth of john is sydney;
|: the place_of_birth of mary is sydney;
|: the location of sydney is australia;
```

(Note that there are no quotes.)

DESS stores OAVs in *frame* structures. You don't need to know about these for now — they will be described in detail later. However, you might find the command `show frames` useful for inspecting objects' properties.

```
|: show frames;
john
    place_of_birth: sydney

mary
    place_of_birth: sydney

sydney
    location: australia

|:
```

The three OAVs entered above are added to working memory:

```
|: wm;
3 elements in working memory.
      1 : the place_of_birth of john is sydney
      2 : the place_of_birth of mary is sydney
      3 : the location of sydney is australia
|:
```

Querying OAVs

We can include variables in OAVs in place of either the object identifier or attribute value (but not both). DESS variables begin with an upper-case letter or a question mark. OAVs typed in by the user which contain variables are *queries*, and all known values matching the variable are displayed, e.g.

```
|: the place_of_birth of Who is sydney;
john
mary
|: the place_of_birth of john is Where;
sydney
|:
```

If there is no known value for an object-attribute pair, DESS replies ‘UNKNOWN’ in response to a query.

```
|: the nationality of john is What;
UNKNOWN
|:
```

Rules containing OAVs

OAVs can appear on the left hand side and/or the right hand side of a rule. Here is a rule that infers someone is Australian if they were born in an Australian city:

```
|: rule nationality
|:         if      the place_of_birth of Person is City and
|:                 the location of City is australia
|:         then    the nationality of Person is australian;
```

Notice that both object and value in an OAV triple can be variables, and that the same variable can occur on both sides of a rule. This is an improvement, but is still restricted to reasoning about Australians. We can generalise the nationality rule to work for countries other than Australia:

```
|: rule nationality
|:         if      the place_of_birth of Person is City and
|:                 the location of City is Country and
|:                 the name_for_citizen of Country is Adjective
|:         then    the nationality of Person is Adjective;
```

To support this rule, the following OAV introduces an object called ‘australia’, and records the ‘name_for_citizen’ to be ‘australian’:

```
|: the name_for_citizen of australia is australian;
```

Backward Chaining

Backward chaining can be used to test whether a given value corresponds to a particular object-attribute pair. If the given OAV is stored or can be inferred from known information using the current rule base, DESS replies ‘YES’. Otherwise, DESS replies ‘NO’.

```
|: deduce the nationality of john is american;
NO
|: deduce the nationality of john is australian;
YES
```

If backward chaining is invoked with an OAV that has a variable in place of an explicit value, DESS will attempt to infer possible values for the variable.

```
|: deduce the nationality of john is What;
australian
```

User Input During Backward Chaining

When trying to satisfy a condition on the left hand side of a rule, DESS first tried to find whether a piece of information is known to the system or can be derived using rules and what is already known. Sometimes the system will not know all relevant information when backward chaining is initiated. In some of these cases it is reasonable for the user to be asked to provide some missing piece of information. This can only be done when the system has been told that a particular attribute is ‘askable’.

The syntax for notifying DESS that a particular attribute is ‘askable’ is:

askable <*Attribute*> <*Prompt*>

The following command will cause the system to prompt the user to type in a person’s place of birth when this attribute is needed:

```
|: askable place_of_birth 'where was this person born? ';
```

If we invoke backward chaining to infer Sue's nationality, the system will prompt for Sue's place of birth:

```
|: deduce the nationality of sue is What;
Frame sue: where was this person born? sydney
australian
|:
```

(Note that the user must **not** type a semi-colon after the reply.)

"Why" Explanations

During backward chaining, the user may type 'why' then asked for an attribute value. In response to this, DESS will explain why the user is being asked for this particular piece of information. This explanation consists of the instantiated rule whose antecedents the system is attempting to satisfy. After providing this explanation, the system repeats the prompt.

```
|: deduce the nationality of tom is What;
Frame tom: where was this person born? why

rule nationality
if
    the place_of_birth of tom is City and
    the location of City is Country and
    the name_for_citizen of Country is Adjective
then
    the nationality of tom is Adjective


Frame tom: where was this person born? sydney
australian
|:
```

DESS Remembers Inferred OAVs

Attribute values entered by the user or inferred by backward or forward chaining are stored in object frames so that these values can be used later without having to be derived a second time. If we look at the frame recording information about Tom, we see that both his place of birth and nationality have been stored.

