Software User Manual

for

A lightweight and modular expert system shell for the usage in decision support system

Version 1.7

Revision history

Version	Date	Description	Author
1.0	29.04.2011	Initial revision	Pavel Grigorenko
1.1	24.05.2011	Updated section 3.2	Pavel Grigorenko
1.2	25.06.2011	Added section 3.1.8	Pavel Grigorenko
1.3	30.06.2011	Added section 3.1.9	Pavel Grigorenko
1.4	25.09.2011	Extended section 3.2	Pavel Grigorenko
1.5	17.10.2011	Fixes	Pavel Grigorenko
1.6	15.11.2011	Updated table in 2.2	Pavel Grigorenko
1.7	16.11.2011	Added sections 4.2 and 4.3	Pavel Grigorenko

Table of Contents

1 Introduction

The present document represents the instruction manual for the users of the Expert System Shell and its integration with model-based software development platform CoCoViLa. CoCoViLa is used for implementing software packages for performing modeling and simulations and also decision making in various application domains. Expert System Shell is developed to enhance decision support applications in CoCoViLa.

In this section, the brief introduction to Expert Systems is given.

1.1 Expert systems

Expert system is defined as a problem-solving and decision-making system based on knowledge of its task and logical rules or procedures for using the knowledge. In other words, it is a computer program that simulates the judgment of a human that has expert knowledge and experience in some specific domain. The three basic components of an expert system are a knowledge base (where the knowledge is stored), an inference engine (which controls reasoning with the knowledge), and a user interface.

Expert system shell is a complete environment for developing, maintaining and using knowledge-based applications. Shell provides a graphical user interface for domain experts for encoding the knowledge and for engineers and analysts to use the expert system in the most convenient way.

Expert systems are a subfield and a practical application of Artificial Intelligence. Knowledge lies in the basis of Artificial Intelligence and Artificial Intelligence as science is concerned how knowledge can be represented and handled. Where are many different approaches and algorithms for knowledge representation and handling, such as brute force deduction and value propagation algorithms, resolution method and logic programming, production rules and decision tables. The present expert system implementation uses the two latter methods.

1.2 Production rules

Production rules are a well-known form of knowledge representation. A production rule is a condition-action pair of the form *(condition, action)* with the meaning: "If the *condition* is satisfied, then the *action* can (or must) be taken". Systems that employ production rules are called *production rule systems*. Such systems store rules in a *knowledge base,* a database which is aware of an order and a priority of rules to be triggered. Inference engine tries to match the input data to conditions of the production rules and decides which corresponding action of a suitable production rule to fire (i.e. execute). There are two basic execution strategies called chaining – forward chaining and backward chaining. Most of the expert systems use data-driven forward chaining algorithms for selecting productions to execute. A production rule can be executed only if its condition is satisfied. In fact, a condition of a rule may contain many sub-conditions and they all must hold.

For example, very trivial decision making production rule based expert system can depending on the weather conditions suggest what to do. A list of production rules is as follows:

if it is raining **then** the weather is bad;
if it the sun is shining **then** the weather is fine;
if the weather is bad **then** sit at home;
if the weather is fine **then** go for a walk;
if sit at home **then** read a book;
if go for a walk **then** feed the birds;

If a user inputs that it is raining, inference engine fires the rules that suggest to sit at home and read

a book. If the sun is shining, a user should go for a walk and feed the birds. Of course it is possible to extend the knowledge base with more rules for the system to be more precise. One of the advantages of expert systems is that the knowledge is not hard-coded, it is possible to edit the knowledge-base, add, update and remove rules using a user interface without a need to reprogram anything.

1.3 Decision tables

Expert systems are typically used in law, banking, medicine, but also in engineering domains. Engineering knowledge usually is very precise and is presented in textbooks using tables. Tables provide a clear and concise way of presenting large amounts of data in a small space.

For example, Table 1 is a table picked from a textbook of the engineering calculations of the regimes of the cutting on the machine tools. The table is meant for choosing a value of admissible feed for a machine tool depending on various parameters (IM – instrument material, FGr – feed group, σ – density, etc).

Table 1. Example of a table from an engineering textbook

$h = 8$, $s_{z\,adm}N = 5$

	IM = VK6		IM ≠ VK6	
FGr	σ ≤ 180	σ ≤ 1800	σ ≤ 180	σ ≤ 1800
1	0.28	0.24	0.38	0.32
≤ 3	0.56	0.48	0.76	0.64

The engineering knowledge from this table can be included into the expert system and encoded using the production rules:

```
if h = 8  and sz admN = 5  and FGr = 1  and σ ≤ 180  and IM = VK6
       then sz adm = 0.28;
if h = 8  and sz admN = 5  and FGr = 1  and σ ≤ 1800  and IM = VK6
      then sz adm = 0.24;
if h = 8  and sz admN = 5  and FGr μ ≤ 3  and σ ≤ 180  and IM ≠ VK6
      then sz adm = 0.76;
...
```

However this approach leads to a problem, because the number of rules may be very large and it will take too much effort to manage such a large set. A convenient way of storing such data in the knowledge base is by using so-called *structural decision tables*. The tables we are going to describe have advantages over conventional tables (as shown above) in the following – it is easier for a computer to read tables with the well-defined structure (Table 1 contains duplicate columns), it is easier for a programmer to implement such tables, and finally, structural tables can help a user to avoid ambiguity and mistakes while filling tables with data.

