Constraints for Active Templates

Jim Blythe
USC Information Sciences Institute

December 19th, 2001

This document specifies the language used to communicate information about constraints in tools built under the Active Templates program. Tools refer to constraints in order to apply them to particular values, to advertise their input and output specifications to other tools, and to define how they are computed. The language specified here allows tools to communicate with each other about constraints and also to store them in a central location in a tool-independent way.

We use a general definition for "constraints". Constraints are represented functionally and can be applied to a number of arguments to give a return value. Depending on how this value is interpreted, constraints can be used for a number of purposes, including the following:

- Computing the true or default value of an element in a template given values for other elements,
- Computing a range, or set of allowable values for an element in a template given values for other elements,
- Deciding whether a particular combination of values is allowable or in violation of the constraint,
- Deciding whether one value is preferred over another under the terms of the constraint.

This representation for constraints subsumes those typically used in constraint satisfaction systems (see for example [Jeavons, Cohen, & Gyssens1999]), which usually represent constraints over a set of variables as tuples of allowable values for those variables.

In order to reason about constraints, some tools may also make use of further information such as a measure of the constraint's relative or absolute importance, ranges of values that are allowed, marginal or disallowed, and recommendations for recovering from constraint violations. This document provides a way to represent this information, however, there is no requirement that other tools make use of this information in order to comply with the constraint specification language.

Several tools being developed within the Active Templates program already make use of explicit representations of constraints, and these have influenced the development of this representation language. For example, the Heracles project uses constraints to describe the data that is gathered from web wrappers, and constraints on combinations of values in fields. Its constraint representation is compatible with the one described here. CODA uses constraints to describe when a change in part of a plan should be communicated to a separate planning group. In COMIREM, constraints are used for reasoning about resource allocation in plans. Tools for active forms, including those from SoftPro, BBN and Alphatech, can attach constraints to elements in forms to provide information about legal values and to compute values from other elements. This document includes a section on how to specify the way that a constraint, which is defined independently of particular forms, can be attached to the elements in an active form.

The next sections give a tutorial overview of representing constraints and sending requests to tools to evaluate constraints. Section 7 contains a short, precise specification of the constraint language. The final section explains how the Constable constraint engine interprets constraints represented in this language. It is intended for groups who use Constable to compute constraints, while the rest of the document is independent of Constable.

Constraints are represented in XML using the `<constraint>` tag. A request to apply a constraint is sent between tools with the `<apply>` tag. In the remaining subsections we discuss how to represent a number of features of constraints based on the different constraint-based tasks that tools perform. Section 1 gives examples of representing simple constraints. Section 2 shows how to attach constraints to elements of active forms. Section 3 shows how to express the input/output profile of constraints that tools can use to decide how to process them or attach them to

template elements. Section 4 describes how to pass a request between tools to apply a constraint to specific input. Section 5 shows how to represent the definition of a constraint, specifying how the constraint works as well as its input/output specification. Section 6 describes place holders for further information such as a constraint's source, its importance or degree of violation. Fields in this section are not provided in detail and are intended to be further specified in the course of discussion among the participants.

1 Representing simple constraints

We begin with some short examples of constraints to give a flavor of the language. Details about the constructs used here can be found in the following sections.

Suppose that a template contains a field called "water temperature" containing a number. A request to apply a constraint that water temperature (a number) must be > 60 looks like this:

```
<apply>
  <name> greater-than </name>
  <arg><field>water temperature</field></arg>
  <arg>60</arg>
</apply>
```

More details are given below on how the tag `<field>` attaches the constraint to the active form from which the constraint is called, and how the `<name>` tag along with the arguments designates a particular constraint to be applied to the values.