```
|: show frame tom;
tom
    nationality: australian
    place_of_birth: sydney
|:
```

If we now invoke backward chaining to deduce Tom's nationality, DESS replies 'australian' by retrieving the stored value from the frame.

```
|:deduce the nationality of tom is What;
australian
|:
```

(If the trace option that shows the names of rules used was on, we would see that the rule called ‘nationality’ is only used the first time that we deduce Tom’s nationality).

Forward Chaining

Suppose working memory contains the following OAVs:

```
|: wm;
4 elements in working memory.
      1 : the place_of_birth of john is sydney
      2 : the place_of_birth of mary is sydney
      3 : the location of sydney is australia
      4 : the name_for_citizen of australia is australian
|:
```

The `fc` command invokes forward chaining. If the the nationality rule is in the rule base, this rule will fire twice — once for John and once for Mary:

```
|: fc;
..
All applicable rules fired
|: wm;
6 elements in working memory.
      1 : the place_of_birth of john is sydney
      2 : the place_of_birth of mary is sydney
      3 : the location of sydney is australia
      4 : the name_for_citizen of australia is australian
      5 : the nationality of john is australian
      6 : the nationality of mary is australian
|:
```

“How” Explanations

The `how` command is used to obtain an explanation of how an OAV in working memory has been derived.

```
|: how the nationality of john is australian;

By using rule nationality and knowing:
    the place_of_birth of john is sydney and
    the location of sydney is australia and
    the name_for_citizen of australia is australian

|:
```

As an alternative to typing the OAV in full, the keyword `how` can be followed by the number of a working memory element.

Arithmetic Expressions

In addition to scalar values and variables, arithmetic expressions can be given as attribute values within OAVs. This enables us to include calculations within rules. For example, the following rule will derive someone's weight in pounds if their weight in kilograms is known:

```
|: rule kg_to_lb
|:         if      the weight_in_kg of Person is X
|:         then    the weight_in_lb of Person is X * 2.20462;
```

If the system knows that John's weight in kilograms, then John's weight in pounds can be derived by either forward or backward inference.

As another example, suppose we have a rule for converting someone's height in feet to their height in inches:

```
|: rule feet_to_inches
|:         if      the height_in_feet of Person is X
|:         then    the height_in_inches of Person is 12 * X;
```

and we note that John is 6 feet tall:

```
|: the height_in_feet of john is 6;
```

We can then invoke backward chaining to test whether John is 72 inches tall:

```
|: deduce the height_in_inches of john is 72;
YES
|:
```

Expressions in DESS can contain integers, floating point numbers, and variables (whose values must be established before the expression is evaluated, e.g. the variable may appear in an earlier antecedent, as in the two rules shown above). DESS expressions can include the usual arithmetic operators: '+', '-', '*' and '/'. Sub-expressions can be nested inside round brackets.

Arithmetic Comparisons

The six comparison operators supported by DESS are: '=', '<>', '>', '<', '>=', '=<'. The expressions on either side of a comparison operator are as described in the previous section.

The following rule identifies whether someone is a teenager:

```
|: rule teenager
|:         if      the age of Person is Age and
|:                 Age >= 13 and
|:                 Age =< 19
|:         then    the age_group of Person is teenager;
```

Suppose John is 17 years old:

```
|: the age of john is 17;
```

DESS will be able to infer that John is a teenager using either forward or backward chaining. The following script shows the result of invoking forward chaining:

```
|: wm;
1 elements in working memory.
      1 : the age of john is 17
|: fc;
.
All applicable rules fired
|: wm;
2 elements in working memory.
      1 : the age of john is 17
      2 : the age_group of john is teenager
|: how 2;

By using rule teenager and knowing:
    the age of john is 17 and
    17 >= 13 and
    17 =< 19

|:
```

Multi-valued Attributes

All of the OAVs considered so far have single-valued attributes. In order to model many real-world situations we must be able to represent attributes with more than one value — a book may have more than one author; a patient may have more than one symptom; a mechanical assembly can have several sub-components.

We use the term *cardinality* to describe the number of values an attribute may have.

Some systems allow the user to specify the maximum and/or minimum number of values an attribute may have, and the system will report an error of this number is violated. DESS supports single-valued attributes and multi-valued attributes (which may have zero or more values), but does not enforce the cardinality of the latter.

In DESS, OAVs with multi-valued attributes have the keyword ‘include’ instead of ‘is’:

```
|: the hobbies of liz include swimming;
|: the hobbies of liz include skating;
```

The following rule states that skaters will own skates:

```
|: rule owns_skates
|:         if      the hobbies of Person include skating
|:         then    the possessions of Person include skates;
```

Using this rule, DESS will infer that Liz owns skates:

```
|: deduce the possessions of liz include What;
skates
|: show frame liz;
liz
    possessions: skates
    hobbies: skating
    hobbies: swimming
|:
```

Chapter 4

Frames

Introduction to Frames

In some expert systems, real world objects are represented by *frames*. The properties of an object are represented by *slots*. The properties of a slot are represented by slot *facets*.

Specific, or *instance* frames represent actual real world entities. *Generic*, or *prototype* frames represent knowledge about a general concept. That is, they represent classes to which individual instances may belong.

Frames can be related to one-another, reflecting relationships between classes and entities in the real world. The most important relationships that need to be modelled are ‘‘sub-class-of’’ and ‘‘instance-of’’. Both of these relationships allow properties to be inherited.

Relationship to OAVs

In Chapter 3, we used object-attribute-value triples to represent data. In DESS, OAVs are stored in frame structures. When we type in an OAV,

```
|: the place_of_birth of john is sydney;
```

the value ‘sydney’ is inserted into the ‘value’ facet of the ‘place_of_birth’ slot of the frame called ‘john’. If no frame called ‘john’ exists, one is created.

When an OAV containing a variable is typed, the query is answered by retrieving all possible values for the variable from frame structures.

Creating New Frames Explicitly

We can introduce new frames and initialise their slots explicitly. When introducing a new frame in this way, we must declare it to be a generic frame by stating that it is a subclass of some other generic frame, or an instance frame by stating that it is an instance of some generic frame. (It is not necessary for the related class frame to exist! It is sufficient to invent the name another class simply as a ‘‘place-holder’’.) Thus, a new frame representing the class of employees can be introduced as follows:

```
|: employee subclass of person;
```

This command will create a new generic frame called ‘employee’. There does not have to be a generic frame called ‘person’.

We can create a new instance frame as follows:

```
|: tom instance of employee;
```

When introducing a new class, we can give default values for particular slots. For example, the following declaration crates a generic frame called ‘accountant’ and specifies a default salary of 10000 for all accountants:

```
|: accountant subclass of employee with
|:   salary: 10000;
```

We can also give initial values for properties in an instance frame. Several values can be given for a multi-valued property by listing these inside square brackets (separated by commas).

```
|: liz instance of employee with
|:   salary: 16000
|:   hobbies: [cycling,squash,tennis];
```

Slot Cardinality

By default, all slots in DESS are single-valued. Thus, only one value can be stored in their value facet. However, DESS also supports multi-valued slots. There are two ways to make a slot multi-valued.

The first is to give several values for that slot when the frame is introduced (as with Liz's hobbies in the example above).

The second is to put 'multi' into a slot's cardinality facet. In DESS, any number of facet and facet value pairs can be given in a list following the slot name. For example, we can create a frame representing Sue, inserting 'reading' into the value facet of the slot called 'hobbies' and 'multi' into the cardinality facet of that slot.

```
|: sue instance of person with
|:   hobbies: [value: reading, cardinality: multi];
```

Since the 'hobbies' slot has been declared to be multi-valued, if we state that Sue also enjoys archery at some later stage, then archery will be added to the values for this slot, rather than replacing reading as Sue's hobby:

```
|: the hobbies of sue include archery;
|: the hobbies of sue include What;
archery
reading
|:
```

Listing Frames

The command `list frames` prints the names of all frames known to the system. These are indented with respect to the classes to which frames belong, thus reflecting the class hierarchy. For example, suppose we load a frame version of the creatures knowledge base (given in a separate handout). The command `list frames` will give the following output:

```
|: list frames;
creature
    mammal
        carnivore
            cheetah
            tiger
        ungulate
            giraffe
            zebra
    bird
        ostrich
        penguin
        albatross
|:
```

Inspecting Frames

The command `show frame` is used to display information about a particular frame, e.g.