A structural decision table consists of three kinds of subtables: table of conditions, selection matrix and a table of values. Table of conditions includes atomic conditions for selecting production rules. Selection matrix combines them into complete conditions for selecting a rule by showing whether an atomic condition is required. Table of values contains results of a selection that can be either values or actions. The following figure shows the structure of a 2-dimensional structural decision table that contains two tables of conditions and two selection matrices.

Table 2. The template of a structural decision table

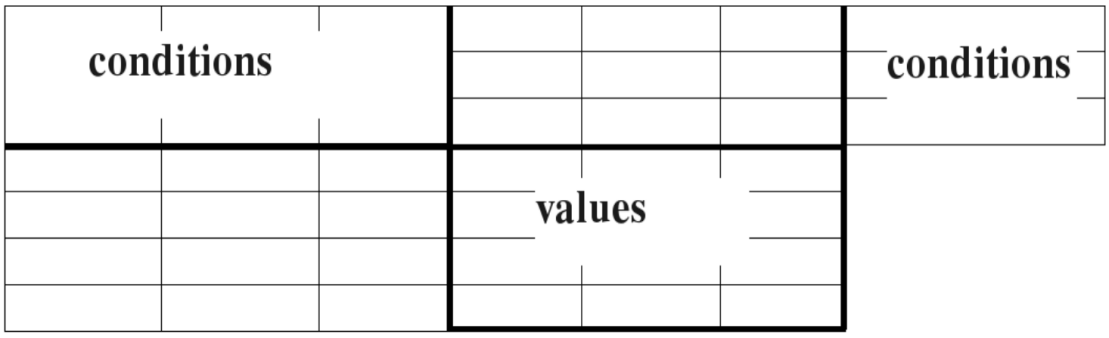


Table 3 is the corresponding decision table for the example above.

Table 3. Decision table with engineering data

				X		X		$\sigma \leq 180$
					X		X	$\sigma \leq 1800$
h = 8	FGr = 1	FGr ≤ 3	$s_{z\,adm}N = 5$	X	X			IM = VK6
X	X		X	0.28	0.24	0.38	0.32	
X		X	X	0.56	0.48	0.76	0.64	

It is easy to see in Table 3 that in order to choose a row and a column for selecting a value, all conditions marked with X have to be satisfied. Conditions that are not marked for a particular row or column are inessential and they are ignored.

2 Implementation

This section describes main implementation details of the core of the Expert System Shell (graphical components will be explained in the Section 3).

2.1 Data types

The expert system shell operates with inputs of certain datatypes that allow querying decision tables using external interfaces. The following immutable Java datatypes are supported:

- String
- int
- double
- long
- boolean
- float
- short
- byte

Input types can also be arrays of the types listed above.

2.2 Built-in predicates

The “If” part of a production rule is represented as a conjunction of predicates (atomic conditions). If all predicates evaluate to true, a production rule is picked by the shell. An empty list of conditions also evaluates to true. Supported predicates are listed in Table 4.

Table 4. The list of available predicates (X - input, V - value)

Predicate	**Icon**	**Input type**	**Type of value to compare against**	**Example**
Equals	=	Single value or array of any supported type	Same as input	X = 5
Not equal	≠	Single value or array of any supported type	Same as input	X ≠ 0
Less	<	Single value of any supported type	Same as input	X < 5
Less or equal	≤	Single value of any supported type	Same as input	X ≤ 10
Greater	>	Single value of any supported type	Same as input	X > 55
Greater or equal	≥	Single value of any supported type	Same as input	X ≥ 1
Contains	∈	Single value of any supported type	Array type of input	X ∈ [1,3,5,7]
Excludes	∉	Single value of any supported type	Array type of input	X ∉ [1,3,5,7]
Contains	∋	Array of any supported type	Element type of input	V ∈ X
Excludes	∌	Array of any supported type	Element type of input	V ∉ X
Matches regular expression	^$	String	Regular expression	X matches “^.*[a-zA-Z] $”
Does not match regular expression	¬^$	String	Regular expression	X matches “^.*[a-zA-Z] $”
Substring	⊑	String	String	X is a substring of V
Not substring	⋢	String	String	X is not a substring of V
Substring	⊒	String	String	V is a substring of X
Not substring	⋣	String	String	V is not a substring of X
Subset	⊆	Array of any supported type	Same as input	X ⊆ V
Not subset	⊈	Array of any supported type	Same as input	X ⊈ V
Strict subset	⊂	Array of any supported type	Same as input	X ⊂ V
Not strict subset	⊄	Array of any supported type	Same as input	X ⊄ V

2.3 Storage format

All information regarding an expert system's decision table is stored in XML format. Appendix 1 contains XSD schema that is used for validating saved expert tables. All table files start with the root element *table* and include the definition of a namespace (“cocovila”) and define a schema against which table XML description is validated. Elements *input* and *output* define input and output variables of the table. Next, elements *hrules* and *vrules* consist of conditions and condition entries in the selection matrix for rows and columns of the table. Element *data* contains data table with values of the type described by the *output*.

