INCO - COPERNICUS

A Distributed Expert System
for the Management of a National Network

Workpackage 2: Implementation of the Prototypes.
Task 10: ***Implementation of the Expert System Shell Prototype.***

Thessaloniki - 11/97

1. INTRODUCTION 1

2. THE DEVICE ACTIVE OOKB SYSTEM 2

2.1 THE DEVICE RULE INTEGRATION SCHEME 3
2.2 THE CONDITION COMPILATION METHOD 3
2.3 THE COMPLEX EVENT NETWORK 3
2.4 PRODUCTION RULE FIRING 4
2.5 EXTENSIBILITY OF DEVICE 5

3. USER'S GUIDE 6

3.1 DEFINING THE SCHEMA 6
3.2 DEFINING PRODUCTION RULES 8
3.2.1 Rule Condition 8
3.2.2 Rule Action 9
3.3 INITIALISING OBJECT INSTANCES 10
3.4 DEVELOPMENT AND DEBUGGING FACILITIES 10

4. EVALUATION: THE CHEMICAL SPILL 16

4.1 PROBLEM STATEMENT. 16
4.2 SCHEMA 16
4.3 KNOWLEDGE BASE 17
4.4 TEST SESSION OF THE CHEMICAL SPILL 20

5. FUTURE WORK 22

5.1 CONCERNING THE FACILITIES OF THE EXPERT SYSTEM SHELL 22
5.2 CONCERNING THE INTERFACES OF THE EXPERT SYSTEM SHELL 23

APPENDIX A - PRODUCTION RULE SYNTAX 25

APPENDIX B - THE RULES (KB) OF THE CHEMICAL SPILL EXPERT SYSTEM. 26

APPENDIX C - INITIALISATION FILE OF THE CHEMICAL SPILL EXPERT SYSTEM. 32

1. Introduction

This booklet describes the prototype expert system shell DEVICE that will be used in the ExperNet project for building the distributed expert system for managing a TCP/IP based WAN. DEVICE is an active Object-Oriented Knowledge Base System that is built on top of an active Object-Oriented Database. DEVICE supports multiple rule types, like event-driven rules, data-driven (or production) rules, and deductive rules. Furthermore, it gives the user the ability to define new rule types by extending the basic functionality of production rules.

DEVICE was originally built in ECLiPSe Prolog on top of the EXACT active OODB. EXACT was an extension of the ADAM OODB, which was built in Prolog. For the sake of the ExperNet project, DEVICE was ported to SICStus Prolog, in order to use (as a backup solution) the network communication primitives that the UPM partner has implemented for SICStus Prolog. Luckily, the network version of CSP-II is ready, so the next step will be to port DEVICE in CSP-II. This porting will not be very difficult, since both versions of Prolog follow (more or less) the ISO standard.

DEVICE was extensively tested and many of its bugs were corrected during the evaluation period of the shell. Numerous single-rule examples, a tiny expert system with 5 rules, and a small expert system with 30 rules, were used as a testbed. DEVICE language proved to be quite flexible and expressible to encode all such examples, which were taken from numerous research papers of the literature, plus examples from textbooks on expert systems. The two small expert systems were originally written in the OPS5 and Mike expert system shells, respectively.

The booklet contains several examples that could help a knowledge base developer to use the rule definition language of DEVICE, along with the class and object definition facilities of ADAM for building an expert system. Furthermore, we describe various debugging facilities for using the DEVICE system. Finally, we describe various future extensions both for the extending the shell facilities and for interfacing DEVICE to the rest of the ExperNet components.

The rest of this booklet is organised as follows: Section 2 focuses on the expert system shell model that will be the vehicle for the implementation of the various expert modules of ExperNet. Here we describe in detail the DEVICE Active OOKB System that will be the core of the Expert System Shell. In Section 3 guidelines are given on how to use the DEVICE Expert System shell and namely, how to define classes, object instances and rules. In addition a list of development and debugging facilities, that have been implemented, are described. In Section 4 we describe in detail a small expert system that we have developed by using this prototype, in order to evaluate it. Finally, in Section 5, we give guidelines on additional features for DEVICE, that will be required in the frame of ExperNet.

2. The DEVICE Active OOKB System

DEVICE (**D**ata-driven & **EV**ent-driven rule **I**ntegration using **C**omplex **E**vents) is a sequential KBS that runs on top of ADAM, an OODB built in Prolog, and EXACT, an extension of ADAM with events and ECA rules (Figure 1). DEVICE is an active OODB enhanced with high-level rule facilities. It provides the infrastructure for the smooth integration of production and deductive rules into an active OODB that generically supports event-driven rules only. The integration is based on the compilation of the condition of both high-level rule types into a discrimination network that consists of simple and complex events which record and combine database modifications that could possibly make a rule fire.

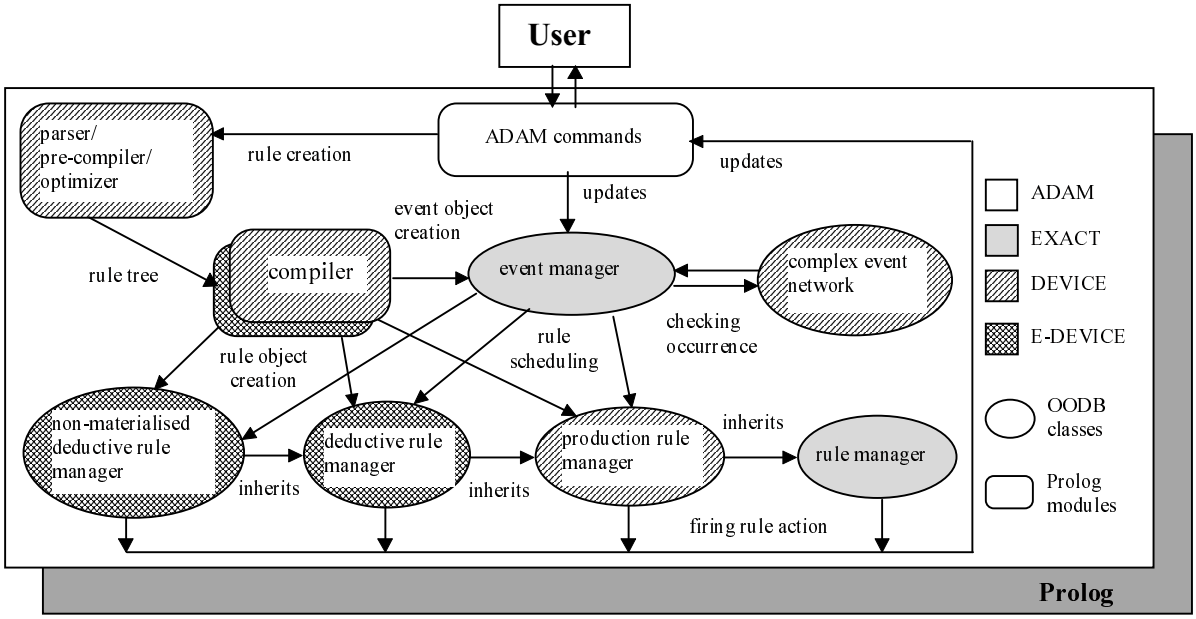


Figure 1. The architecture of the DEVICE system.

The resulting system is a flexible, yet efficient, KBS that gives the user the ability to express knowledge in a variety of high-level forms for advanced problem solving in data intensive applications. The extensibility of the system is due to the use of an object-oriented (OO) infrastructure which allows:

- partial redefinition of rule managers, and
- transparent extensions and improvements of the rule matching and execution phases.

Classes in DEVICE are related through *is-a* links that define an inheritance hierarchy, while instance objects are related through the *is-instance-of* relationship with their classes. Classes inherit slot and method definitions from their super-classes, whereas instance objects inherit slot and method names from their classes. Attribute (synonym for slot) values are defined at object creation value (if the slot is mandatory) or later through an object-modification method. Default values usually do not exist for OODBs, unless explicitly programmed through the use of metaclasses.

Attributes in OODBs have certain sub-attributes which are actually part of the original slot definition and define the attribute data type, its multiplicity, its optionality, etc. Inheritance of slots is usually mandatory, i.e. each slot is inherited to all of its subclasses. Values however are not inherited.

Constraints on attributes are usually expressed using two ways: Methods and Rules. The former incorporates parameter and value checking inside the methods that modify the object attributes and either prevent or correct those modifications. On the other hand, there could also exist certain systematic method wrapping constraint checking algorithms.

Active rules have also been used to enforce semantic integrity constraints by either being activated before the modifications and failing the transactions upon constraint violation, or by providing constraint repairing actions that try to recover the database from the incorrect state that has been taken into after the illegal modification.

Demons can be very easily emulated using events and ECA rules. More specifically, events can be defined and signalled for each of the recognisable method for any attribute of any class. Events can be signalled both before and after the execution of the related method and their raising causes one or more associated ECA rules to be triggered. Then the condition of those rules is checked and if it is satisfied then their actions can be executed. This active rule scheme can be used for: a) providing derived attributes, b) constraint maintenance, c) application programming, d) security/auditing, etc.

Another sort of demon is the method itself, which can perform many things before and/or after the actual data access or modification is performed. However, the use of ECA rules provides more flexibility, re-usability, extensibility and transparency to the application programmer.

Finally, concerning the rule base of the expert system, this can be a mixture of ECA, production and deductive rules. The two latter are high-level rules, whose integration into the active OODB has been smoothly achieved in DEVICE. Backward chaining or goal-driven rules already exist in DEVICE in the form of methods. Methods are pieces of Prolog code, therefore a backward chaining declarative language is provided.

2.1 The DEVICE Rule Integration Scheme

The main idea behind rule compilation is that high-level rules are in the form: `IF condition THEN` *`consequent`*, while ECA rules are in the form: `ON event IF condition THEN action`. Production and deductive rules are compiled to ECA rules, in order to be constantly monitored by the active database. The condition of a rule is compiled into a complex event network, which is associated with the event-part of the ECA rule, while the action-part of the ECA rule depends on the type of high-level rule. More specifically, procedural production rule actions are directly used in the ECA rule.