```
|: show frame mammal;
mammal subclass_of creature
    body_covering: hair
    reproduction: live_birth
    feeds_young_on: milk
    motion: walks
|:
```

The name of the class of which the given frame is an instance or a subclass is printed, together with the kind of relationship that holds between the two frames. The values stored in value facets of the local frame are also displayed.

The contents of all frames can be displayed using the command `show frames`.

Deleting Frames

The command `delete frame` can be used to delete a particular frame.

Any other frames which depend on that frame are also deleted. That is, if a generic frame is deleted, all subclasses and instances of that frame will also be deleted, e.g.

```
|: delete frame penguin;
|: delete frame carnivore;
|: list frames;
creature
    mammal
        ungulate
            giraffe
            zebra
    bird
        ostrich
        albatross
|:
```

Rules Involving Class Membership

Statements about class membership with the syntax:

<*Class Variable*> **is an instance of** <*Class Name*>

can appear in DESS rules. Thus, rules can be used to infer class membership.

```
rule mammal
        if      the body_covering of Creature is hair or
                the feeds_young_on of Creature is milk
        then    Creature is an instance of mammal;
```

Class membership statements can also appear among a rule's antecedents, e.g.

```
rule ungulate_1
        if      Creature is an instance of mammal and
                the food of Creature is grass
        then    the feeding_type of Creature is ungulate;
```

Inheritance

Generic frames are also known as prototype frames, because they can be thought of as providing a prototype for information about object instances. When no value is stored for a particular attribute in a particular frame, it is possible for a default value to be *inherited* from a another frame.

Considering the frames describing creatures, no information about an ungulate's body covering is stored in the 'ungulate' frame. However, an 'ungulate' is a subclass of 'mammal' and thus can inherit the default body covering stored in the 'mammal' frame.

```
|: the body_covering of ungulate is What;
hair
|:
```

In this situation, we refer to 'mammal' as the *superclass* of 'ungulate'. Moreover, it is a direct superclass — mammal is an indirect superclass of 'giraffe'.

Suppose we add two instances: Stretch (a giraffe) and Swifty (a cheetah):

```
|: stretch instance of giraffe;
|: swifty instance of cheetah;
|: list frames;
creature
    mammal
        carnivore
            cheetah
                swifty
            tiger
        ungulate
            giraffe
                stretch
            zebra
    bird
        ostrich
        penguin
        albatross
|:
```

We can now ask questions about these particular creatures, e.g.

```
|: the legs_and_neck of stretch is What;
long
|: the legs_and_neck of swifty is What;
short
|:
```

We have not made any statements about the legs and neck of either animal. However, Stretch inherits the value ‘long’ for this attribute because it is an instance of giraffe. Swifty inherits the value ‘short’ from the frame ‘creature’, which is a distant superclass.

An attribute value stored with a particular instance can override a default value stored in a generic frame. For example, suppose Snowy is an white tiger:

```
|: snowy instance of tiger with
|:   colour: white;
```

When we enquire about Snowy’s colour, this overriding value is returned:

```
|: the colour of snowy is What;
white
|:
```

All inheritance in DESS is *overriding* (or ‘supersedes’ inheritance). That is, a value for a particular attribute in a more specialised frame will be used instead of any in more general frames. Some systems also support ‘merge’ inheritance, where the values in a more specialised frame are used in addition to those in more general frames.

Some systems allow instance frames to be directly related to more than one generic frame, and allow each generic frame to be a direct subclass of more than one generic

frame. This enables a frame to inherit properties from several others. This situation is known as *multiple inheritance* and care must be taken when a frame inherits a particular property from more than one ‘‘parent’’ frame — the system must resolve the conflict by selecting which inheritance route will be used. DESS ‘‘avoids’’ this problem by only allowing single-inheritance.

Demons

Demons are pieces of code that can be stored in a frame slot. This is sometimes called *procedural attachment*. These pieces of code are usually executed either when a slot value is accessed or modified.

A demon can be used to derive a slot value on demand. In DESS, this is achieved by giving a rule as the value of a slot’s ‘‘when_accessed’’ facet. For example, rather than storing both a person’s year of birth and age in instance frames, it is sufficient to store just the year of birth, and then calculate their age when it is needed.