A concrete example of an expert table's XML with input `x` of type `String` and output `z` of type

`int` is the following:

```
<?xml version='1.0' encoding='utf-8'?>
<tables xmlns="cocovila"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="cocovila file:./table.xsd"
        xmlns:cocovila="cocovila">
  <table id="tbl name">
    <input>
      <var id="x" type="String"/>
      ...
    </input>
    <output>
      <var id="z" type="int"/>
    </output>
    <hrules>
      <rule var="x" cond="eq" value="some value">
        <entry id="0"/>
      </rule>
      ...
    </hrules>
    <vrules>...</vrules>
    <data>
      <row id="0">
        <cell id="0">0</cell>
    ...
```

2.4 External interface

Expert tables implement `IStructuralExpertTable` Java interface with the following two methods:

```
public Object queryTable( Object[] args );
public String getTableId();
```

The class `TableManager` contains several static methods for accessing tables from outer Java packages:

```
static IStructuralExpertTable getTable( Package pack, String tableId );
static void updateTables( Package pack );
```

CoCoViLa API can be used for querying tables from synthesized programs. The class `ProgramContext` contains the following static method:

```
public static Object queryTable( String tableName, Object... args );
```

where `args` is a list on input values.

3 Graphical User Interface

This section describes the user interface of two essential visual components of the Expert System Shell: Expert Table Visual Editor and Expert System Consultant.

3.1 Expert Table Visual Editor

Visual Editor is a tool for creating new and editing existing expert tables. This section explains how to use main GUI components of it.

3.1.1 Opening Expert Table Visual Editor from CoCoViLa

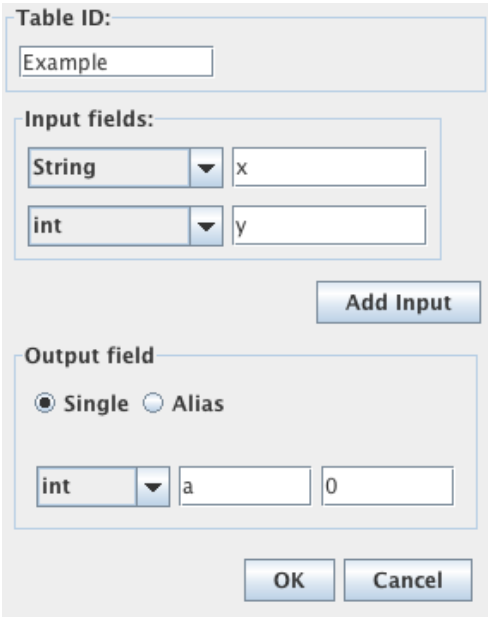
From the Scheme Editor click the menu item "Tools" -> "Expert Table Editor...".

3.1.2 Opening existing table

To open an existing expert table click “File” → “Open”. Tables are stored in XML format in files with .tbl extension.

3.1.3 Creating new Expert Table

After clicking "File" -> "New", the Table properties dialog will be displayed.



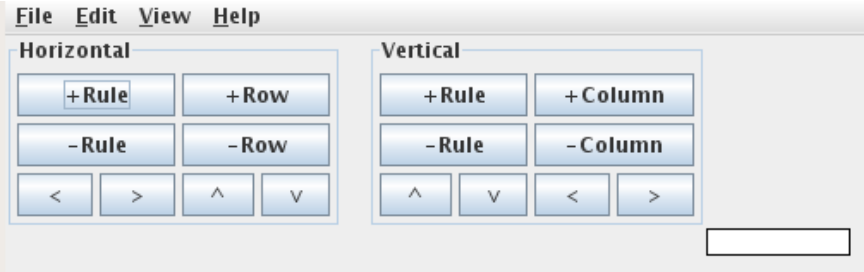
1. Enter Table ID.
2. At least one input variable has to be entered.
Use "Add Input" button to add more input fields.
Empty text fields will be ignored.
Duplicate variable names are disallowed.
3. Output can be either a single variable, or an alias consisting of one or more elements.
Variable names should not appear in the list of input variables.
4. Click "OK" and if there are no errors, new empty table will be created.

3.1.4 Saving a table

Click “File” → “Save” to save the table or “File” → “Save As” to select a file where a table should be saved. Multiple tables can be stored in one file.

If a table was not saved previously, file chooser dialog will be displayed allowing either to select a new or existing file.

If a table with the same Table ID already exists in the file, the confirmation dialog will be displayed asking to overwrite the table or cancel the action.



3.1.5 Table Editing

There are two sets of rules in the table. Horizontal rules are used to select a row and vertical rules for choosing a column from the data table.