2.2 The Condition Compilation Method

Our condition compilation method uses complex events to translate a high-level rule into only one ECA rule. For example consider the following (abstract) rule:

P1: IF a & b & c THEN <*consequent*>

which is translated into the following ECA rule:

CA1: ON e_a & e_b & e_c [IF true] THEN <*consequent*>

where primitive events e_a, e_b, e_c detect the insertion of the data items `a`, `b`, `c`, respectively, and the operator `&` denotes the conjunction of the events. The complex event manager of the OODB monitors the above events and incrementally combines their parameters in order to detect the occurrence of the complex event. When the latter is detected, the condition of the rule has been matched and the rule manager is responsible to schedule it for firing.

Notice that the condition part of the ECA rule is always `true`, because all condition tests have been incorporated into the complex event. However, some small static conditions are allowed in the condition part of the ECA rule through the `prolog{}` construct.

2.3 The Complex Event Network

The efficient matching of production rules is usually achieved through a discrimination network, like RETE, TREAT, etc. DEVICE smoothly integrates a RETE-like discrimination network into an active OODB system as a set of first class objects by mapping each node of the network onto a complex event objects of the active database system.

2.4 Production Rule Firing

When the last event in the DEVICE network signals, the recipient is the ECA rule that corresponds to the original production rule. The signal means that the production rule condition is satisfied and it must be selected for firing. The rule instantiation token is then forwarded to the production rule manager, which stores it in a data structure, that is traditionally called the "conflict set".

When multiple matching rule-condition tokens are placed in the conflict set there is an ambiguity concerning the number and the order of rules to be executed. The OPS5 approach is to apply heuristic conflict resolution strategies, namely refractoriness, recency and specificity, to select a unique rule instantiation to fire. In order to bring the full functionality of production systems into an active database system, the conflict resolution strategies have been incorporated into the rule selection mechanisms of the integrated environment. The centralised approach of rule selection eases such an implementation. The rule manager maintains a set-valued attribute that stores the conflict set, along with various other flags that control the rule conflict resolution mechanism.

After the rule manager selects one rule for firing, the action part of the rule must be scheduled for execution. Since we rely on the procedural language of the host OODB system (ADAM), it is trivial to execute a rule action as a compound Prolog goal. We assume that a rule is executed as soon as it is selected from the conflict set, i.e. in immediate Condition-Action coupling mode.

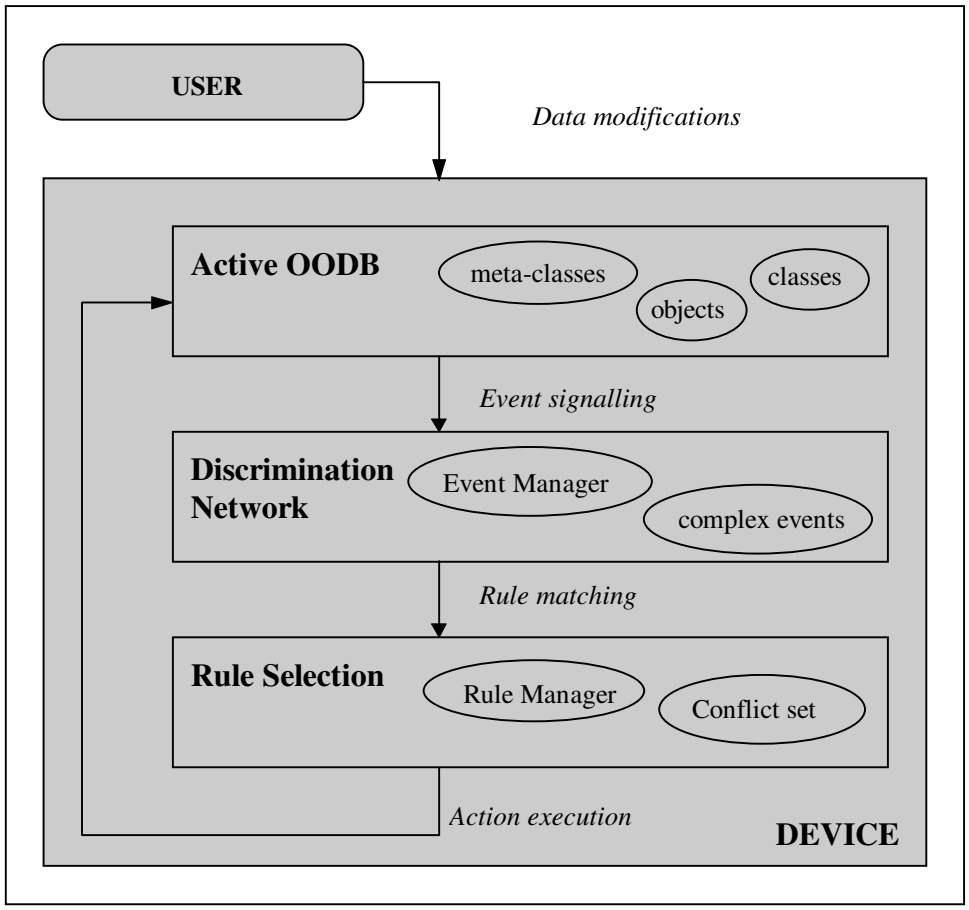
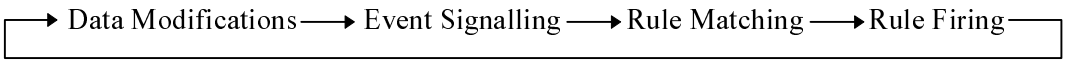


Figure 2: The Production Cycle of DEVICE.

On the other hand, the "deferred" coupling mode must be used between event detection and rule condition evaluation, in order to be able to execute multiple actions in one rule, without triggering and executing other rules in the medium. Simple and complex event detection due to action execution is simply recorded and the corresponding tokens are forwarded, without any further rule selection or execution. When the end of the transaction is signalled, the rule manager starts the rule selection and execution sequences. Figure 2 displays the above *production circle* of DEVICE, that is:



2.5 Extensibility of DEVICE

DEVICE is very easily extensible mainly because its OO architecture allows the transparent modification of existing system components or the introduction of new ones. Furthermore, the generality of the production and ECA rule managers allows the re-use of many of its components to support new rule types.

Deductive rules are an example of such an extension to the basic DEVICE system. Rule extensions can be added quite straightforward, because DEVICE components are Prolog modules and OODB meta-classes supplied with the adequate structure and behaviour to accomplish rule integration into EXACT. The whole architecture is not a layered but a modular one and new components are simply plugged into the existing system and extend its functionality. Figure 1 shows the relationships among the various system components.

When adding new rule types special care must be taken about the maintenance of the conflict set of the new rule manager. Specifically, if the new rule type is a sub-type of an existing one and they both have identical selection criteria, then the corresponding rule managers can share their conflict sets. This sharing is controlled by an attribute of the rule manager, which if set to *no* the rule manager is independent from its super-type and is activated individually by the rule scheduler.

3. User's Guide

DEVICE in its current state runs sequentially in SICStus Prolog and, from the user's point of view, consists of the following modules (Figure 3):

- The *main module* which is the actual Expert System Shell
- The *loader*, which is responsible for loading the knowledge base into the main system.

  When in use, three additional modules are used:
- The *schema*, in which all the user defined classes and methods related to the problem are defined.
- The *knowledge base*, which is a file containing the domain knowledge of the problem to be solved
- An *initialisation file* in which initial values for some problem parameters are given

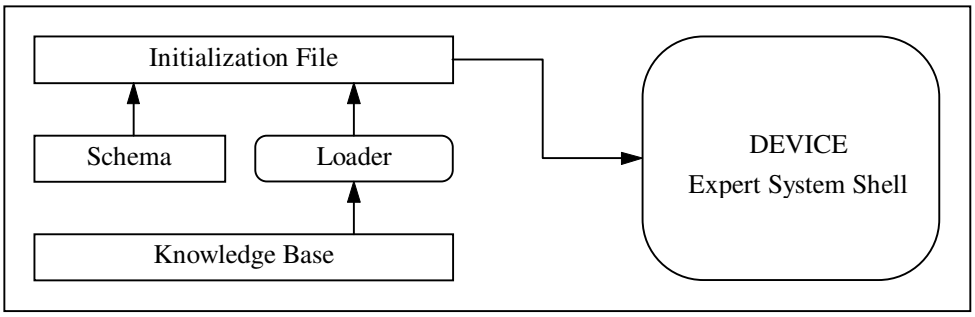


Figure 3: The modules of an expert system built on top of the DEVICE Expert System Shell.

The user, first starts the main module by typing `device` at the command prompt of a UNIX shell window. Then he consults the initialisation file. The first action defined in this file is the loading of the schema. Right after, the loader is invoked which loads the domain knowledge in to the main system. Then, initial values are given to wherever this is required and finally, the expert system starts by explicitly calling the `fc` command from within the initialisation file.

3.1 Defining the Schema

```
class1:-
     new([person,[
          slot(slot_tuple(name,global,single,total,string)),
          slot(slot_tuple(age,global,single,optional,integer)),
     ]]) => mm_entity_class.

class2:-
     new([emp,[
          is_a([person]),
          slot(slot_tuple(phone,global,single,optional,integer)),
          slot(slot_tuple(salary,global,single,optional,integer)),
          slot(slot_tuple(manager,global,single,optional,emp)),
          slot(slot_tuple(promotion_year,global,set,optional,integer))
     ]]) => mm_entity_class.

:- class1, class1.
```

Figure 4: The schema file for the bricks-world expert system.

In Figure 4, we give an example of a schema file. Two new object classes are defined, namely the `person` and `emp` ones. In DEVICE, objects are considered to be metaclasses, classes or instances. A

metaclass is an object the instances of which are all classes. When DEVICE starts, the metaclass called *mm_entity_class* already exists. All subsequent classes are created by sending messages to this metaclass. Messages are sent using the operator '=>' which has the generic syntax described in Figure 5. In our example, the method *new* of the metaclass *mm_entity_class* is called. This makes DEVICE aware of the two new classes. Note that, if the system cannot find a method to respond to a message sent to an object, an error is signalled.