```
person subclass of living_thing with
    age: [when_accessed:
            if    the year_of_birth of Person is Year
            then  the age of Person is 1997 - Year];
```

Clearly, a more sophisticated rule could take into account a person’s date of birth, and could retrieve the present year from the system’s clock instead of assuming that it is still 1997 — however, this simplistic example illustrates the idea.

It is expected that a ‘‘when-accessed’’ demon will have as its consequent an OAV:

the <*Attribute*> **of** <*Object*> **is** <*Value*>

where <*Attribute*> is the name of the slot whose value the demon will calculate, <*Object*> is a variable representing the object instance, and <*Value*> is an expression that will provide the slot value. DESS invokes backward chaining to establish the consequent of a ‘‘when-accessed’’ rule.

Suppose we add a frame representing a John, who was born in 1980:

```
john instance of person with
  year_of_birth : 1980;
```

If we now ask the system for John’s age, the ‘‘when-accessed’’ demon from the ‘‘person’’ frame’s ‘‘age’’ slot is inherited since John is an instance of person, and the system calculates that John is seventeen years old:

```
|: the age of john is What;
17
|:
```

An additional action performed by DESS is to store the newly calculated value in the frame called ‘‘john’’, so that if John’s age is needed again, the ‘‘cached’’ value can be retrieved, and it is not necessary to calculate that value again. This is not so important for this example, but if the when-accessed demon is computationally expensive, the performance of the system will be improved if derived values are cached.

Since backward chaining is invoked to establish a value for a slot, it is possible that the user may be asked for additional information if an antecedent of the ‘‘when-accessed’’ rule uses an attribute value that is unknown, but askable. For example, suppose we ask about Tom’s age, when nothing is known about Tom, other than that he is a person, and suppose that someone’s year of birth is askable:

```
|: tom instance of person;
|: the age of tom is What;
Frame tom: when was this person born? 1976
21
|:
```

We could, of course, ask ‘‘why’’ the question prompting for Tom’s year of birth is being asked, and DESS would print the text of the ‘‘when-accessed’’ rule.

Algorithm used by DESS for Retrieving Attribute Values

When answering an OAV query, DESS uses the following algorithm when trying find the value for a given attribute (when value(s) are found, they are returned):

(a) look for stored values in the local frame

(b) look for a ‘when-accessed’ method in the local frame, and use it

(c) if the local frame cannot provide a value, and the local frame is an instance or sub-class of another, let that ‘super’ frame be the new local frame, and repeat from (a)

(d) otherwise, no value can be retrieved

Even if no value can be found by inspecting the frames, it may be possible to establish a value for a given property using inference.

Copying Data from Frames to Working Memory

When a frame is introduced explicitly, the data that it contains is not added to working memory automatically. Thus, if we want to use forward chaining to reason with this data, we must first copy it into working memory. The command `add frame` copies into working memory the contents of a given frame, together with any default attribute values that are inherited from other frames, e.g.

```
|: wm;
0 elements in working memory.
|: show frame stretch;
stretch instance_of giraffe
|: add frame stretch;
|: wm;
12 elements in working memory.
      1 : stretch is an instance of giraffe
      2 : the body_covering of stretch is hair
      3 : the colour of stretch is tawny
      4 : the eyes of stretch is point_sideways
      5 : the feeds_young_on of stretch is milk
      6 : the feet of stretch is hoofs
      7 : the food of stretch is grass
      8 : the legs_and_neck of stretch is long
      9 : the marking of stretch is dark_spots
     10 : the motion of stretch is walks
     11 : the reproduction of stretch is live_birth
     12 : the teeth of stretch is blunt
|:
```

Chapter 5

Uncertainty

Certainty Factors

DESS uses a MYCIN-like certainty factor model for representing and reasoning with uncertain knowledge and data. In DESS, each rule, fact and object-attribute-value triple has a certainty factor. Certainty factors are values in the range -1.0 to +1.0. A value of +1.0 means that we are sure of something; a value of -1.0 means that something is definitely untrue. A certainty factor of zero means that we know nothing about whether a piece of knowledge is true or not.

In DESS, if no certainty factor is given explicitly, the system assumes that a piece of knowledge or data is true, and uses a certainty factor of 1.0 implicitly. Thus, in earlier sections, we did not need to declare certainty factors when adding to the knowledge base.

Certainty Factors of Facts

Suppose facts ‘A’, ‘C’, ‘D’ and ‘E’ are known with certainty factors of 0.8, 0.9, 0.7 and 0.5, respectively. These facts and their certainty factors are made known to DESS as follows:

```
|: 'A' with cf 0.8;
|: 'C' with cf 0.9;
|: 'D' with cf 0.7;
|: 'E' with cf 0.5;
```

Since all of the certainty factors are greater that zero, this means that we believe that each is true to a greater or lesser extent — there is most confidence in ‘C’ being true, and ‘E’ is the fact about which we are least certain.

Certainty Factors of Rules

A certainty factor is associated with each rule consequent. This represents our confidence in a consequent being true if all of the antecedents are true.

First, consider rule b1 which states that ‘B’ is definitely true if ‘A’ is true. In DESS, we would express this rule as

```
rule b1
    if    'A'
    then  'B' with cf 1.0;
```

or simply:

```
rule b1
    if    'A'
    then  'B';
```

since DESS assumes 1.0 as a default value for a rule's certainty factor.

Of course, we may not know with complete certainty that the antecedents are true. In these cases, we need to combine the certainty factors of the antecedents and of the rule to establish a certainty factor for the consequent. This is done in two steps. The first is ''antecedent combination'' which means finding the antecedent certainty factor for the rule. This is done simply by taking the minimum of the certainty factors of the antecedents to be the certainty factor for the left hand side of the rule. The second step involves multiplying the antecedent certainty factor by the certainty factor that the consequent is true given the antecedents.

As an example, rule b1 has only one antecedent, so its certainty factor (0.8) will be the antecedent certainty factor for rule b1. The rule certainty factor for rule b1 is 1.0. Multiplying 0.8 by 1.0 gives 0.8 as the certainty factor for fact 'B' using rule b1.

Suppose we have a second rule for establishing 'B':

```
rule b2
    if    'C' and 'D' and 'E'
    then  'B' with cf 0.6;
```

The antecedent certainty factor for rule b2 is the minimum of the certainty factors of the individual antecedents, i.e. the minimum of 0.8, 0.9, 0.7 and 0.5, which is 0.5. Multiplying this by the rule certainty factor for rule b2 (0.6), we get 0.3 as the certainty factor for fact 'B' using rule b2.

Multiple Consequent Combination

In the previous section, we saw that 'B' could be inferred with a certainty factor of 0.8 using rule b1 and a certainty factor of 0.3 using rule b2. Both of these rules on their own support the conclusion that 'B' is true. Intuitively, considering these two pieces of evidence together increases our confidence that 'B' is true. We now show how numerical certainty factors can be combined to reflect this.

Suppose some fact (or attribute value) is already known with a certainty factor of CFp, and that using some other rule now brings evidence that the same item is true with a certainty factor of CFn. The way that these two certainty factors are combined depends on their signs. We consider three cases.

First, if both certainty factors are positive then each supports the conclusion, and combining will result in a new certainty factor CF that is greater than both CFp and CFn. CF is calculated as follows:

CF = CFp + CFn * (1 - CFp)

i.e.

CF = CFp + CFn - CFp * CFn

The second case is when both certainty factors are negative. Both suggest that the conclusion is false, and taken together reduce still further out belief that the conclusion might be true. In this case, the new certainty factor CF will have a lower value than both CFp and CFn, and is calculated using the following formula:

CF = CFp + CFn * (1 + CFp)

i.e.

CF = CFp + CFn + CFp * CFn

Finally, if the two certainty factors have opposite signs, then one piece of evidence supports the conclusion, while the other argues against the conclusion. In this case, the new certainty factor CF is calculated using the following formula:

CF = (CFp + CFn) / (1 - min(|CFp|, |CFn|)

The new certainty factor will lie between CFp and CFn.

Certainty Factors of OAVs

Each object-attribute-value has an associated certainty factor. If no certainty factor is given explicitly, a default value of 1.0 is used.

Here are some OAVs based on [Jackson, page 47].

```
the stain of organism_1 is gramneg;
the morphology of organism_1 is rod with cf 0.8;
the morphology of organism_1 is coccus with cf 0.2;
the aerobicity of organism_1 is aerobic with cf 0.6;
the aerobicity of organism_1 is faculative with cf 0.4;
```

The attributes are single-valued, so the morphology cannot be both rod and coccus. However, DESS allows alternative values to be recorded when there is uncertainty about the true value, and the certainty factors of each alternative value is stored.