Rules represent the conditions that have to be satisfied in order to use it. For example, if there is a rule “x < 5”, the rule can be used only if condition is true, i.e. value of x is less than five.

For each row/column in the table a subset of rules is checked in the selection matrix. This means that only checked rules will be considered for a given row/column and to choose a row/column, all conditions have to be satisfied for given inputs, otherwise the next row/column will be considered.

3.1.6 Changing table structure

After a table has been created, it is possible to change its properties.

From the Table Editor window, click “Edit” → “Table Properties...” to open the dialog box.

1. To change the Table ID, just edit it in the corresponding text field.

2. To delete an input, just delete its identifier in the corresponding text field.

3. To add more inputs use “Add Input” button.

4. To change the type of an existing input, use the combo box with types.

Note: all rules that correspond to the input whose type is to be changed will be deleted!

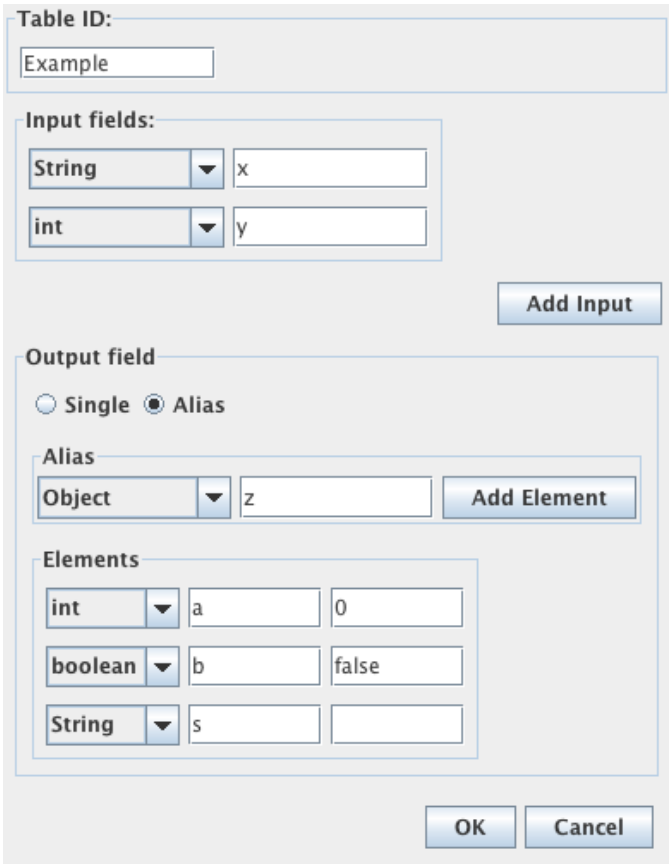
5. The type of an output can also be changed. Values in the data table will be erased only if the old type cannot be casted to the new one.

3.1.7 Using Alias as table output

In the case when a user needs to build a table with several outputs, alias output should be used. It is possible to do in the properties dialog by clicking the menu *Edit -> Table Properties*.

In the properties dialog's *Output field* section, *Alias* radiobutton should be clicked. First, the type and name of the alias should be specified. On the code level, aliases are represented as Java arrays, and if a users chooses one of the primitive types (`int, double, byte, boolean, etc.`),

then all elements of this alias will be of the selected primitive type. If the `Object` type is chosen, elements could have different types (both primitive and reference ones). In the example below, an alias with the name `z` is assigned as the table's output with three elements: `a` with type `int` and default value `0`, `b` with `boolean` type and default value `false` and `s` with type `String` having empty string as its default value.

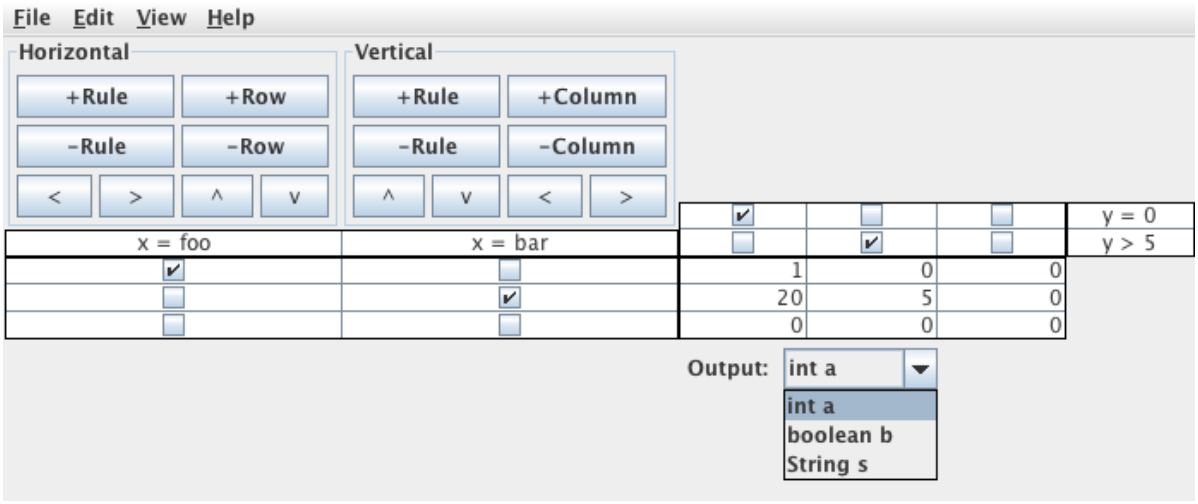


Table's output can be changed from single to alias and from alias to single using the following rules:

- Single → Alias: the type of a single output becomes the element type of an alias and the name of a single output becomes the first element's name of an alias.
- Alias → Single: the first element of an alias becomes the single output.

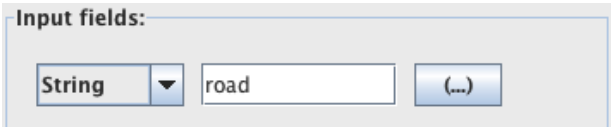
To delete an element of an alias output, the corresponding name should be erased using the *Table Properties* dialog.

The *Output* combobox enables one to switch between data tables of alias elements.

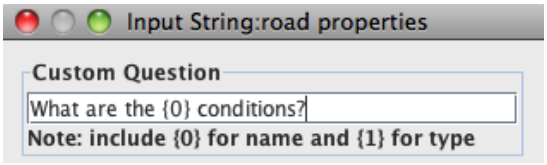


3.1.8 Setting table input's custom question

Expert System Consultant, described in section 3.2 asks user a question about particular input's value. The default question is "What is the value of a variable <*id*>?" Table properties windows enables to open additional settings for each table input and set a custom question.



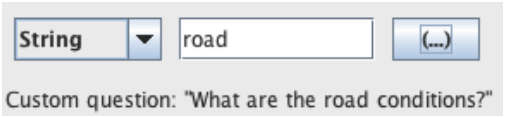
By pressing a "(...)" button, a new dialog appears.



User can type a custom question. Special text placeholders can be optionally used.

- Placeholder {0} is later substituted by the name of an input (e.g. "road")
- Placeholder {1} is substituted by the type of an input (e.g. "String")

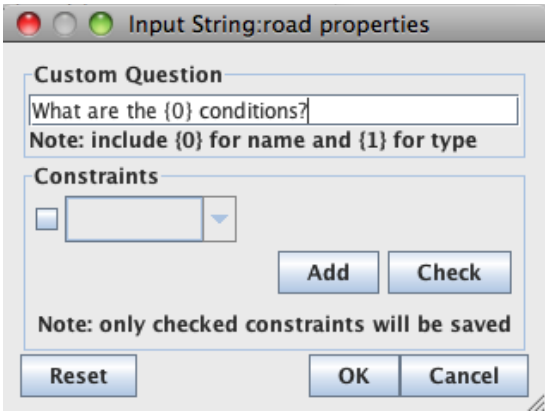
After the entry of a custom question and pressing "OK" button, the dialog closes and Table properties window displays the custom question.



3.1.9 Adding constraints to table inputs

Table input constraints enable one to specify a list of concrete values or a range of values for an input. Constraints also affect the editing and rendering functionality of inputs. For instance, if a list of values has been specified for an input, user will not be able to enter a random value in Rule dialog or Expert System Consultant, a combo-box with values from a given constraint will be

shown. For a range constraint, spinner or slider is displayed. To specify constraints, one needs to press the “(...)” button next to an input in the Table Properties window, a new dialog appears.

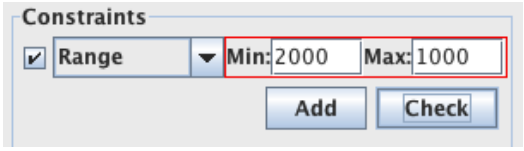


To start adding a constraint, first, a check-box has to be clicked. It activates a list of available constraints. Currently, two kinds of constraints are available:

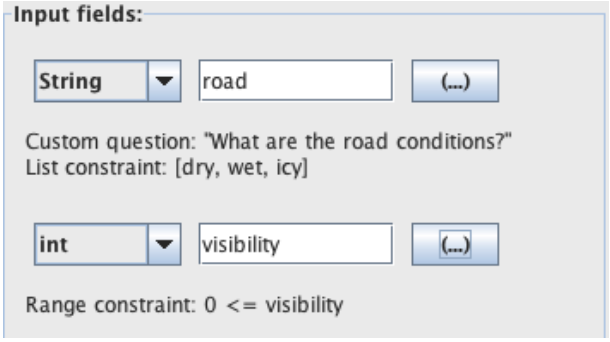
- List – Allows typing in a list of values separated by '%%' token.

- Range – Allows entering a *min* and *max* values which can also be empty.