Message::=	<method_name> params '=>' <message_recirient>
params::=	'(' elements ')' | []
elements::=	<res> ',' <input_list> | <input_list> | <res>
<method_name>	is a Prolog atom representing the name of method
<message_recirient>	is the unique identifier of an object
<input_list>	is a Prolog list of values passed into the method
<res>	is the result of the method

Figure 5: The generic syntax of the => operator.

In DEVICE, the attributes of an object and relationships between objects are stored in slots. A slot is accessible from methods attached to the class on which the slot is defined, and is not inherited by methods attached to its subclasses. Each slot has associated with it a *name*, a *visibility*, a *cardinality*, a *status* and a *result type*, defined in that order. In the schema of Figure 4, the class person has two slots, namely name and age. All two slots have the visibility attribute set to global.

The *visibility* given for the slot is used to define the visibility of its associated methods and its value may be `global`, `family` or `local`. By default, when a slot is created, DEVICE attaches automatically to the class on which the slot is defined, a number of methods to provide access to the slot. These methods permit values to be retrieved from the slot, added to the slot, deleted from the slot or replaced in the slot. The names of these methods are the name of the slot with the prefixes, `get_`, `put_`, `delete_` and `update_`. Thus, for the slot *name* of the class *person* in Figure 4, there already exist four methods: `get_name`, `put_name`, `delete_name` and `update_name`.

When a slot is declared with the visibility attribute set to global then, the above four methods are accessible without any restriction. If visibility is set to local these four methods are accessible only from inside the class. If visibility is family, then these methods are accessible from within the class and from any descendant classes. Note that, by default, the slots of a class are accessible only by using these four methods. Formally speaking, the updateability of the user defined slots is always local. Certain slots which are maintained by the system have an updateability of *system* - that is, programs and methods written by the users cannot update them, although they can be queried. This facility is used to guarantee the integrity of system-maintained slots as `is_a` and `instance_of`.

The cardinality attribute can be either single or set, that is, the slot can store only one or multiple values, respectively. When a slot is multi valued then querying the slot generates backtrack points.

The status attribute can be total, mandatory or optional. It indicates which update operations may be performed on the slot. A *total* slot is one which cannot be empty. That is, the slot must be given a value by the message which creates the object (new), and the value of the slot must never be deleted. Any attempt to delete the value from the slot results in deletion of the object itself. Thus any object with a total slot which contains a reference is dependent upon the object referred to by the total slot. A *mandatory* slot need not be given a value when the object is created, but once a value has been assigned to the slot, any attempt to delete the value results in deletion of the object to which the slot is attached. Only single-valued slots can be total or mandatory. An *optional* slot is one which can be empty on object creation. Furthermore, the value of the slot can be deleted at any time.

Finally, the result type attribute defines the data type of the value of a slot. Allowable types are `string`, `integer`, `float` and `<object_reference>`.

The hierarchical relationships between user defined classes are defined by placing an *is_a* slot in the descendant class. For example, in Figure 4, the class `emp` is of type `person`, as the `is_a([person])` slot declares.

3.2 Defining Production Rules

This section presents the system's rule language, which mainly follows the OPS5 [3] paradigm, influenced by the OODB context of DEVICE. All the rules of an expert system built with the DEVICE Expert System shell should be placed in the knowledge base file. The only restriction in the syntax of this file for successful loading by the loader module is that the rules must be enclosed in double quotes. Actually, whatever is enclosed in double quotes is considered to be a rule and is forwarded by the loader to the DEVICE main module for further processing. Anything else is considered to be comments and is therefore, rejected. Note that the loader modules does not perform any syntax checking. This is done by the main DEVICE module.

Production rules in DEVICE are expressed as a *condition*, which defines a pattern of objects to be detected over the database, followed by an *action*. The BNF syntax of the production rule definition language is given in Appendix A.

3.2.1 Rule Condition

The condition of a rule is an *inter-object* pattern, that consists of the conjunction of one or more (either positive or negative) *intra-object* patterns. Consider for example the following rule (which is based on the schema of Figure 4):

Rule 1:

```
IF   E1@emp(name='Mike', salary:S, manager:M1) and M1@emp(salary<S)
THEN delete ⇒E1
```

The intra-object pattern denotes the conjunction of two instances of class `emp`, that are connected to each other through two common attributes, denoted by the variables `E1`, `M1`. Disjunction of intra-object patterns is not included in the initial design for simplicity of implementation, since it is implied by the presence of multiple rules. The above rule deletes an employee named 'Mike' if his salary exceeds his/her manager's salary.

The intra-object patterns consist of one or more *attribute* patterns. The first of the above intra-object patterns denotes an instance `E1` of class `emp` with attribute `name` equal to `'Mike'`, with salary `S` and manager `M1`. The second intra-object pattern describes the manager `M1` of employee `E1`, who is himself/herself an instance of class `emp` and his `salary` attribute is less than the salary `S` of `E1`.

Variables in front of class names denote instances of the class. Inside the brackets, attribute patterns are denoted by relational comparisons, either directly with constants or indirectly through variables. Variables are also used to deliver values for comparison to other intra-object patterns (joins) in the same condition or to the action part of the rule. The values delivered by variables can be both object references and normal values, e.g. integers, strings. The variables are expressed as valid Prolog variables.

A limited form of disjunction inside attribute tests is allowed:

```
E1@emp(name='Mike' ; ='Nick')
```

where the above pattern denotes an employee whose name is either 'Mike' or 'Nick'. Similarly, conjunction can also be used inside attribute patterns. For example, the following pattern identifies a person with age more than 30 and less than 50:

```
P1@person(age>30 & <50)
```

Note that the condition of Rule 1 could be also written as:

```
E1@emp(name='Mike', salary:S1, salary.manager<S1)
```

therefore attribute patterns can navigate through object references of complex attributes, like the complex attribute `salary.manager`. The innermost attribute should be an attribute of class `emp`, i.e. the class

that the instances of the intra-object pattern stand for. Moving to the left, attributes should belong to classes related through object-reference attributes of the class of their predecessor attributes.

Notice that a right-to-left order of attributes has been adopted, contrary to the C-like dot notation that is commonly assumed, because we would like to stress out the functional data model origins of OODBs. Under this interpretation the chained "dotted" attributes can be seen as function compositions. During a pre-compilation phase each rule that contains complex attribute expressions, is transformed into one that contains only simple attribute expressions, by introducing new intra-object patterns. The above pattern is transformed into the actual condition of the Rule 1.

Intra-object patterns can also denote classes instead of class instances. For example, the following intra-object pattern "checks" whether the class variable `task`[1] of the class `production_rule` has the value `start`:

```
production_rule(task=start)
```

The difference between the two notations is just the presence of a variable in front of the class name. During parsing the above pattern is rewritten as follows:

```
X@production_rule_manager(task=start)
```

where `production_rule_manager` is the meta-class of `production_rule`. The new pattern is handled as a normal intra-object pattern.

There can also be negated intra-object patterns in the condition. A negated intra-object pattern denotes a negative condition that is satisfied when no objects in the database satisfy the corresponding positive intra-object pattern. For example, the following rule does not allow 'Mike' to have alone the largest salary:

Rule 2:

```
IF   E1@emp(name='Mike',salary:S1) and
     not E2@emp(salary>S1) and prolog{NewSal is 0.9 * S1}
THEN update_salary([S1,NewSal]) ó E1
```

We notice that only safe rules are allowed, i.e. a) variables that appear in the action must also appear at least once inside a non-negated condition, and b) variables that are used inside a negated condition must also appear at least once inside a non-negated condition, otherwise they are just ignored.

The choice for the logic-like condition language is justified by the fact that the condition is supposed to be a declarative specification of the state of the database and therefore it is not appropriate to use the procedural interface of the OODB as the condition language. However, the use of arbitrary Prolog or ADAM goals to express some small static conditions or to compute certain values is allowed in the condition through the special `prolog{}` construct.

Note that, currently, only attribute names can be used in the condition and not general methods. Instead, derived attributes can be used in condition expressions, since the matching is done transparently through event detection, as we shall see later.

3.2.2 Rule Action

The action part of a production rule defines a set of updates to be performed on the database objects that were identified in the rule condition. The action is expressed in Prolog extended with the default procedural data manipulation language of ADAM. Examples of production rule actions were given in the previous section. In Rule 1 the employee named 'Mike' is deleted when his salary is higher than his manager's salary, whereas in Rule 2 Mike's salary is lowered by 10%, when no other employee has a higher salary.

[1] In Section 4.3, we demonstrate how this variable can be used to clusterize the user defined rule set and simplify the rule selection.

3.3 Initialising Object Instances

Instances of objects can be created by using the `new` method (constructor) of the associated class. Next, we give some examples to demonstrate how the user can create object instances for the two most important built in classes of DEVICE, as well as for user-defined classes.

Given the schema of Figure 4, the following statement creates an instance of the object class `emp` and assigns values to slots `name` and `salary`.

```
new([E1,[name(['Mike']),salary([40])]]) => emp
```

Since the `name` slot of the class `emp`, which is inherited by the ancestor class `person`, has the *status* attribute equal to `total`, assigning a value to that slot is compulsory. On successful creation of the object instance, the variable E1 will be bound to an OID value (Object IDentifier).

To create/add a new rule into the system, a call is made to the `new` method of the built-in class `mm_production_rule`. The following statement:

```
new([R,[rule(["if   E1@emp(name='Mike', salary:S1) and
                    E2@emp(name='John', salary:S2>S1) and
                    prolog{NewSal is 2*S2}
              then  update_salary([S1,NewSal]) => E1"]),
              is_it_enabled([yes])
     ]])  =>   mm_production_rule
```

asserts a new rule into the rule base which declares that: "If there exists an employee `E1` named Mike, who has a salary `S1`, and there exists another employee `E2` named John with a salary `S2>S1` then double Mike's salary. The `is_it_enabled([yes])` parameter, tells device to immediately enable the rule. When the above method is successfully executed, the `R` variable is bound to an OID value.

