|  |  |
| --- | --- |
| [User Guide](http://www.opendial-toolkit.net/user-manual)‎ > ‎  Advanced modelling   |  | | --- | | **1. References**  Conditions may be more complex than simple checks between a state variable and a fixed value. For instance, one may want to specify that two variables A and B must have an identical value (whatever this value might be). This can be encoded via *variable references*inside the value attribute of an <if ..\.> markup. References to state variables are denoted with curly brackets { } to distinguish them from usual strings.  For instance, assuming two state variables firstvar and secondvar, the condition firstvar==secondvar` may be written as :  <if var="firstvar" relation="=" value="{secondvar}"/>  The condition will therefore be true if e.g. firstvar=foo and secondvar=foo. The curly brackets are important: without them, the condition states that the firstvar must have as value the string secondvar (instead of the value denoted by the variable).  More complex conditions can be constructed:  <if var="firstvar" relation="=" value="something({secondvar},{thirdvar})"/>  The condition will in this case be true if e.g. secondvar=foo, thirdvar=bar, and firstvar=something(foo,bar).  Variable references may similarly be used in effects:  <set var="firstvar" value="{secondvar}"/>  The value of firstvar will in this case be assigned to the value of secondvar.  Each variable referenced in such a manner is automatically added as an input node of the rule node constructed from the probabilistic rule.  **2. Quantification**  The expressive power  of probabilistic rules can be greatly improved by introducing universally quantified variables in the rule specification.[[1]](http://www.opendial-toolkit.net/user-manual/advanced-modelling-1#%5B1%5D) For instance, one may write a rule that determines how the graspability of an object *X* depends on its shape:  <rule id="r3">   <case>     <condition>       <if var="shape({X})" value="sphere"/>     </condition>     <effect prob="0.9">       <set var="graspable({X})" value="true"/>     </effect>     <effect prob="0.1">       <set var="graspable({X})" value="false"/>     </effect>   </case>   <case>     <condition>       <if var="shape({X})" value="cone"/>     </condition>     <effect prob="0.2">       <set var="graspable({X})" value="true"/>     </effect>     <effect prob="0.8">       <set var="graspable({X})" value="false"/>     </effect>   </case> </rule>  The rule will then apply for any arbitrary value *X* for which a state variable shape(*X*) exists in the dialogue state. In order to distinguish the free variable *X* from normal strings, free variables must be surrounded by curly brackets { }.  Free variables can also be employed in utility rules:  <rule id="r4">   <case>     <condition>       <if var="task" value="grasp({X})"/>       <if var="graspable({X})" value="true"/>     </condition>     <effect util="2">       <set var="a\_m" value="grasp({X})"/>     </effect>   </case>   <case>     <effect util="-2">       <set var="a\_m" value="grasp({X})"/>     </effect>   </case> </rule>  You may notice that the same notation (curly brackets) are employed for both references to state variables (cf. above) and free variables. OpenDial automatically decides whether the notation represents a reference or a quantified variable by assuming that a given {X}represents a reference if the variable X exists in the dialogue state, and will assume that X represents a free variable otherwise.  **3. Predictive variables**  Probability rules can be employed for two distinct purposes. The first purpose is to update the values of a specific state variable given some new information (for instance, inferring the user dialogue act *au* given the observed user utterance *uu*). But probability rules can also be used to provide a *prior* distribution for a future state variable that is expected to be observed in the next time steps. For instance, the last user intention *iu* may be used to predict the next dialogue act from the user. This prediction does not by itself create a new dialogue act (it's only a prediction), but it can be used as a useful prior for this observation.[[2]](http://www.opendial-toolkit.net/user-manual/advanced-modelling-1#%5B2%5D)  To distinguish these priors from actual observed values, OpenDial relies on the convention that predictive variables are denoted with a superscript ^p. A variable X^p represents therefore a prediction on the variable X to be observed in the future.  At runtime, OpenDial will then automatically connect together predicted and observed distributions and derive the resulting posterior distribution.  The following rule from the [step-by-step example](http://www.opendial-toolkit.net/user-manual/step-by-step-example) states for instance that, if the system asks the user to repeat, the user is expected to comply (and repeat the same dialogue act) with a probability 0.95.  <rule id="repeatpredict">   <case>     <condition>       <if var="a\_m" value="AskRepeat" />     </condition>     <effect prob="0.95">       <set var="a\_u^p" value="{a\_u}" />     </effect>   </case> </rule>  Note the reference to the existing dialogue act a\_u in the effect. The above rule allows the system to prime the probability of a repeated dialogue act compared to other hypotheses.  **4. String matching**  Most variables of a dialogue system have values encoded as strings. OpenDial includes special functions to easily manipulate strings in probabilistic rules. Rule conditions can notably applied to perform string matching, as in this example:  <rule id="r14">   <case>     <condition>       <if var="u\_u" value="take the {OBJ}" relation="contains"/>     </condition>     <effect>       <set var="a\_u" value="Request({OBJ})"/>     </effect>   </case> </rule>  The rule above indicates that if the variable u\_u contains a substring that matches the pattern take the {OBJ} (where {OBJ} is a placeholder), the variable a\_u will be set to Request({OBJ}). Note again the use of curly brackets to denote the placeholders.  