The following constraint checks that the airline is in the set { american, us-airways }, showing how set-based constraints are expressed.

```
<apply>
  <name>one-of</name>
  <arg><field>airline</field></arg>
  <arg><set><element>american</element>
            <element>us-airways</element>
       </set>
  </arg>
</apply>
```

The above constraints take a set of arguments and return a boolean value, but this is not necessary for constraints. The following constraint, shown in a declarative form rather than as an `<apply>` request, takes as input a destination airport and returns a set of preferred airlines.

```
<constraint>
  <capability>
    <predicate>recommended-airlines</predicate>
    <arg><type>airport</type></arg>
  </capability>
  <result-type><set>airline</set></result-type>
</constraint>
```

The `<capability>` tag, described in more detail in Section 3, specifies the constraint's predicate name and expected input types. The `<result-type>` tag describes the kind of objects that the constraint will return. Here `airport` and `airline` are types, which must be agreed on by the tools sharing information about this constraint.

The following example shows how a constraint can be used to describe a preference between two alternatives. This would be used, for example, in a field where the airports are ordered by distance from the target location. The constraint takes two airports and returns the one that is preferred. The criterion is not shown here, but it can be expressed using the `<method>` tag as shown below.

```
<constraint>
  <capability>
    <predicate>prefer</predicate>
    <arg><type>airport</type></arg>
    <arg><type>airport</type></arg>
  </capability>
  <result-type>airport</result-type>
</constraint>
```

2 Attaching constraints to active forms

All of the constraints described in the previous section are intended to be applied to the fields of templates. The primary way that tools attach constraints to fields is through the `<field>` tag, which can be used either in the arguments when a constraint is applied, or in the method definition as described in Section 5. The text inside the tag designates a field in the template whose value will be used for the argument where the tag appears. The intention is for the constraint engine to query the template tool for the value of this field, although the form of this query is beyond the scope of this document.

Figure 1 shows a Constable template for SEAL beach infiltration. The field labeled "Lunar Illumination" has a value of 0.815 and is colored red because it violates the constraint that the illumination (really the proportion of the moon that is visible) be not greater than 0.3. This can be expressed with the following constraint attached to the field:

```
<apply>
  <name> less-than-or-equal-to </name>
  <arg><field>Lunar Illumination</field></arg>
  <arg>0.3</arg>
</apply>
```

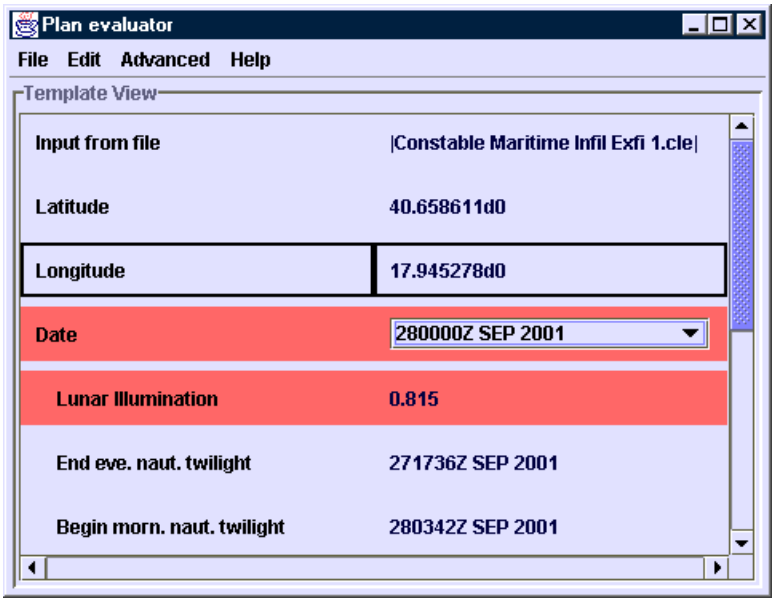

Figure 1: A Constable template for beach infiltration

Tools using constraints need not use the `<field>` tag if they use another means to attach constraints to data. In any call using an `<apply>` tag, the values in the arguments could be provided explicitly by the tool making the call. For example, if the first constraint from the previous section is attached to a field called "water temperature" in some template, the tool calling the constraint might look up the value directly and formulate this request:

```
<apply>
  <name> greater-than </name>
  <arg>55</arg>
  <arg>60</arg>
</apply>
```

The `<field>` tag is a useful way to record the attachment but it is not required.