Finally, we remind that, to create a new object class, a call to the `new` method of the built-in class `mm_entity_class` is required. For the completeness of this section, we repeat such a case from the examples of Figure 4.

```
new([person,[slot(slot_tuple(name,global,single,total,string)),
             slot(slot_tuple(age,global,single,optional,integer)),
     ]]) =>  mm_entity_class.
```

3.4 Development and Debugging Facilities

A number of development and debugging facilities have been implemented to ease the construction of expert systems in the DEVICE Expert System shell. These include:

1) **Development Facilities**

- *Loader* facility for consulting a knowledge base (set of rules) from a knowledge base file.

  This also enables the user to write directly the rules of the expert system without required to embed them as a parameter in the proper method. For example, instead writing:

  ```
  new([R,[rule(["if     E1@emp(name='Mike', salary:S1) and
                        E2@emp(name='John', salary:S2>S1) and
                        prolog{NewSal is 2*S2}
                  then update_salary([S1,NewSal]) => E1"]),
         is_it_enabled([yes])
         ]])  =>   mm_production_rule
  ```

  the user writes just the quoted rule, that is: `"if E1@emp(name... then ... => E1"`

  In the following we describe, informally, the syntax that a knowledge base file should follow, in order to succesfully be loaded into DEVICE by the LOADER module.

  The LOADER is invoced by a call to the top-level predicate which is:

  ```
  load_kb(KBFile).
  ```

where `KBFile` is the filename of a file that comforms to the syntax that is described bellow.

1. Comments

Comments are allowed in the same fashion as in most Prolog systems. That is, whatever is enclosed between `/*` and `*/` or whatever follows the `%` char until the end of the line is considered to be comment text.

2. Reserved Symbols

Char `#` is used to designate a header (see bellow) and is reserved.

3. Structure of a KB

A KB starts with a header

```
#start.
```

and ends with the header

```
#end.
```

Therefore, whatever text is before `#start.` and after `#end.` is ignored, even if it is not commented.

In between these two headers the user is allowed to define rules and modules.

3.1 RULES

A rule should introduced with a

```
#rule(Name, Status, Priority).
```

header, where `Name` is an atom used as a rule label, `Status` should be either `'control'` or `'_'` and `Priority` is an integer defining the priority of the rule. Note that all three parametrs are optional.

The body of the rule should follow the ADAM notation, must be enclosed in double quotes and it is possible to extend to more than on lines.

The end of a rule is designated by using the following header:

```
#endrule.
```

A rule could belong to one or none modules.

3.2 MODULES

A module includes none, one or more rules and is introduced with a

```
#module(Name, Priority).
```

header, where `Name` is an atom used as module label and `Priority` is an integer defining the priority of the module. The `Name` argument must always have a value. The `Priority` argument is not currently used by DEVICE and should be always `'_'`. A module section end with the following header:

```
#endmodule.
```

4. Generic Structure of a KB file

```
#start.

     #rule(foo,_,_).       % rule not belonging to a module
          "if ..............
           then ........"
     #endrule.

#module(mod1,_).           % a dummy module
#endmodule

     #rule(hello,_,_).% another rule not belonging to a module
          "if ..............  then ........"
     #endrule.

/* this is some comments */
% some more comments

#module(mymodule,_).
     #rule(hello1,_,_).       % a rule belonging to a module
          "if ..............  then ........"
     #endrule.

     #rule(hello2,_,_).
          "if ..............  then ........"
     #endrule.
#endmodule.

#end.
```

NOTE: Optional arguments inside headers must have a value of '_'.

4. Error messages.

Possible error messages include:

- KB ERROR: End of file reached while waiting for any of `[list of headers].`
- KB ERROR: Any of `[list of headers]` expected but [header] found.
- KB ERROR: `End_Of_Stream` reached during rule reading.

The possible reason of these messages is straightforward.

Please, note that the LOADER does not perform any rule processing. This is done latter by DEVICE.

1) **Debugging Facilities**

- Facilities for examining the structure of the discrimination network for a certain rule.

  `display_net => OID`

  For example, the following message examines the structure, i.e. the complex event network interconnection for rule with OID `21#mm_production_rule`:

```
display_net => 21#mm_production_rule
```

The network for this rule is shown in Figure 6.

```
   The DEVICE network of rule  21#mm_production_rule  is:
21#mm_production_rule
     88#mm_positive_event  --  Cartesian
          87#mm_positive_event  --  Loc
               80#mm_positive_event  --  Loc1
                    79#mm_positive_event  --  Loc1
                         69#mm_logical_event  --  _8498=possible
                              53#mm_inheritable_ms_event  --  put_status at location
                              53#mm_inheritable_ms_event  --  put_status at location
                              53#mm_inheritable_ms_event  --  put_status at location
                         76#mm_logical_event  --  _15635=no
                              60#mm_inheritable_ms_event  --  put_focused at location
                              60#mm_inheritable_ms_event  --  put_focused at location
                              60#mm_inheritable_ms_event  --  put_focused at location
                    55#mm_inheritable_ms_event  --  put_downStream at location
               78#mm_positive_event  --  Loc
                    69#mm_logical_event  --  _25931=possible
                         53#mm_inheritable_ms_event  --  put_status at location
                         53#mm_inheritable_ms_event  --  put_status at location
                         53#mm_inheritable_ms_event  --  put_status at location
                    75#mm_logical_event  --  _33068=yes
                         60#mm_inheritable_ms_event  --  put_focused at location
          77#mm_logical_event  --  _36199=trace_back
               11#mm_inheritable_ms_event  --  put_task at mm_production_rule_manager
```

Figure 6: The network for the rule `display_net=>21#mm_production_rule`

We can see that at the top-left corner the OID of the rule appears. Right below of the rule OID (and slightly displaced on the right) the last event of the discrimination network appears. Then each pair of event that is joined in a two-input event appears slightly displaced on the right, but in the same column. The variable on which the join is performed is also included. Furthermore, some information on the logical and primitive events is also included, such as the test of the logical event and the method and class that triggers each primitive event that positively activate the network.

This network can be used to set the appropriate event spypoints in order to check why a certain rule is or is not activated when it should. In the future, if appropriate graphical facilities are available, this network will appear in a more friendly way.

Alternatively, the user can use the name of the rule instead of its OID:

```
display_net(trace_path)
```

- Facility for examining the run-time contents of the conflict set.

  `spy_cs/1`, `nospy_cs/1`

  With this predicate we can set-on/off the conflict set spying facility for a certain rule manager, i.e. for a certain rule type. For example, the call `spy_cs(mm_production_rule)` spies the conflict set of production rules only, where `mm_production_rule` is the name of the rule manager for production rules. To avoid such lengthy rule manager names one can use `spy_cs(all)` to spy the conflict set of all rule managers at once.

  In Figure 7 it is shown the output of the conflict set spying facility for one production cycle. At each cycle the rule instantiations that enter the conflict set are shown along with the rule instantiation that has been selected for execution. Notice that the latter can be an instantiation that has been entered the conflict set at previous cycles.

- Facility for examining the input/output ports of the discrimination network nodes.

  `spy => EventOID, no_spy => EventOID`

```
   Cycle no: 25
   New rule instantiations:
   (22#mm_production_rule)-
positive([var(Loc,[11#manhole]),var(LocName,[manhole11]),var(Loc1,[13#manhole]),var(
Loc,[11#manhole])])
   (21#mm_production_rule)-
positive([var(Loc,[10#station]),var(Loc1,[12#manhole]),var(Loc,[10#station])])
   (25#mm_production_rule)-
positive([var(StoreName,[mon_station]),var(Loc,[10#station])])
   (26#mm_production_rule)-positive([var(Loc,[11#manhole])])

   Executing rule instantiation:  (22#mm_production_rule)-
positive([var(Loc,[11#manhole]),var(LocName,[manhole11]),var(Loc1,[13#manhole]),var(
Loc,[11#manhole])])
```

Figure 7: Output of the conflict set spying facility for one production cycle

This message, sent to a logical or two-input event, will display the tokens that this event inputs and outputs throughout the inferencing process. If an input token cannot be propagated, after it is processed at a node, then the phrase 'No Ouput!' is displayed. In Figure 8 a sample output of spying the event `85#mm_positive_event` is shown.

When more than one messages are to be spied the user can use the following commands instead:

`spy_events(List of Events), no_spy_events(List of Events)`

In the future, given the availability of advanced graphical facilities, this output may be improved.

In addition to spying events by their OID, the user can also spy the entire network of a certain rule by giving its name:

`spy_net(RuleName), no_spy_net(RuleName)`

Furthermore, the user can declare the depth at which the network is to be spied, in order to prune some unwanted outputs.

`spy_net(RuleName,Depth), no_spy_net(RuleName,Depth)`

- Finally, it is possible to watch when the current rule module changes by issuing the following command:

  `spy_tasks, no_spy_tasks`

  The task can be changed from the action of a production rule (preferrably by a rule that has been declared as control rules) using the following predicate:

  `switch_task(NewTask)`

  Examples are given in the next chapter.

- Another utility that can be used from inside the `prolog{}` part of the rule in order to ask a

```
Event: 85#mm_positive_event Left-Input:
occurrence_tuple(78#mm_positive_event,positive([var(Loc1,[29#store]),var(Loc,[18#man
hole])]))
Event: 85#mm_positive_event No Output!
----
Event: 85#mm_positive_event Right-Input:
occurrence_tuple(76#mm_positive_event,positive([var(Loc,[10#station])]))
Event: 85#mm_positive_event Output:
occurrence_tuple(85#mm_positive_event,positive([var(Loc1,[12#manhole]),var(Loc,[10#s
tation])]))
```

Figure 8: A sample output of spying the event `85#mm_positive_event`

question to the user is:

```
query(Question,Answer)
```

where `Question` is a list of terms to be displayed on the screen constituting a question, and `Answer` is a variable that receives the answer, which should be one character long.