Constraint parameters are verified for correctness by pressing 'Check' button and only valid ones can be saved, invalid constraints are highlighted with red:

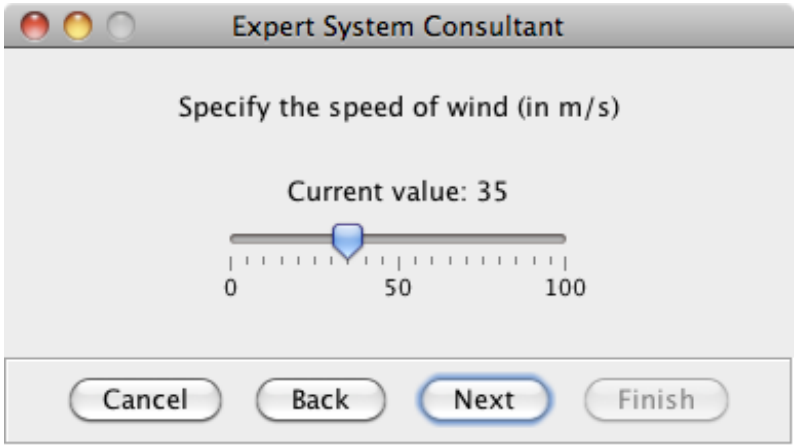


After pressing 'OK' button, Table Properties window will display valid constraints.



3.2 Expert System Consultant

This section describes the usage of interactive Expert System Consultant that performs consultations with users in the form of a dialog to get an answer from the knowledge base. The interface is automatically generated from an expert table used in the context. The consultant asks a question for each input (either the default one, i.e. "What is the value of a variable <*id*>?", or displays expert's predefined question). User can specify the value and press "Next" or, to change previously entered value, press "Back". To cancel consultation, "Cancel" button is used.



After providing answers to all questions, Expert System Consultant shows the summary and the outcome. Button "Finish" ends the successful consultation and, if required, returns a value (or a set of values) to the caller program.

The detailed explanation of Expert System Consultant will be given using the following example Expert System table:

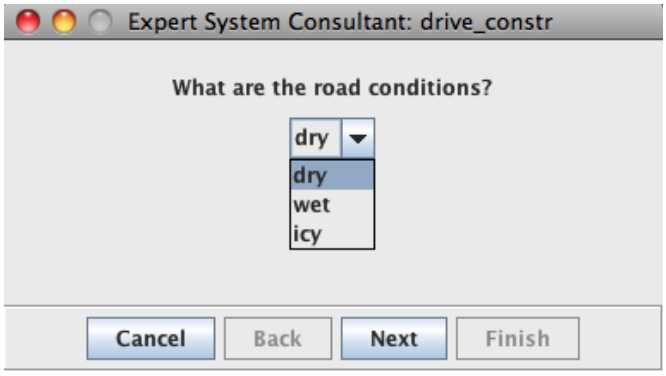
Horizontal: +Rule, +Row, -Rule, -Row, <, >, ^, v

Vertical: +Rule, +Column, -Rule, -Column, ^, v, <, >

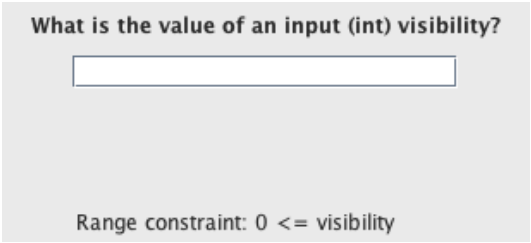
road = dry	road = wet	road = icy	visibility > 100	visibility ≤ 100	tech condition = good	tech condition = bad tires	experience = beginner
✔			✔		90		
✔				✔	80		
	✔		✔		90		
	✔			✔	70		
		✔				stay at home	
		✔			30		

The Consultant starts asking questions and automatically tries to minimize number of questions depending on already provided values of inputs.

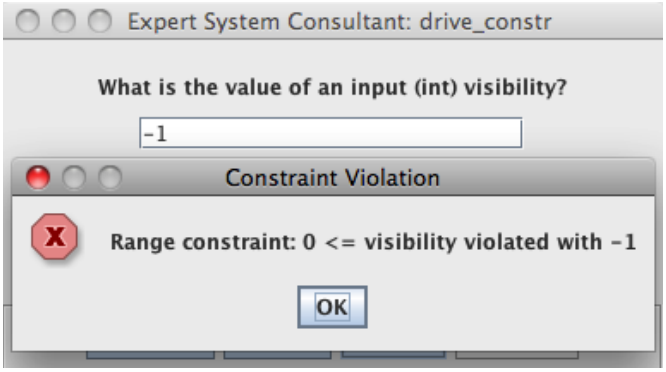
As a first question, obviously, it asks for the value of an input 'road'.