When constraints are used in various ways in tools, a number of facts about the constraints are left to the tool to represent. As well as how to fill the arguments of the constraint, the tool must decide how to use the return value. For example, a boolean value might be used to decide if a violation has occurred, or a set or objects might be used to define the range of possible values. The language defined here does not currently capture this aspect of constraints since it is determined by the tool, but this is a point where further information could be captured if it would benefit the developers.

3 Advertising constraint capabilities

As the above examples show, constraints can be used in a number of ways by different tools. The tools themselves are likely to handle the details of attaching constraints in various ways to the information that the tools manipulate or display, but guidelines on valid ways to use the constraint should be represented in the constraints themselves. This is done through the constraints *capability*, which specifies the input types the constraint takes and usually some indication of what the constraint does, and the *result-type* which specifies the output type of the constraint.

Suppose a constraint takes two numbers as input and checks whether the first is bigger than the second. We want to advertise the capability of the constraint, which includes the types of its arguments and its return value, and a predicate name, *e.g.* "greater-than". The following XML specification does this:

```
<constraint>
  <name> greater-than </name>
  <text>Text written here is not processed.</text>
  <capability>
    <predicate> greater-than </predicate>
    <arg><type>number</type></arg>
    <arg><type>number</type></arg>
  </capability>
  <result>
    <type>boolean</type>
  </result>
</constraint>
```

The information in the `<capability>` tag specifies the calls that can be made to the constraint, using a predicate name and typed arguments. The arguments can also be named as is shown below. Any symbol can go in the `<type>` tag, including `number`, `boolean`, `aircraft`, or `fuel-capacity`. It is intended that the communicating tools agree on the use of these symbols as types. For example, if `aircraft` and `boat` have been defined as types with the usual subtypes, a tool cannot use `MK-V` to fill an argument whose type is `aircraft`.

Arguments may also be given names, which can be used when the constraint is called as shown in the next section. It is also good practice to put information about what the constraint is doing into the capability to help with grouping and searching for constraints. This often entails reifying part of the task specification so that it appears in the parameters:

```
<constraint>
  <name>check-landing-distance-for-aircraft-1</name>
  <capability>
    <predicate>check</predicate>
      <arg>
        <name>obj</name>
```

```
        required-landing-distanced
      </arg>
      <arg>
        <name>of</name>
        <type>aircraft</type>
      <arg>
      <arg>
         <name>against</name>
         <type>number</type>
         <text>Available landing distance when the constraint is called
         </text>
       </arg>
  </capability>
  <result>
    <type>boolean</type>
  </result>
</constraint>
```

Here, `required-landing-distance` is the value of the first argument, since it is not enclosed in `<name>` or `<type>` tags. Since this is a constraint specification, the first argument is therefore defined as having a constant value.

4 Applying constraints

Although constraint specifications can be verbose, this is unlikely to create a bandwidth problem because systems will spend more time using constraints than they will spend on checking their specification. A request to apply a constraint is specified with the `<apply>` tag specifying the constraint name and its arguments. For example:

```
<apply>
  <name>check-landing-distance-for-aircraft-1</name>
  <arg>required-landing-distance</arg>
  <arg>C130</arg>
  <arg>750</arg>
</apply>
```

Other examples were shown in Section 1.

You can also specify argument names within `<apply>` tags. You do not need to specify the `<name>` tag for the constraint if all the names of the arguments are supplied as well as the `<predicate>`. For example, this is equivalent to the call above:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>required-landing-distance</arg>
  <arg><name>of</name>C130</arg>
  <arg><name>against</name>750</arg>
</apply>
```

In order to uniquely determine the constraint, though, the call should either use the constraint name or its predicate along with all argument names.

5 Defining constraints

One reason to to define constraints centrally is to allow reuse of constraints across several systems, avoiding redundant work in defining them. For instance, for a tool to check the runway length constraint for an aircraft at an airport using

the constraint defined in the last section, it would have to retrieve all the runway lengths for that airport from the NIMA database, and test them until one was found that was long enough. A new constraint can be defined that checks the aircraft against an airport and hides this detail from the system calling the constraint. It's specification might be as follows:

```
<constraint>
  <name>check-landing-distance-for-aircraft-2</name>
  <capability>
    <predicate>check</predicate>
      <arg>
        <name>obj</name>
        required-landing-distance
      </arg>
      <arg>
        <name>of</name>
        <type>aircraft</type>
      <arg>
      <arg>
         <name>against</name>
         <type>airport</type>
       </arg>
  </capability>
  <result>
    <type>boolean</type>
  </result>
</constraint>
```

The definition of this constraint can be given explicitly using a `<method>` tag. Calls to other methods, either defined or in an initial library, can be made within this tag. The calls need to refer to the input arguments, and for this reason the `<var>` tag is used to name variables:

```
<constraint>
  <name>check-landing-distance-for-aircraft-2</name>
  <capability>
    <predicate>check</predicate>
    <arg>
      <name>obj</name>
      required-landing-distance
    </arg>
    <arg>
      <name>of</name>
      <type>aircraft</type>
      <var>?aircraft</var>
    </arg>
    <arg>
      <name>against</name>
      <type>airport</type>
      <var>?airport</var>
    </arg>
  </capability>
  <result>
    <type>boolean</type>
  </result>
  <method>
```

```
    <apply>
      <predicate>check</predicate>
      <arg><name>obj</name>required-landing-distance</arg>
      <arg><name>of</name>?aircraft</arg>
      <arg><name>against</name>
        <apply>
          <predicate>find-maximum</predicate>
          <arg><name>of</name>
           <retrieve>landing-distances<arg>?airport</arg></retrieve>
          </arg>
        </apply>
      </arg>
    </apply>
  </method>
</constraint>
```

This method definition assumes that the constraint `find-maximum` takes a set of numbers as input and finds the maximum number, and that a field called `landing-distances` can be retrieved from an object of type `airport`, yielding a set of numbers (one for each runway). The `<retrieve>` tag indicates that the tool implementing the constraint should get this information from a data base. At present, this data base is available to the tool in a way that is not specified. In the longer term, the syntax being developed to express retrievals from the central AcT data server will be used here. `<retrieve>` and `<apply>` tags can be nested arbitrarily, either in the `<method>` tag here or in `<apply>` messages at the top level.

For more examples of constraints defined in this language, see the definition of the Metoc constraints from USSOCOM manual M525-6 as used in Constable.

6 Further information about constraints

In order to reason about constraints, some tools may also make use of further information such as

- a measure of the constraint's relative or absolute importance,
- ranges of values that are allowed, marginal or disallowed, and
- recommendations for recovering from constraint violations.

This section describes how to represent this information. However there is no requirement that other tools make use of this information in order to comply with the constraint specification language. Each of the constraint information fields described in this section is optional. The form of the information contained within the following tags is not defined at present, but is intended to be defined after debate about this RFC.

6.1 The source of a constraint

Constraints can come from several different sources and this is often important to a planner who must decide between alternative plans that might violate different constraints. For example, constraints might come from manuals such as the SOF manual of Metoc constraints M525-6, from operating manuals for platforms or equipment such as an MH47. Other constraints might be from the personal library of the planner or group and represent the group's personal experience.

The source of a constraint is indicated though the `<source>` tag. Currently, the string contained within the tag is not processed, but is stored and can be shown to the user on request.

6.2 Constraint importance

While some constraints may render the mission inoperable if they are violated, others may simply represent preferences. Information about the constraint's impact on the mission can be included through the `<impact>` tag. There is

no restriction placed on the kind of information that is stored in this tag, and its interpretation is left to the tools that work with the constraints. The tag represents static information about the constraint, however, *i.e.* it is not re-computed for each set of input variables given to the constraint.

6.3 Degree of violation

Many constraints are threshold tests, for example a maximum wind speed or minimum water temperature. For these and others it is often important to distinguish between different degrees of violation of the constraint. For example, users might want to be notified if the water temperature is below 60 degrees, but see a stronger caution if it is below 55 degrees. This can be expressed using the `<degree>` tag. Potential fillers of the tag include `note`, `caution` and `warning`, or `green`, `yellow` and `red`.

7 Language summary

The `<constraint>` tag represents information about a constraint. This can include a specification of its input and output types, and optionally a definition in terms of other constraints.