4. Evaluation: The Chemical Spill

Apart from quite a few single rule examples that have been used for debugging purposes, a tiny expert system with 5 rules and 2 user-defined classes, and a small expert system consisted of 29 rules and 6 user-defined classes have been implemented to test and evaluate the DEVICE. In this section we present in detail the latter.

4.1 Problem Statement.

In the drainage system displayed in Figure 10, we want to identify the kind and the source of a possible spill, which is detected at the monitoring station with certain sample analysis procedures. The source of the spill could be one or more of the chemicals stored at the stores.

The human expert works as follows. He first identifies the chemicals by matching the observations against the known properties of the chemicals (Table 1). This may or may not uniquely identify the spill. He then works back up the drainage system from the monitoring station. His knowledge of where the chemicals are stored may allow him to constrain the possible route. However, if this information doesn't uniquely identify the source, he may have to open a manhole to see if the spill is present there; we assume that this is done by simple inspection.

4.2 Schema

We have defined six object classes to model the problem (Figure 9). The class **`chemical`** models the information displayed in Table 1. An additional slot (`status`) has been added to store information regarding the possibility of presence or absence of a particular chemical at the monitoring station. The value of this slot could be either `possible` or `impossible`.

The class **`measurement`** will be used to make the system aware of the results of the chemical analysis at the monitoring station. Since the human expert uses linguistic values for all measurements, we have add two slots, namely `value` and `descr`, that will be used as follows:

- For all measurements for which the monitoring station gets linguistic values we use the `descr` slot to store that value.
- For all measurements for which the monitoring station gets numerical values we use the `value` slot to store that value.

Note that, these two slots have the *status* attribute set to `optional`, that is, it is not required to assign values to that slots when we create object instances of type `measurement`. Later on, we will see that there exist some rules in the knowledge base that convert numerical values to linguistic ones, that is, assign a value to the slot `descr`, based on the value of the slot `value` and proper conversion rules.

The rest four classes are used to model the topology of the problem, that is the stores, the manholes, the monitoring station and the draining system. Note that the classes **`store`**, **`manhole`** and **`station`** are all descendants of the class **`location`** which has the following slots:

locName: the name of the location, used to distinguish the various locations.

status: a flag to denote the presence or absence of the spill at this location; it could be either `possible` or `impossible` respectively.

downStream: it holds the OID of the next location down the way from the stores to the monitoring station.

focused: a flag to mark if a location has been checked or not; it could be either `yes` or `no` respectively.

answer: when a location should be checked (by simple inspection) for the presence of absence of the spill, this slot stores the result; possible values are `y` (yes) or `n` (no) respectively.

In addition to the above slots which are inherited, the `store` class has an extra slot named `chemicals`, that is used to store the OIDs of the chemicals stored in a store. Since it is possible to have more than one chemicals in each store, the *cardinality* of this slot has been set to `set`.

```
class1:- new([chemical,[
     slot(slot_tuple(name,global,single,total,string)),
     slot(slot_tuple(ph,global,single,total,string)),
     slot(slot_tuple(solubility,global,single,total,string)),
     slot(slot_tuple(spectrometry,global,single,total,string)),
     slot(slot_tuple(colour,global,single,total,string)),
     slot(slot_tuple(smell,global,single,total,string)),
     slot(slot_tuple(gravity,global,single,total,string)),
     slot(slot_tuple(radioactivity,global,single,total,string)),
     slot(slot_tuple(status,global,single,total,string)) ]])=> mm_entity_class.

class2:- new([measurement,[
     slot(slot_tuple(mname,global,single,total,string)),
     slot(slot_tuple(value,global,single,optional,float)),
     slot(slot_tuple(descr,global,single,optional,string)) ]])=> mm_entity_class.

class3:- new([location,[
     slot(slot_tuple(locName,global,single,total,string)),
     slot(slot_tuple(status,global,single,optional,string)),
     slot(slot_tuple(downStream,global,single,optional,location)),
     slot(slot_tuple(focused,global,single,optional,string)),
     slot(slot_tuple(answer,global,single,optional,string)) ]])=> mm_entity_class.

class4:- new([store,[
     is_a([location]),
     slot(slot_tuple(chemicals,global,set,total,chemical)) ]])=> mm_entity_class.

class5:- new([manhole,[
     is_a([location]) ]])=> mm_entity_class.

class6:- new([station,[
     is_a([location]) ]])=> mm_entity_class.

:- class1, class2, class3, class4, class5, class6.
```

Figure 9: The schema for the "Chemical Spill" expert system.

4.3 Knowledge Base

The complete set of the 29 rules for the "Chemical Spill" expert system is given in Appendix B. The rules are referenced by their name.

The knowledge base is split into a number of tasks/modules:

`module start`

This is the initial module/task at which only the rules `ph_*` and `grav_*` are possible to execute. Rules `ph_*` handle the conversion of numerical value of pH to a linguistic one according to the information provided in Table 2. The same holds for rules `grav_*`, but this time for the gravity sensitivity values. The control rule `start_control` is fired after all conversions have been done and it changes the current module to `identify`.

`Module identify`

This module consists of rules `ident_*`. During this task the system decides which of the stored chemicals are impossible to be present in the spill, based on the results of the sample analysis. The control rule `ident_control` is fired at the very end, and changes the current module to `mark_possible`.

`Module mark_possible`

This module constitutes of the rules `mark_*`. The rule `mark_store`, checks for store locations that contain a spilled chemical and changes their *status* from `impossible` (initial value) to `possible`. Rule `mark_loc` marks as possible any location which is downstream for a location with *status*=`possible`. Control rule `mark_control` switches module to `trace_back` and is executed only when the two other rules are not possible to do so.

`Module trace_back`

During this task, the system tries to find a way back from the monitoring station to a possible source of the spill. If there is more that one pathways on the way back (rule `trace_cross`), the system asks the user to check a manhole, gets a response to this question, stores the answer into the *answer* slot of the manhole instance under question, and changes the task to `handle_reply`. If, on the other hand, there is just one possible pathway on the way back to a store, then rule `trace_path` traces it back automatically.

Note here how the slot *focused* is used. Having as possible values `yes` or `no` it is used to denote which is the location under consideration on the draining system.

Rule `prune` is fired when the location `Loc` with `status=impossible` which is under consideration, is the `downstream`, of another location `Loc1` for which it is also `status=possible`. The result is the update of the later location's status from possible to impossible and the shift of focus from the former to the later location.

Rule `write_store` announces the result when we get to a location which is a store. Rule `remove_focus`, on the other hand, removes the focus from stores that are not possible sources of the spill.

When all possible locations on a pathway from the monitoring station back to the stores, have been considered, then rule `task_control` changes the current module to `end`.

`module handle_reply`

It consists of rules `pos_reply`, `neg_reply` and `reply_control`. These handle user's reply received by rule `trace_cross`. More specifically, `pos_reply` handles the positive (y) answer and `neg_reply` the negative one. The module changes back to `trace_back` from the control rule `reply_control`.

`module end`

The only rule here is `write_chem`, which checks for chemicals that have the value of their `status` slot set to `possible`, and announces their name on screen.

	Name	pH	Solubility	Spectrometry	Colour	Smell	Gravity	Radioactivity	Store Location
1	Sulphuric_Acid	strong_acid	soluble	sulphur	clear	none	equal	none	S1, S5, S7
2	Hydrochloric_Acid	strong_acid	soluble	none	clear	choking	equal	none	S2, S7
3	Acetic_Acid	weak_acid	soluble	none	clear	vinegar	equal	none	S2, S4, S8
4	Carbonic_Acid	weak_acid	soluble	carbon	clear	none	equal	none	S4, S6, S8
5	Sodium_Hydroxide	strong_base	soluble	sodium	clear	none	equal	none	S8
6	Chromogen_23	strong_base	soluble	none	red	none	lighter	none	S5
7	Aluminium_Hydroxide	weak_base	soluble	none	white	none	heavier	none	S6
8	Rubidium_Hydroxide	weak_base	soluble	metal	clear	none	heavier	present	S3
9	Petrol	neutral	insoluble	carbon	clear	none	lighter	none	S1, S4, S5
10	Transformer_oil	neutral	insoluble	carbon	clear	none	equal	none	S3, S6

Table 1: Features of the chemical in the "Chemical Spill" problem

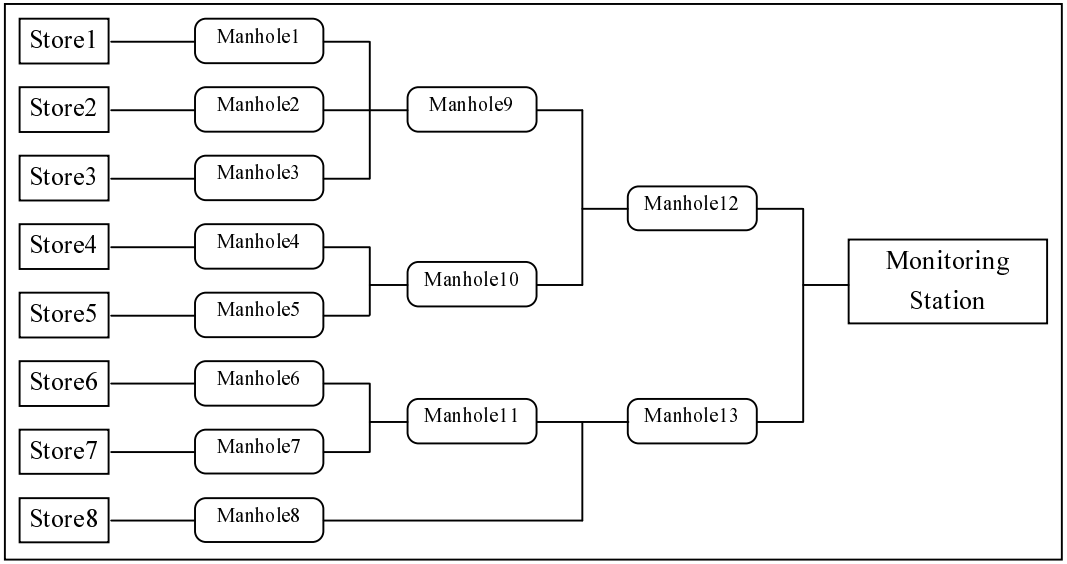


Figure 10: The drainage system in the "Chemical Spill" problem.