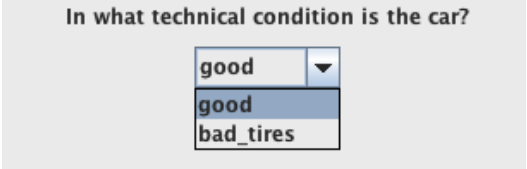
If the value “dry” is chosen, next table input that requires defining a value is 'visibility' (Note that a custom question was displayed for the input 'road', whereas for the 'visibility' a default one was used):



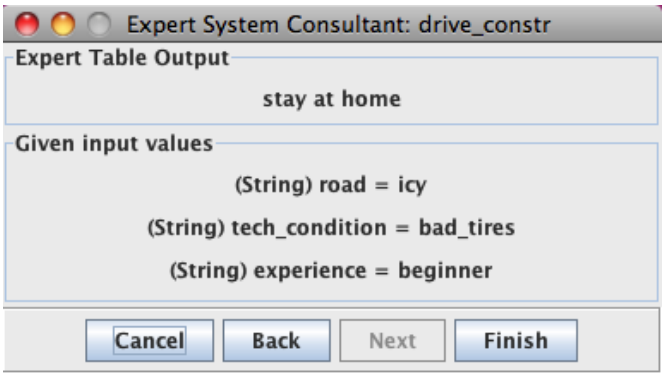
As a hint for a user, constraints are also printed in the question panel. Entered values are checked against constraints when user presses 'Next' button. If a constraint is violated, error message pops up:



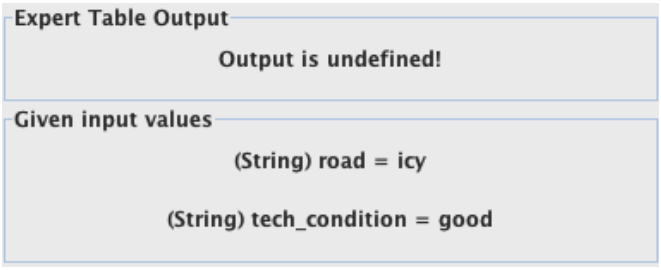
Going back one step and selecting “icy” as a value for 'road', there are no horizontal production rules left that require additional value entry. The next question is for the value of the input 'condition':



Choosing 'condition'=“bad_tires” and 'experience'=“beginner”, Consultant shows the output from the Expert System table:



Choosing 'condition'="good" leads to a problem, because the output value for this combination does not exist in the table. The Consultant windows displays a message "Output is undefined!"



The "Finish" button ends the conversation. The Consultant window closes and, optionally, an output value is returned to the caller (see Section 4.3).

4 Using Expert System Shell with CoCoViLa

4.1 Using tables in specifications

In CoCoViLa API the ProgramContext class contains a method for querying tables:

```
Object queryTable( String tableId, Object... args )
```

In the specification of a metaclass, the method queryTable() should be called from the method that implements an axiom used for getting a value from a table. The left hand side of such axiom should contain a table's id and a list of inputs of a table in the particular order, the right hand side is the corresponding output value of a table.
Example:
Let a table have the id="test" and two inputs of type `int`, output type is also `int`.

```
class TableTest {
    /*@ specification TableTest {
        int x, y, value;
        const String tableId = "test";
        tableId, x, y -> value{queryTable};
    }@*/

    int queryTable( String id, int x, int y ) {
        try {
```

```
            return
           (Integer)ee.ioc.cs.vsle.api.ProgramContext.queryTable( id, x, y );

        } catch(Exception e) {
           e.printStackTrace();
        }
        return -1;
    }
}
```

4.2 Using *@table* keyword

If a method implementing an axiom for querying a table does not perform any specific actions, it can be omitted and in the specification of an axiom @table keyword should be used instead of a method's name in the curly brackets:

```
class TableTest {
    /*@ specification TableTest {
        int x, y, value;
        const String tableId = "test";
        tableId, x, y -> value{@table};
    }@*/
}
```

If @table keyword is used, the first input of an axiom should be the table id followed by the list of table arguments. Names of variables (inputs of an axiom) do not have to match names of table inputs, only the order matters, it has to be precisely as defined in the table.
The generated code will be as follows:

```
public void compute( Object... cocovilaArgs ) {
    ...
    try {
        value = (java.lang.Integer)ProgramContext.queryTable(tableId, x, y);
    }
    catch( java.lang.Exception ex0 ) {
        ex0.printStackTrace();
        return;
    }
}
```

4.3 Providing partial list of table arguments

In CoCoViLa API the ProgramContext class contains another method for querying tables:

```
Object queryTable( String[] inputIds, String tableId, Object... args )
```

which allows to specify the mapping between input identifiers (names) and their corresponding values. In this case, the order of table arguments can be arbitrary, but (NB!) ids of specification variables must match ids of table arguments.
Another important aspect of using this methods is that it allows to provide partial (incomplete) list of table arguments. If some inputs are required but not given, Expert Consultant (see Section 3.2) will be displayed asking the values from the user.
The simplest way to use this feature is to specify the @tablewithinputmapping keyword as the realization of the axiom. The required code will be generated automatically.