```
rule enterobacteriaceae
    if    the stain of X is gramneg and
          the morphology of X is rod and
          the aerobicity of X is aerobic
    then  the class of X is enterobacteriaceae with cf 0.8;
```

The antecedent certainty factor is 0.6 (the minimum of 1.0, 0.8 and 0.6). This is multiplied by the rule certainty factor to give 0.48 as the certainty factor that the class of organism_1 is enterobacteriaceae.

```
|: deduce the class of organism_1 is What;
enterobacteriaceae with cf 0.48
|:
```

The same conclusion would be reached with the same certainty factor by forward chaining:

```
|: fc;
.
All applicable rules fired
|: wm;
6 elements in working memory.
      1 : the stain of organism_1 is gramneg
      2 : the morphology of organism_1 is rod with cf 0.8
      3 : the morphology of organism_1 is coccus with cf 0.2
      4 : the aerobicity of organism_1 is aerobic with cf 0.6
      5 : the aerobicity of organism_1 is faculative with cf 0.4
      6 : the class of organism_1 is enterobacteriaceae with cf 0.48
|:
```

Chapter 6

Reaction Systems

Introduction to Reaction Systems

In the previous chapters, we have been concerned with *deduction systems* where the outcome of inference is that some new assertions become known by virtue of rule antecedents being known to be true. Deduction systems deal with static worlds where nothing established to be true can ever become false.

In contrast, *reaction systems* can model dynamic situations, and thus can be used for constructive problem solving. As an example of a dynamic situation, suppose we want to model the task of packing machine parts into boxes for shipping. In performing this task, part number 4321 may initially be on a shelf in the warehouse. When it is placed into a packing box, its location changes — it is now in a box and no longer on the warehouse shelf. Thus, an initial assertion that part 4321 is on the warehouse shelf, which was true at one time, is now false.

A simple reaction system that bags groceries (Adapted from P.H. Winston, ‘‘Artificial Intelligence’’, Third Edition, Addison-Wesley, 1992) is given in a lecture handout.

“If-delete-add” Rule Syntax

In reaction systems, if the conditions on the left hand side of a rule are true then actions on the right hand side of that rule can be performed. These actions may include adding a new assertion or deleting an existing assertion. This extra freedom is reflected in DESS syntax through ‘‘if-delete-add’’ rules. The following rule (taken from the ‘‘BAGGER’’ program) illustrates this syntax:

```
rule B4
    if      'step is bag-large-items' and
            the items_to_be_bagged of order include Item and
            the size of Item is large and
            the current_bag of order is Bag and
            the number_of_large_items of Bag is L and
            L < 6
    delete  the items_to_be_bagged of order include Item and
            the number_of_large_items of Bag is L
    add     the contents of Bag include Item and
            the number_of_large_items of Bag is L + 1;
```

Note that several items may be deleted from working memory and several added by a single rule.

Either the ‘‘delete’’ part or the ‘‘add’’ part may be omitted.

In addition to the contents of working memory changing, frame structures are modified when reaction rules are fired.

Conflict Resolution

Conflict resolution strategies are needed in reaction systems because it is generally important that a particular rule in the conflict set should be selected for firing in preference to others. Since firing that rule may delete elements from working memory, other rules in the conflict set for that cycle may now be ineligible for firing. Thus the conflict resolution can select which rules will fire and which will not, rather than just selecting the order in which rules will fire (as in deduction systems).

The conflict resolution strategies supported by DESS are *rule ordering* and *refractoriness*. Thus, the first rules loaded into the rule base are given priority over those loaded later (rules will be loaded in the order in which they appear in the source file).

Contexts

Some synthetic tasks can be divided into subtasks which must be completed in sequence. During each of these steps, a subset of the rules will be applicable. A reaction system can use *contexts* uses to ensure that only rules relevant current subtask are considered. This means associating *context symbols* with rules, and only considering rules for the current subtask when building the conflict set.

Contexts can be implemented simply in DESS using facts representing context symbols in the antecedents of a rule. It is best to make the context symbol the first antecedent, since the current context can be checked quickly and easily.

Some rules serve only to manipulate contexts. The following rule, taken from the BAGGER program, changes the current context from the ‘check order’ subtask to the ‘bag large items’ subtask:

```
rule B2
    if      'step is check-order'
    delete  'step is check-order'
    add     'step is bag-large-items';
```

Since the context symbol is this rule’s only antecedent, this rule will always be in the conflict set while the system is performing the ‘check order’ subtask. Therefore, it should be the last rule for that context, so that all other applicable rules for the context are tried first before the system progresses to the next subtask.

User Input During Forward Chaining

In Chapter 3, we saw how some attributes could be declared to be ‘askable’, thus enabling the system to prompt the user to supply a value during backward chaining. However, reaction systems work by forward chaining. Therefore we need some other way of prompting for input. This can be done by introducing a prompt within a rule using the following syntax:

ask <*prompt*>

This pattern can occur in a DESS rule wherever a value can appear. For example, the following rule, taken from the BAGGER program, has a question on the left hand side of a comparison.

```
rule B1
    if      'step is check-order' and
            the items_to_be_bagged of order include potato_chips and
            not(the items_to_be_bagged of order include pepsi) and
            ask 'Do you want Pepsi added to the order (y/n)? ' = y
    then    the items_to_be_bagged of order include pepsi;
```

If the first three antecedents are satisfied, the value typed by the user in response to the question is compared with 'y'. If the user types 'y', the rule fires and the consequent is added as a new assertion.

DESS cannot (at present) respond to ''why'' requests typed by the user in response to these prompts.

Tracing Reaction Systems

If trace option 2 is on, DESS will report all elements removed from working memory as well as those added to working memory.

Appendix A

Reasoning with Facts — Creatures Knowledge Base

(Adapted from P.H. Winston, ‘‘Artificial Intelligence’’, Third Edition, Addison-Wesley, 1992)

Assume that the file ‘‘creatures_A’’ contains the following rules:

```
rule mammal_1
     if   'body covering is hair'
     then 'creature is a mammal';

rule mammal_2
     if   'feeds young on milk'
     then 'creature is a mammal';

rule bird_1
     if   'body covering is feathers'
     then 'creature is a  bird';

rule bird_2
     if   'motion is flies' and
          'reproduction is eggs'
     then 'creature is a bird';

rule feeding_type_1
     if   'creature is a mammal' and
          'food is meat'
     then 'feeding type is carnivore';

rule feeding_type_2
     if   'creature is a mammal' and
          'teeth are pointed' and
          'feet are claws' and
          'eyes point forward'
     then 'feeding type is carnivore';

rule feeding_type_3
     if   'creature is a mammal' and
          'food is grass'
     then 'feeding type is ungulate';
```

```
rule feeding_type_4
     if   'creature is a mammal' and
          'feet are hoofs'
     then 'feeding type is ungulate';

rule species_1
     if   'feeding type is carnivore' and
          'colour is tawny' and
          'marking is dark spots'
     then 'species is cheetah';

rule species_2
     if   'feeding type is carnivore' and
          'colour is tawny' and
          'marking is black stripes'
     then 'species is tiger';

rule species_3
     if   'feeding type is ungulate' and
          'colour is tawny' and
          'marking is dark spots' and
          'legs and neck are long'
     then 'species is giraffe';

rule species_4
     if   'feeding type is ungulate' and
          'colour is black and white'
     then 'species is zebra';

rule species_5
     if   'creature is a bird' and
          'motion is walks' and
          'colour is black and white' and
          'legs and neck are long'
     then 'species is ostrich';

rule species_6
     if   'creature is a bird' and
          'motion is swims' and
          'colour is black and white'
     then 'species is penguin';

rule species_7
     if   'creature is a bird' and
          'motion is flies'
     then 'species is albatross';
```

To load these rules, we type:

```
|: load "creatures_A";
```

Suppose we now add the following facts:

```
|: 'body covering is hair';
|: 'food is grass';
|: 'colour is black and white';
```

We can use either backward chaining of forward chaining to infer the species.

Backward Chaining

```
|: deduce 'species is tiger';
NO
|: deduce 'species is zebra';
YES
|:
```

Forward Chaining

```
|: wm;
3 elements in working memory.
      1 : 'body covering is hair'
      2 : 'food is grass'
      3 : 'colour is black and white'
|:fc;
...
All applicable rules fired
|: wm;
6 elements in working memory.
      1 : 'body covering is hair'
      2 : 'food is grass'
      3 : 'colour is black and white'
      4 : 'creature is a mammal'
      5 : 'feeding type is ungulate'
      6 : 'species is zebra'
|:
```