pH value	pH type	grav.sens. value	grav. type
pH<3	strong_acid	gravity<1	lighter
3≤pH<6	weak_acid	gravity=1	equal
6≤pH<8	neutral	gravity>1	heavier
8≤pH<11	weak_base		
11≤pH	strong_base		

Table 2: Rules to convert numerical values of pH and gravity

4.4 Test Session of the Chemical Spill.

In this section we present a slightly modified (commented) screen dump of the "Chemical Spill" expert system running. This test session is based on the input data located in Appendix C. This is actually the initialisation file. Here we create all the instances of the objects declared in the schema file, that is instances of `chemical`, `station`, `manhole`, `store` and `measurement` classes. The instances of measurement we create correspond to features of the chemicals that exclude all but the transformer_oil as possible spilled chemical. According to the content of the stores, the possible sources of the spill are stores 3 and 6.

```
?- test.
```

The loader outputs messages like the following (29 in total), for each rule that is successfully initialised.

```
Rule ph_strong_acid inserted
Rule ph_weak_acid inserted
Rule ph_neutral inserted
Rule ph_weak_base inserted.
.....
```

All the rules have been initialised and now starts the creation of the various object instances.

```
Chemicals loaded.
Topology loaded.
Measurements loaded.
```

All object instances have been successfully created and now the `fc` command is issued which starts the inference.

```
Starting...

Switching task from start to identify
Switching task from identify to mark_possible
```

Here the initialization has been performed and the possible chemicals that are present in the spill have been identified and the system marks the stores that contain these chemicals as possible sources of the spill (rule `mark_store`). In our example the `transformer_oil` has been identified.

```
changing store6 status because of transformer_oil
changing store3 status because of transformer_oil
```

Starting from Store3 which has been already marked as a possible source of the spill, the system follows the way down to the monitoring station and marks all the intermediate locations as possible (Rule `mark_loc`)

```
changing manhole3 status from impossible to possible
changing manhole9 status from impossible to possible
changing manhole12 status from impossible to possible
changing mon_station  status from impossible to possible
```

Starting from Store6 which has been already marked as a possible source of the spill, the system follows the way down to the monitoring station and marks all the intermediate locations as possible (Rule `mark_loc`)

```
changing manhole6 status from impossible to possible
changing manhole11 status from impossible to possible
changing manhole13 status from impossible to possible
```

The mark_possible task is over and the system switches to trace_back (rule `mark_control`)

```
Switching task from mark_possible to trace_back
```

Starting from the monitoring station, which is the initially focused location the system follows the way back to the stores. Since there are tow possible ways to follow, rule `trace_cross` is fired and the system asks the user to check on of them and reply to the question displayed right after. Note that after this point the task is switched to `handle_reply`.

```
Is the spill present at manhole13 ? (y/n): n
```

Since we reply with `n` (no), rule `neg_reply` is fired next and makes manhole13 impossible. In addition it changes the task back to `trace_back`.

```
pruning manhole13
```

Now rule `prune` will be fired two times successively followed by rule `remove_focus` with the following results, respectively:

```
pruning manhole11
pruning manhole6
pruning store6
```

Now the system follows the alternative path at manhole12 and silently follows the way back to the store store3 (rule `trace_path`) and then announces the source of the spill (rule `write_store`).

```
The source of the spill is: store3
```

Now rule `task_control` changes the task to end and then rule `write_chem` announces the chemical which is possible for the spill.

```
The spill may be caused by transformer_oil
Finished!
yes
```

5. Future Work

We give here some guidelines regarding additional components and/or facilities that are required to enhance the factionality and the flexibility of the DEVICE Expert System shell within the framework of INCO but also in general. In (Figure 11), our current understanding of the way DEVICE will be integrated into ExperNet, is illustrated.

5.1 Concerning the facilities of the expert system shell

a) **Backward chaining rules:** Currently DEVICE has only forward-chaining production rules. Backward-chaining, Prolog-like rules can be currently implemented only through Prolog. However, if such rules are needed they will be incorporated fully in DEVICE. Each predicate will be a separate object that will start producing output results when sent the *deduce* message. This method will bind the variables of the head with the bodies of the alternative clauses for this predicate and will send the message deduce to each of the body literals. If the literal is a base class, i.e. a set of facts, then these are directly retrieved from the database. Otherwise, the deduce message behaves exactly the same, in a recursive manner. Backtracking exists, just like in Prolog. This will actually be a naive replica of the Prolog interpreter. The only "good" thing about it will be the use of the DEVICE language. Backward chaining rules require moderate effort to be implemented.

   Since backward chaining rules are mostly used for expressing goal-driven computations, their use in the shell can be replaced by forward chaining rules and task definitions. Each goal to be met will be a task that is inserted in the working memory. All rules that solve this goal reference this task in their condition. This strategy is called Means-Ends-Analysis and can be used for goal-driven computations. Rule modules can ease the definition of tasks, as described above.

b) **Uncertainty:** Uncertainty, in the form of Certainty Factors, could be implemented if there is room for uncertainty on the knowledge base of EXPERNET. However, it requires certain extensions to the inference engine of DEVICE. First of all the base classes (data) of DEVICE should become uncertain. This would be reflected by a number in the [-1..1] range. Then these certainty factors should be propagated throughout the discrimination network, therefore the structure of complex events and tokens should be extended with such a number. Furthermore, rules for combining certainty factors should extend the run-time behaviour of complex events. Finally, the rule manager must be extended to use the certainty of a rule instantiation in order to choose it for firing. Then, the derived facts should be given a certainty, combined possibly with previously derived certainty. All these could theoretically be implemented on top of DEVICE. However, we are not sure how much time would these require. Furthermore, the use of certainty would require the change of the nature of the rule action language. The OODB will not be used as much as currently and special new methods are needed in order to store certainty in the derived objects.

   However, uncertainty requires some truth maintenance mechanism to be implemented in order to update the certainty of derived facts. Deductive rules that have already been implemented support such a maintenance mechanism but without the use of uncertainty. Specifically, deductive rules defined derived objects that should be in the database when certain conditions hold. If the condition ceases to hold then the derived facts are retrieved. This mechanism is suitable for uncertainty; instead of retrieving derived facts, their certainty factor would be just updated.

c) **Default values:** In DEVICE classes and objects are completely different entities, unlike expert systems where frames are used uniformly for the equivalent of the above two entities. Specifically, classes are prototypical frames and objects are instances of prototypical frames that fill some of the slots with values while they inherit default values for the rest of the slots. In OODBS, values are not inherited from a class to an object. However, ADAM is quite extensible to allow the

definition of new meta-classes that support default values. A new meta-class can be created that allows, when an object is created, certain slots of the object to be empty and to be filled with default values stored at the class definition.

d) **Meta-rules:** Meta-rules are extremely difficult to be implemented right-now, because they require a new meta-level inference engine to be built. The current engine cannot be used because the discrimination network is built around event of first-class objects, while rule instantiations in the conflict set are mere data structures stored in a slot. Major language modifications are needed. However, we believe that control rules will provide enough expressibility to define certain conflict resolution knowledge explicitly.

e) **Graphical user interaction:** When the graphical/windowing facilities of CSP-II are implemented then, the interaction with the user (questions-answers) will be done through windows. This could also improve the explanation and debugging facilities of DEVICE.

f) **Explanation facilities:** The advanced graphical interface could allow the existence of how/why buttons at each dialog window. When the user hits the 'why' button in a question window the system will reply why it asks this question. This explanation can be a text attached to the rule or the question, if questions become 1st class objects.

When the user hits the 'how' button in a system answer window the system will reply how it reached such conclusion. Of course, this explanation facility needs certain objects to be extended in order to include explanation texts that will be displayed. Furthermore, a mechanism is needed to keep track of the inference process. This could be probably be implemented by flushing the key-points of the inference process to a log file that is properly formatted. When a conclusion is reached and the user asks 'how', then the log must be traversed backwards. Each rule must have a text attached to it that explains what its firing practically means. This 'texts' are displayed to the user one at-a-time. If the user is satisfied with the first few explanations, then he/she can cut off the sequence.

g) **Better debugging facilities:** Currently there are certain debugging facilities in DEVICE. However, in the light of the graphical interface, better debugging facilities can be developed that will help the knowledge base developer to focus and watch exactly to the suspicious, faulty point.

5.2 Concerning the interfaces of the expert system shell

In order to integrate DEVICE into ExperNet, some additional modules are required. These are described right after and are depicted in (Figure 11). Note how the OO approach followed by DEVICE, makes this integration smooth, by comparing Figure 2 with Figure 11.

a) **Interface with HNMS:** the database of DEVICE will reflect the database of HNMS, at least for the MIB variables that are needed. When the OODB schema is created, it will be declared which DEVICE objects will take values from HNMS variables. This will probably be achieved my creating a class through a special meta-class. The values defined in the schema will be subscribed in HNMS. When these variables change (in HNMS) it will forward the data to the subscriber (Prolog). A special interface module (Figure 11) will wait in the shell for such messages. When they arrive it will update the local database objects. The update of the local objects will trigger events and rules. In this way only a low-level mechanism is needed. DEVICE will not have to be altered. It will "see" the changes of the HNMS variables through the local database.

b) **Interface with remote DEVICE agents:** The same principle applies here. When a remote DEVICE agent (Figure 11) wants to communicate with another agent (communication means that a database object that belongs to the local agent is changed by a remote agent), then it sends the appropriate message to the appropriate CSP-II instance in the network. The interface will change the status of a local object accordingly and DEVICE will work as if the object were local.

c) **Interface with DBMS:** Currently objects reside only in the main-memory. When the CSP-II-DBMS interface is built, then some of the classes can be declared to be 'persistent'. This means that

they are read and written through an interface that re-directs the read-write messages to the database interface and the DBMS itself.