If the actual content of the slot can be ignored, the placeholder can be replaced by a wildcard:  <rule>   <case>     <condition>       <if var="u\_u" value="turn \* left" relation="contains" />     </condition>     <effect>       <set var="a\_u" value="Request(Left)" />     </effect>   </case> </rule>  The above rule will be fired for a user utterance such as "turn left", "turn to the left", or even "now robot please turn to the left" (since the match can be partial).  OpenDial also support patterns with optional or alternative substrings. Optional substrings can be expressed via the (the\_substring)? construction, as in:  <if var="u\_u" value="turn (a bit)? to the left" relation="contains" />  The above condition will match both "turn to the left" and "turn a bit to the left".  Finally, alternative elements can be expressed via the construction (alternative\_1|alternative\_2|...), such as in the following condition:  <if var="u\_u" value="turn (a bit|a little bit)? to the left" relation="contains"/>  This last conditition matches the strings "turn to the left", "turn a bit to the left" and "turn a little bit to the left".  **5. Manipulating sets of elements**  In addition to strings, many state variables are also associated to values represented as *sets*of elements (for instance, the *n* most recent dialogue acts, or the set of objects perceived in the current scene). Probabilistic rules come with a built-in syntax to manipulate such sets.  *Specifying values as sets*  In order to specify that a particular value is a set, simply use a comma-separated list surrounded by square brackets, such as:  <variable id="some\_set">    <value prob="0.6">[value1, value2, value3]</value>    <value prob="0.2">[value2,value4]</value>    <value prob="0.2">[]</value>  </variable>  *Relations contains, !contains, in and !in*  Rule conditions can check for the presence/absence of particular items in a variable, as for instance:  <if var="some\_set" value="value1" relation="contains"/>  The above condition will simply check whether the value of the variable some\_set contains the element value1. To check whether an element is *not* part of a given set, the relation !contains(does not contain) can be used.  One can also check whether some individual element is contained in a set through the two relations in and !in, as for instance:  <if var="a\_u" value="[Request(Left),Request(Right)]" relation="in"/>  *Effects allowing multiple values*  By default, the output variable of a rule can only be associated to a single value. Consider for instance the two following rules:  <rule id="r1">   <case>     <effect prob="0.9">       <set var="A" value="val1"/>     </effect>   </case> </rule>  <rule id="r2">   <case>     <effect prob="0.9>       <set var="A" value="val2"/>     </effect>   </case> </rule>  The two rules r1 and r2 are in conflict, which means that the distribution for the output variable A will be P(A=val1)=0.495, P(A=val\_2)=0.495 and P(A=None)=0.01. Sometimes, one might want to lift the constraint of a single value per variable. For instance, the last dialogue act of the user might contain several basic elements which can be combined with one another. In such a case, the last dialogue act can be represented as a set of elements.  To specify that the values of the output variable are not mutually exclusive , one simply needs to insert the attribute exclusive="false". For instance, one can rewrite the two rules r1and r2 as such:  <rule id="r1">   <case>     <effect prob="0.9">       <set var="A" value="val1" exclusive="false"/>     </effect>   </case> </rule>  <rule id="r2">   <case>     <effect prob="0.9">       <set var="A" value="val2" exclusive="false"/>     </effect>   </case> </rule>  The distribution of the output variable A will in this case be P(A=[val1,val2])=0.81, P(A=[val1])=0.09, P(A=[val2])=0.09 and P(A=None)=0.01**.**Note that such effects with multiple values are only available for probability rules (not utility rules).  *Adding elements to an existing list*  If you have a variable some\_set whose values are defined as lists, and you want a rule to add a new element val to it, you can define the updated variable as the union of the existing list and the new element:  <set var="some\_set" value="{some\_set}+{val}" />  **6. Nested conditions**  We explained in the section on [dialogue domains](http://www.opendial-toolkit.net/user-manual/dialogue-domains) that rule conditions are essentially defined as conjunctions or disjunctions of basic conditions. If needed, one can also define more advanced, nested conditions, such as for instance A=a1 **and** **not**(B=b1 **or** C!=c1):  <condition>   <if var="A" relation="=" value="a1"/>   <not>     <or>       <if var="B" relation="=" value="b1"/>       <if var="C" relation="!=" value="c1"/>     </or>   </not> </condition>  Three XML markups are available for such nested conditions: or, and and not, respectively denoting disjunctions, conjunctions and negations.  **7. Probability and utility functions**  The probability or utility of a given effect are typically encoded by a fixed value (as in <effect prob="0.8">) or a parameter (as in <effect util="theta1">).  However, it is also possible to define more complex values that are defined through mathematical functions.  As an example, imagine you want to write a rule that finds potential matches for a referring expression containing size properties ("big", "small") based on a set of perceived visual objects.  