```
<constraint>
  <name>SYMBOL</name>
  <capability>SEE BELOW</capability>
  <result>TYPE SYMBOL</result>
  <method>SEE BELOW</method> (optional)
</constraint>
```

The `<apply>` tag represents a request to apply a constraint with a particular set of data. As described in Section 4, information within the tag should include either the constraint name and a list of argument values, or the constraint predicate and a list of values for named arguments.

```
<apply>
  <name>SYMBOL</name> (optional)
  <predicate>SYMBOL</predicate> (optional if name supplied)
  <arg>VALUE</arg>*
</apply>
```

Within `<capability>` tags:

```
<capability>
  <predicate>SYMBOL</predicate>
  <arg><name>SYMBOL</name><type>SYMBOL</type><var>SYMBOL</var></arg>*
</capability>
```

`<var>` is optional and allows the constraint definition to refer to the argument. The constraint definition is represented in the `<method>` tag.

Within `<method>` tags, the `<apply>` tag can be used just as in a top-level message.

`<retrieve>SYMBOL<arg></arg>*</retrieve>`.

The `<retrieve>` tag specifies information that can be retrieved from the information servers that the system applying the constraint has access to, as described in Section 5.

The `<apply>` or `<retrieve>` tags can be nested in the argument specifications.

8 Using Constable as a constraint server

Constable accepts the XML forms defined above as messages, and will reply with XML messages as defined here.

When Constable receives a message using the `<constraint>` tag, it defines the constraint if the `<method>` tag is supplied and otherwise searches its internal knowledge base for a constraint that matches the specification. If the new

constraint is successfully defined, Constable replies with the message `<defined>SYMBOL</defined>`, where `SYMBOL` is the name of the constraint. If there was an error in the definition, Constable replies with a message of the form `<error><name>SYMBOL</name>...</error>` including information about the error. If no constraint definition was given using a `<method>` tag, but the constraint was already defined in Constable, it replies with `<found>SYMBOL</found>`, and if the constraint is not already defined it replies with the `<error>` tag.

When Constable receives messages using the `<apply>` tag, it searches for a constraint matching the call, either using the name of the constraint or the predicate and the names and types of the arguments. If a match is found, and the constraint can be applied to the data without error, Constable replies with a message of the form `<value>VALUE</value>` giving the value of the constraint. If the constraint cannot be found or there is some error in its application, Constable replies with the `<error>` tag providing information about the error.

In Constable, constraint parameters always have names, although they do not have to be used when the constraint is applied. Types as decribed in Section 3 are implemented using Loom, which is a description logic. The only requirement on a system using Constable is that it is consistent in using type members.

References

[Jeavons, Cohen, & Gyssens1999] Jeavons, P.; Cohen, D.; and Gyssens, M. 1999. How to determine the expressive power of constraints. *Constraints* 4:113–131.

A Suggestions for commonly occurring constraint types

The constraint specification in this document has concentrated on the language for describing constraints rather than on specific constraint definitions. This appendix includes suggested forms for some commonly occurring types, including arithmetic, set-based and temporal constraints.

Arithmetic constraints

Many tools will include constraints based on thresholds or make use of simple arithmetic computations for constraints. Here we suggest constraints or tags for these:

Logical tests

Logical tests understood in constraints include `and`, `or` and `not`. Here is how `not` can be used. `and` and `or` are similar but can take an arbitrary number of arguments.

```
<method>
  <not><apply>(constraint returning a boolean)</apply></not>
</method>
```

Threshold tests

Numeric thresholds $>, <, >=, <=$, and $=$ are represented as constraints, using full names rather than symbols. Use negation from logical tests above rather than $\neq$.