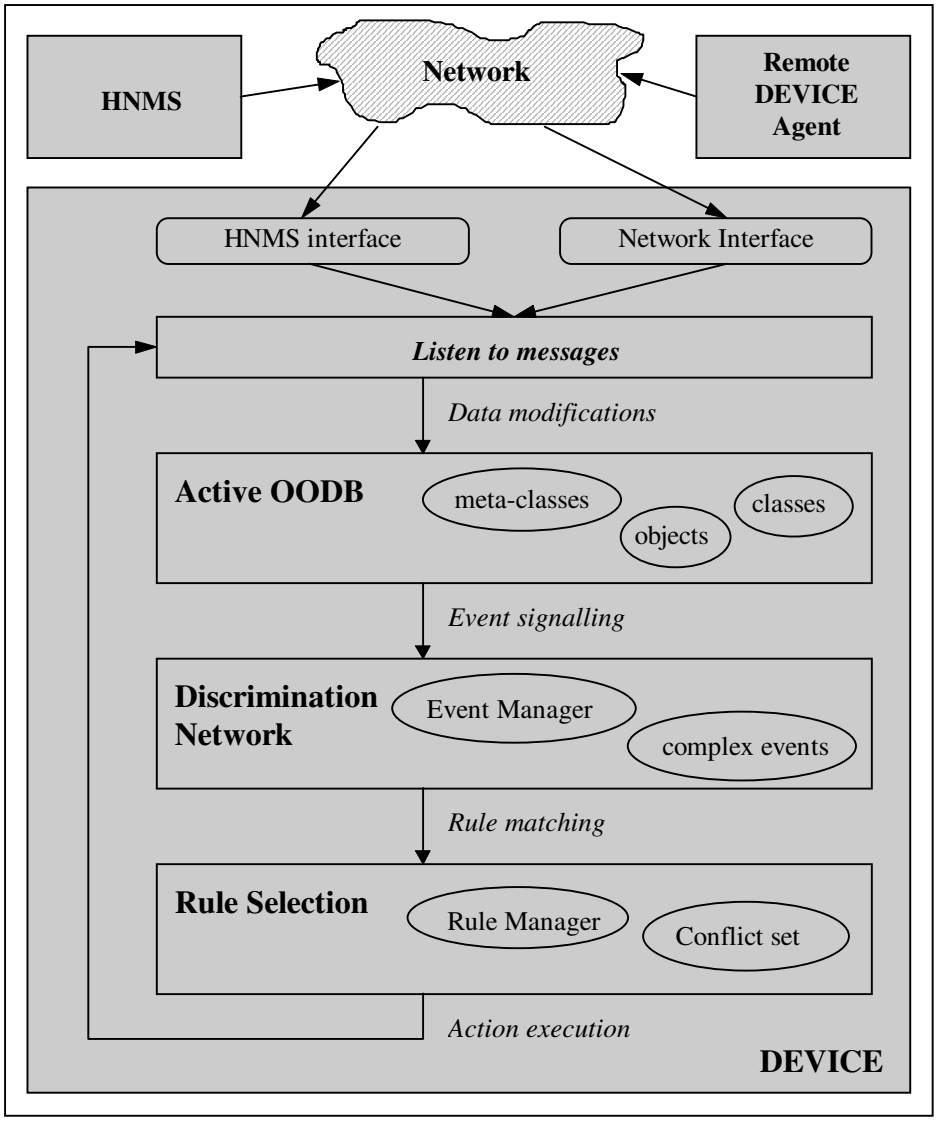


Figure 11: Suggested way to integrate DEVICE into ExperNet.

Appendix A - Production Rule Syntax

```
    <production_rule> ::=  if <condition> then <action>
          <condition> ::=  <inter-object-pattern>
<inter-object-pattern> ::=  <condition-element> ['and' <inter-object-pattern>]
<inter-object-pattern> ::=  <inter-object-pattern> 'and' <prolog_cond>
   <condition-element> ::=  ['not'] <intra-object-pattern>
<intra-object-pattern> ::=  [<variable>'@']<class>['('<attr-patterns>')']
       <attr-patterns> ::=  <attr-pattern>[','<attr-patterns>]
        <attr-pattern> ::=  <var-assignment> | <predicate>
        <attr-pattern> ::=  <attr-function>':'<variable> <rel-operator> <value>
      <var-assignment> ::=  <attr-function>':'<variable>
          <predicate> ::=  <attr-function> <predicates>
         <predicates> ::=  <rel-operator> <value> [{&|;} <predicates>]
       <rel-operator> ::=  = | > | >= | =< | < | \=
              <value> ::=  <constant> | <variable>
      <attr-function> ::=  [<attr-function>'.']<attribute>
        <prolog_cond> ::=  'prolog' '{'<prolog_goal>'}'
             <action> ::=  <prolog_goal>
              <class> ::=  An existing ADAM class
          <attribute> ::=  An existing attribute of the corresponding ADAM class
        <prolog_goal> ::=  An arbitrary Prolog/ADAM goal
           <constant> ::=  A valid constant of an ADAM simple attribute type
           <variable> ::=  A valid Prolog variable
```

Appendix B - The rules (KB) of the Chemical Spill expert system.

```
/************************************************/
/*     Problem: CHEMICAL SPIL                   */
/*    Encoding: Fotis Kokkoras & Nick Bassiliades */
/*     Purpose: Evaluation of DEVICE            */
/* No of Rules: 23                              */
/* Schema File: spil_l2.pl  (2 classes)         */
/*        Date: 19/12/97 - requires loader2.pl  */
/************************************************/


#start.


#module(start,_).

  /* 5 rules to classify ph values */

  #rule(ph_strong_acid,_,_).
       "if M@measurement(mname=ph, value<3.0)
        then put_descr([strong_acid])=>M"
  #endrule.

  #rule(ph_weak_acid,_,_).
       "if M@measurement(mname=ph, value>=3.0 & <6.0)
        then put_descr([weak_acid])=>M"
  #endrule.

  #rule(ph_neutral,_,_).
       "if M@measurement(mname=ph, value>=6.0 & <8.0)
        then put_descr([neutral])=>M"
  #endrule.

  #rule(ph_weak_base,_,_).
       "if M@measurement(mname=ph, value>=8.0 & <11.0)
        then put_descr([weak_base])=>M"
  #endrule.

  #rule(ph_strong_base,_,_).
       "if M@measurement(mname=ph, value>=11.0)
        then put_descr([strong_base])=>M"
  #endrule.


  /* 3 rules to classify gravity values */
```

```
  #rule(grav_light,_,_).
       "if M@measurement(mname=gravity, value<1.0)
        then put_descr([lighter])=>M"
  #endrule.

  #rule(grav_equal,_,_).
       "if M@measurement(mname=gravity, value=1.0)
        then put_descr([equal])=>M"
  #endrule.

  #rule(grav_heavy,_,_).
       "if M@measurement(mname=gravity, value>1.0)
        then put_descr([heavier])=>M"
  #endrule.


  /* 1 rule to change from start to identify */
   % The condition of this rule should be shortened by a keyword, e.g. 'true'
   % and replaced at parsing time by the following!

  #rule(start_control,control,_).
       "if-true then switch_task(identify)"
  #endrule.

#endmodule.


#module(identify,_).

  /* 7 rules to detect the type of spil - one for each chemical feature */

  /* case ph */
  #rule(ident_ph,_,_).
       "if M@measurement(mname=ph, descr:D)
           and C@chemical(ph\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.

  /* case  solubility */
  #rule(ident_solub,_,_).
       "if M@measurement(mname=solubility, descr:D)
           and C@chemical(solubility\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.

  /* case  spectrometry */
  #rule(ident_spectro,_,_).
       "if M@measurement(mname=spectrometry, descr:D)
           and C@chemical(spectrometry\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.
```

```
  /* case  colour */
  #rule(ident_colour,_,_).
       "if M@measurement(mname=colour, descr:D)
           and C@chemical(colour\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.

  /* case  smell */
  #rule(ident_smell,_,_).
       "if M@measurement(mname=smell, descr:D)
           and C@chemical(smell\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.

  /* case gravity */
  #rule(ident_grav,_,_).
       "if M@measurement(mname=gravity, descr:D)
           and C@chemical(gravity\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.

  /* case  radioactivity */
  #rule(ident_radio,_,_).
       "if M@measurement(mname=radioactivity, descr:D)
           and C@chemical(radioactivity\=D, status=possible)
        then update_status([possible,impossible])=>C"
  #endrule.


  /* 1 rule to change from identify to mark_possible*/
  #rule(ident_control,control,_).
       "if-true then switch_task(mark_possible)"
  #endrule.

#endmodule.


#module(mark_possible,_).

  /* 3 rules to mark the possible originals of the spil - left2right pruning
  /* if the store contains the spilled chemical, mark it possible */
  #rule(mark_store,_,_).
       "if Chem@chemical(status=possible, name:ChemName)
           and Store@store(chemicals=Chem, status=impossible, locName:StoreName)
       then nl, write_list(['changing',StoreName,'status because of',ChemName]),
             update_status([impossible,possible])=>Store"
  #endrule.

  /* mark as possible any location which is downstream for a possible location
*/
```

```
  #rule(mark_loc,_,_).
       "if Loc1@location(status=possible, downStream:Loc2)
           and Loc2@location(status=impossible, locName:LocationName)
        then nl,
             write_list(['changing',LocationName,
                         'status from impossible to possible']),
             update_status([impossible,possible])=>Loc2"
  #endrule.

  /* switch task to trace_back */
  #rule(mark_control,control,_).
       "if-true then switch_task(trace_back)"
  #endrule.

#endmodule.


#module(trace_back,_).

  /* 8 rules to trace back the store with the spill */

  /* if there is more than one then ask the user to check one manhole on a
possible pathway */
  % The following rule switches a task, although it is mainly a knowledge rule
  % Check later if this can be handled through a pire control rule
  #rule(trace_cross,_,_).
       "if Loc@location(status=possible, focused=yes) and
        Loc1@location(locName:LocName,status=possible,downStream=Loc,focused=no)
       and Loc2@location(status=possible, downStream=Loc, focused=no)\=Loc1 and
       prolog{query(['Is the spill present at',LocName,'? (y/n):'], Answer)}
        then put_answer([Answer])=>Loc1,
             switch_task(handle_reply)"
  #endrule.


  /* if there is just one possible pathway then trace it back  */
  #rule(trace_path,_,_).
       "if Loc@location(status=possible, focused=yes)
           and Loc1@location(status=possible, downStream=Loc, focused=no)
        then update_focused([yes,no])=>Loc,
             update_focused([no,yes])=>Loc1"
  #endrule.

  /* this one performs right to left pruning */
  #rule(prune,_,_).
       "if Loc@location(status=impossible, focused=yes) and
        Loc1@location(locName:LocName,status=possible,downStream=Loc,focused=no)
        then nl, write_list(['pruning',LocName]), nl,
             update_status([possible,impossible])=>Loc1,
             update_focused([yes,no])=>Loc,
             update_focused([no,yes])=>Loc1"
```

```
  #endrule.


  /* when we get to a location which is store, then announce the result */
  #rule(write_store,_,_).
       "if Loc@store(locName:StoreName, focused=yes, status=possible)
        then nl, write_list(['The source of the spill is: ', StoreName]), nl,
             update_focused([yes,no])=>Loc"
  #endrule.

  /* This rule just removes the focus from stores that are impossible */
  #rule(remove_focus,_,_).
       "if Loc@store(focused=yes, status=impossible)
        then update_focused([yes,no])=>Loc"
  #endrule.

  /* If no more focused possible stores exist then announce the results */
  #rule(task_control,control,_).
       "if not Loc@store(focused=yes, status=possible)
        then switch_task(end)"
  #endrule.

#endmodule.



#module(handle_reply,_).

  /* handle the positive (y) answer */
  #rule(pos_reply,_,_).
       "if Loc@location(focused=no, status=possible, downStream:Loc0, answer=y)
        then update_focused([yes,no])=>Loc0,
             update_focused([no,yes])=>Loc"
  #endrule.

  /* handle the negative (n) answer */
  #rule(neg_reply,_,_).
       "if Loc@location(locName:LocName, focused=no, status=possible, answer=n)
        then nl, write_list(['pruning',LocName]), nl,
             update_status([possible,impossible])=>Loc,
             update_focused([no,yes])=>Loc"
  #endrule.

  #rule(reply_control,control,_).
       "if-true then switch_task(trace_back)"
  #endrule.

#endmodule.
```

```
#module(end,_).

  /* 1 rule to print out the results at the end*/
  #rule(write_chem,_,_).
       "if C@chemical(name:N, status=possible)
        then nl, write_list(['The spill may be caused by ',N]), nl"
  #endrule.

#endmodule.



#end.
```