Of course, you want the probability of an object matching an expression containing the word "big" to be proportional with the perceived (numerical) size of the object, while the matching probability for an expression containing the word "small" should be inversely proportional to the object size.  One way to encode this is to express the probability as a mathematical function:  <rule>     <case>         <condition>             <if var="size({Object})" value="{Size}" />                                 <if var="refer\_expression" relation="contains" value="small"/>         </condition>         <effect prob="exp(-{Size})">             <set var="refer\_matches value="{Object}" exclusive="false"/>         </effect>     </case>     <case>         <condition>             <if var="size({Object})" value="{Size}" />                                 <if var="refer\_expression" relation="contains" value="big"/>         </condition>         <effect prob="exp(-1/{Size})">             <set var="refer\_matches value="{Object}" exclusive="false"/>         </effect>     </case> </rule>  As we can see, the probability of the first effect is inversely proportional to the size of the object (i.e. if the size is close to 0, the value of the function will be close to 1). The opposite is true for the second effect. These mathematical functions can be arbitrarily complex and can make use of the usual arithmetic operators (+,\*,-,/) and the following predefined functions: [abs, acos, asin, atan, cbrt, ceil, cos, cosh, exp, floor, log, log10,  log2, sin, sinh, sqrt, tan, tanh].  **8. Custom functions**  What if we want to use a particular function in the rule that is not in the set of predefined mathematical functions listed above? For instance, in natural language understanding tasks, one might want to define some "similarity" measure between two semantic representations, but there's no easy way to encode such measure in the rules directly.  The last version of OpenDial allows you to define your own functions in Java and use them in the probabilistic rules. First, in the XML domain file, you should declare your function in the following way:  <function name="your\_function\_name">class.path.to.your.function</function>  You can then use this custom function anywhere in the probabilistic rule. For instance, assuming you have created a function similarity that takes two arguments and returns a similarity measure between the two, you can write:  <rule>     <case>         <effect prob="similarity({u\_u},(move|go) \* forward)">             <set var="a\_u" value="Move(Forward)"/>         </effect>         <effect prob="similarity({u\_u},(go|turn) \* left)">             <set var="a\_u" value="Move(Left)"/>         </effect>     </case> </rule>  Finally, and most importantly, you need to actually write the Java class class.path.to.your.function for your custom function. This class **must** implement the interface java.util.function<List<String>,Value>. The core of the class if the method  apply(List<String> args), which takes the two arguments of the function, and returns its result (in this case, the result must be a double value, since it represents a probability). Here is a very simple example of code for the similarity function:  package opendial;  import java.util.List; import java.util.function.Function;  import opendial.bn.values.Value; import opendial.bn.values.ValueFactory; import opendial.templates.Template;  public class Similarity implements Function<List<String>,Value> {      public Value apply(List<String> args) {         String utterance = args.get(0);         Template t = Template.create(args.get(1));         if (t.partialmatch(utterance).isMatching()) {             return ValueFactory.create(1.0);         }         return ValueFactory.create(0.0);         } }  **9. Relational structures**  Until now, the state values we used were constrained to simple entities (strings, numbers, etc.). In other words, the underlying state representation remains essentially propositional (slot-value pairs). In many cases one would like to operate with more advanced, relational structures. Think for instance of the semantic content of an utterance, complex user intentions or hierarchical tasks (See [D. Ramachandran and A. Ratnaparkhi. "Belief Tracking with Stacked Relational Trees" (SIGDIAL 2015)](https://aclweb.org/anthology/W/W15/W15-4609.pdf) for a discussion).   The last version of OpenDial allows you to create such relational structures and use probabilistic rules to "manipulate" them. The syntax for the relational structures is essentially the same as in the Stanford Core NLP package. To indicate a relation labelled Rbetween two elements A and B, simply write it as A R>B, and use brackets to delimitate subgraphs. Here is a simple example: [look subject>hearer object>[cylinder attribute>red location>right]]. The above example creates a relational structure with 5 nodes and 4 relations.   You can also associate attributes to each node, e.g. to provide some additional tags on the elements in the graph. This is done by adding a | bar followed by a key:value pair right after the node content. For instance, loves|pos:VB indicates that the pos attribute for the node has the value VB.  You can then create rules that exploit such relational structure. For instance, one can create conditions that check for the occurrence for specific relational patterns, such as [look subject>{X}] (any subgraph with a node look followed by a subject relation). And you can create effects that manipulate graphs, in order for instance to merge or trunk some parts of the relational structure.  [1] See [Lison (2014)](http://folk.uio.no/plison/pdfs/thesis/thesis-plison2014.pdf), p. 67-68 and 74-76 for the theoretical foundations of such quantification mechanisms.  [2] See [Lison (2014)](http://folk.uio.no/plison/pdfs/thesis/thesis-plison2014.pdf), p. 78-79 for details. | |