Example:
This example is similar to the previous one, but it uses @tablewithinputmapping keyword and provides only the first argument (“x”). Expert Consultant will ask user to input the value of “y”.

```
class TableTest {
    /*@ specification TableTest {
        int x, y, value;
        const String tableId = "test";
        tableId, x -> value{@tablewithinputmapping};
    }@*/
}
```

The generated code will be as follows:

```
public void compute( Object... cocovilaArgs ) {
    ...
    try {
        value = (java.lang.Integer)ProgramContext.queryTable(
                  new String[] { “x” }, tableId, x );
    }
    catch( java.lang.Exception ex0 ) {
        ex0.printStackTrace();
        return;
    }
}
```

Appendix 1. Expert table XSD schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            xmlns="cocovila"
            targetNamespace="cocovila">

    <xsd:element name="tables" type="Tables" >
          <!--xsd:unique name="tableIdUnique">
            <xsd:selector xpath="cocovila:table" />
            <xsd:field xpath="@id" />
        </xsd:unique-->
    </xsd:element>
    <xsd:complexType name="Tables">
        <xsd:sequence>
            <xsd:element name="table" type="Table" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Table">
        <xsd:sequence>
            <xsd:element name="input" type="Input" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="output" type="Output" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="default" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="hrules" type="HRules" minOccurs="1" maxOccurs="unbounded"/>
            <xsd:element name="vrules" type="VRules" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="data" type="Data" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string" use="required" />
        <xsd:attribute name="description" type="xsd:string" />
    </xsd:complexType>

    <xsd:complexType name="Input">
        <xsd:sequence>
            <xsd:element name="var" type="InputVar" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Output">
        <xsd:sequence>
            <xsd:element name="var" type="OutputVar" minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="kind" use="optional" default="single">
          <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                              <xsd:pattern value="single|alias" />
                    </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="alias_id" type="xsd:string" use="optional" />
        <xsd:attribute name="alias_type" type="xsd:string" use="optional" />
    </xsd:complexType>

    <xsd:complexType name="Var">
        <xsd:attribute name="id" type="xsd:string" use="required" />
        <xsd:attribute name="type" type="xsd:string" use="required" />
    </xsd:complexType>

    <xsd:complexType name="InputVar">
        <xsd:complexContent>
          <xsd:extension base="Var">
            <xsd:all>
             <xsd:element name="question" type="xsd:string" minOccurs="0" />
             <xsd:element name="constraints" minOccurs="0" >
                    <xsd:complexType>
                     <xsd:all>
                      <xsd:element name="list" minOccurs="0" >
                              <xsd:complexType>
                                       <xsd:sequence>
                                        <xsd:element name="element" minOccurs="1" maxOccurs="unbounded" >
                                         <xsd:complexType>
                                                 <xsd:attribute name="value" type="xsd:string" use="required" />
                                         </xsd:complexType>
                                       </xsd:element>
                                       </xsd:sequence>
                              </xsd:complexType>
                      </xsd:element>
                      <xsd:element name="range" minOccurs="0" >
                        <xsd:complexType>
                             <xsd:attribute name="min" type="xsd:string" />
                             <xsd:attribute name="max" type="xsd:string" />
                        </xsd:complexType>
                      </xsd:element>
                     </xsd:all>
                    </xsd:complexType>
```

```
            </xsd:element>
          </xsd:all>
        </xsd:extension>
      </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="OutputVar">
        <xsd:complexContent>
                  <xsd:extension base="Var">
                  <xsd:attribute name="default" type="xsd:string" use="optional" />
                  </xsd:extension>
        </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="HRules">
      <xsd:sequence>
          <xsd:element name="rule" type="Rule" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="VRules">
      <xsd:sequence>
          <xsd:element name="rule" type="Rule" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Rule">
      <xsd:sequence>
          <xsd:element name="entry" minOccurs="0" maxOccurs="unbounded" >
                          <xsd:complexType>
                          <xsd:attribute name="id" type="xsd:int" use="required" />
                          </xsd:complexType>
          </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="var" type="xsd:string" use="required" />
      <xsd:attribute name="cond" type="xsd:string" use="required" />
      <xsd:attribute name="value" type="xsd:string" use="required" />
  </xsd:complexType>

  <xsd:complexType name="Data">
      <xsd:sequence>
          <xsd:element name="row" minOccurs="1" maxOccurs="unbounded" >
                          <xsd:complexType>
                          <xsd:sequence>
                           <xsd:element name="cell" type="Cell" minOccurs="1" maxOccurs="unbounded" />
                      </xsd:sequence>
                          <xsd:attribute name="id" type="xsd:int" use="required" />
                          </xsd:complexType>
          </xsd:element>
      </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Cell" mixed="true">
      <xsd:sequence>
        <xsd:element name="value" type="Value" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
                  <xsd:attribute name="id" type="xsd:int" use="required" />
  </xsd:complexType>

        <xsd:complexType name="Value">
                  <xsd:simpleContent>
                            <xsd:extension base="xsd:string">
                            <xsd:attribute name="var" type="xsd:string" use="required" />
                            </xsd:extension>
                  </xsd:simpleContent>
        </xsd:complexType>

</xsd:schema>
```