Appendix C - Initialisation file of the Chemical Spill expert system.

```
:-   [spil_schema].

test:-     spy_loader,
           load_kb('spil.kb'),
           load_data,
           write('Starting...'), nl,
           spy_tasks,
           fc,
           write('Finished!'), nl.

load_data:-
     new([C1,[name([sulphuric_acid]),
                ph([strong_acid]),
        solubility([soluble]),
       spectrometry([sulphur]),
              colour([clear]),
               smell([none]),
             gravity([equal]),
     radioactivity([none]),
              status([possible]) ])=> chemical,

     new([C2,[name([hydrochloric_acid]),
                ph([strong_acid]),
        solubility([soluble]),
       spectrometry([none]),
              colour([clear]),
               smell([choking]),
             gravity([equal]),
     radioactivity([none]),
              status([possible]) ])=> chemical,

     new([C3,[name([acetic_acid]),
                ph([weak_acid]),
        solubility([soluble]),
       spectrometry([none]),
              colour([clear]),
               smell([vinegar]),
             gravity([equal]),
     radioactivity([none]),
              status([possible]) ])=> chemical,

     new([C4,[name([carbonic_acid]),
                ph([weak_acid]),
        solubility([soluble]),
       spectrometry([carbon]),
              colour([clear]),
               smell([none]),
             gravity([equal]),
     radioactivity([none]),
              status([possible]) ])=> chemical,

     new([C5,[name([sodium_hydroxide]),
                ph([strong_base]),
        solubility([soluble]),
       spectrometry([sodium]),
              colour([clear]),
               smell([none]),
             gravity([equal]),
     radioactivity([none]),
              status([possible]) ])=> chemical,

     new([C6,[name([chromogen_23]),
                ph([strong_base]),
        solubility([soluble]),
       spectrometry([none]),
```

```
       colour([red]),
        smell([none]),
      gravity([lighter]),
radioactivity([none]),
       status([possible]) ]])=> chemical,

new([C7,[name([aluminium_hydroxide]),
           ph([weak_base]),
   solubility([soluble]),
 spectrometry([none]),
       colour([white]),
        smell([none]),
      gravity([heavier]),
radioactivity([none]),
       status([possible]) ]])=> chemical,

new([C8,[name([rubidium_hydroxide]),
           ph([weak_base]),
   solubility([soluble]),
 spectrometry([metal]),
       colour([clear]),
        smell([none]),
      gravity([heavier]),
radioactivity([present]),
       status([possible]) ]])=> chemical,

new([C9,[name([petrol]),
           ph([neutral]),
   solubility([insoluble]),
 spectrometry([carbon]),
       colour([clear]),
        smell([none]),
      gravity([lighter]),
radioactivity([none]),
       status([possible]) ]])=> chemical,

new([C10,[name([transformer_oil]),
           ph([neutral]),
   solubility([insoluble]),
 spectrometry([carbon]),
       colour([clear]),
        smell([none]),
      gravity([equal]),
radioactivity([none]),
       status([possible]) ]])=> chemical,

nl,write('Chemicals loaded.'),nl,

new([MS1,[locName([mon_station]),
           status([impossible]),
          focused([yes]) ]])=> station,

new([M13,[locName([manhole13]),
           status([impossible]),
       downStream([MS1]),
          focused([no])]])=> manhole,

new([M12,[locName([manhole12]),
           status([impossible]),
       downStream([MS1]),
          focused([no])]])=> manhole,

new([M11,[locName([manhole11]),
           status([impossible]),
       downStream([M13]),
          focused([no])]])=> manhole,

new([M10,[locName([manhole10]),
           status([impossible]),
       downStream([M12]),
          focused([no])]])=> manhole,
```

```
new([M9,[locName([manhole9]),
           status([impossible]),
       downStream([M12]),
          focused([no])]])=> manhole,

new([M8,[locName([manhole8]),
          status([impossible]),
      downStream([M13]),
         focused([no])]])=> manhole,

new([M7,[locName([manhole7]),
          status([impossible]),
      downStream([M11]),
         focused([no])]])=> manhole,

new([M6,[locName([manhole6]),
          status([impossible]),
      downStream([M11]),
         focused([no])]])=> manhole,

new([M5,[locName([manhole5]),
          status([impossible]),
      downStream([M10]),
         focused([no])]])=> manhole,

new([M4,[locName([manhole4]),
          status([impossible]),
      downStream([M10]),
         focused([no])]])=> manhole,

new([M3,[locName([manhole3]),
          status([impossible]),
      downStream([M9]),
         focused([no])]])=> manhole,

new([M2,[locName([manhole2]),
          status([impossible]),
      downStream([M9]),
         focused([no])]])=> manhole,

new([M1,[locName([manhole1]),
          status([impossible]),
      downStream([M9]),
         focused([no])]])=> manhole,

new([S1,[locName([store1]),
          status([impossible]),
      downStream([M1]),
         focused([no]),
       chemicals([C1,C9]) ]])=> store,

new([S2,[locName([store2]),
          status([impossible]),
      downStream([M2]),
         focused([no]),
       chemicals([C2,C3]) ]])=> store,

new([S3,[locName([store3]),
          status([impossible]),
      downStream([M3]),
         focused([no]),
       chemicals([C8,C10]) ]])=> store,

new([S4,[locName([store4]),
          status([impossible]),
      downStream([M4]),
         focused([no]),
       chemicals([C3,C4,C9]) ]])=> store,

new([S5,[locName([store5]),
          status([impossible]),
      downStream([M5]),
```

```
          focused([no]),
        chemicals([C1,C6,C9]) ]])=> store,

new([S6,[locName([store6]),
          status([impossible]),
       downStream([M6]),
          focused([no]),
        chemicals([C4,C7,C10]) ]])=> store,

new([S7,[locName([store7]),
          status([impossible]),
       downStream([M7]),
          focused([no]),
        chemicals([C1,C2]) ]])=> store,

new([S8,[locName([store8]),
          status([impossible]),
       downStream([M8]),
          focused([no]),
        chemicals([C3,C4,C5]) ]])=> store,

nl,write('Topology loaded.'),nl,

% ph  -  (0.0, 14.0)
new([_,[ mname([ph]),
          value([7.0]) ]])=> measurement,

% solubility - {soluble, insoluble}
new([_,[ mname([solubility]),
          descr([insoluble]) ]])=> measurement,

% spectrometry - {carbon, sulphur, metal, sodium, none}
new([_,[ mname([spectrometry]),
          descr([carbon]) ]])=> measurement,

% colour - {white, red, clear}
new([_,[ mname([colour]),
          descr([clear]) ]])=> measurement,

% smell - {choking, vinegar, none}
new([_,[ mname([smell]),
          descr([none]) ]])=> measurement,

% gravity - (0.9, 1.1)
new([_,[ mname([gravity]),
          value([1.0]) ]])=> measurement,

% radioactivity - {present, none}
new([_,[ mname([radioactivity]),
          descr([none])]])=>measurement,

nl, write('Measurements loaded.'), nl.
```