```
<constraint>
  <name> greater-than </name>
  <capability>
    <predicate> greater-than </predicate>
    <arg><type>number</type></arg>
    <arg><type>number</type></arg>
  </capability>
  <result>
    <type>boolean</type>
```

```
  </result>
</constraint>
```

The other predicates are `less-than`, `greater-than-or-equal-to`, `less-than-or-equal-to` and `equals`.

Arithmetic functions

The simple arithmetic constraints $+$, $-$, $*$ and $/$ are represented as constraints with names `add`, `subtract`, `multiply` and `divide`, each taking two arguments.

Set-based constraints

Set-based constraints $\in$ and $\subset$ are represented with constraints named `one-of` and `subset`.

Temporal constraints

Using times as in Larry's RFC; before, after, and supporting function for the time between two time points

B Example Metoc constraints supported in Constable

The following are constraints from USSOCOM manual M525-6 that are supported in Constable. They are included here to provide further examples of the constraint interface language. Anyone wishing to use Constable to try the constraint interface can get a copy of the program by sending email to the author at blythe@isi.edu.

Check platform speed against a distance and time:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>platform-speed</arg>
  <arg><name>of</name>crrc<text>any mode of transport</text></arg>
  <arg><name>against</name>20<text>distance in miles</text></arg>
  <arg><name>starting</name>XX<text>timepoint</text></arg>
  <arg><name>ending</name>XX<text>timepoint</text></arg>
</apply>

<apply>
  <predicate>estimate</predicate>
  <arg><name>obj</name>speed</arg>
  <arg><name>of</name>crrc</arg>
</apply>
```

Check seaworthiness against a sea state:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>seaworthiness</arg>
  <arg><name>of</name>crrc<text>Any boat</text></arg>
  <arg><name>against</name>4<text>Sea state</text></arg>
</apply>
```

The following message to Constable will result in the set of recognised boat names:

```
<instances>boat</instances>
```

Check max wave height against a wave height:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-wave-height</arg>
  <arg><name>of</name>crrc<text>Any boat</text></arg>
  <arg><name>against</name>5<text>wave height</text></arg>
</apply>
```

Check max operating temperature against a temperature:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-temperature</arg>
  <arg><name>of</name>crrc<text>Any equipment</text></arg>
  <arg><name>against</name>85<text>temperature in fahrenheit</text></arg>
</apply>
```

Min operating temperature:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>min-temperature</arg>
  <arg><name>of</name>crrc<text>Any equipment</text></arg>
  <arg><name>against</name>50<text>temperature in fahrenheit</text></arg>
</apply>
```

Max wind speed:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-wind-speed</arg>
  <arg><name>of</name>crrc<text>Any boat</text></arg>
  <arg><name>against</name>35<text>wind speed in knots</text></arg>
</apply>
```

Max current:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-current</arg>
  <arg><name>of</name>sdv<text>Any boat</text></arg>
  <arg><name>against</name>2<text>current in knots</text></arg>
</apply>
```

Aircraft constraints

The following message to Constable will result in a set of recognised aircraft names:

```
<instances>aircraft</instances>
```

Min ceiling:

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>min-ceiling</arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>1200<text>ceiling in meters</text></arg>
</apply>
```

Min visibility

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>min-visibility</arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>1200<text>ceiling in meters</text></arg>
</apply>
```

Max crosswind

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-crosswind</arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>30<text>crosswind in knots</text></arg>
</apply>
```

Max turbulence

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-turbulence</arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>1<text>turbulence</text></arg>
</apply>
```

Icing

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>max-icing</arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>1<text>icing</text></arg>
</apply>
```

Thunderstorm avoidance

```
<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>thunderstorm-avoidance-below-230/arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>12<text>distance in miles</text></arg>
</apply>

<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>thunderstorm-avoidance-above-230/arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>12<text>distance in miles</text></arg>
</apply>

<apply>
  <predicate>check</predicate>
  <arg><name>obj</name>thunderstorm-avoidance-low-level/arg>
  <arg><name>of</name>C130<text>Any aircraft</text></arg>
  <arg><name>against</name>12<text>distance in miles</text></arg>
</apply>
```