The SmartFrog Constraint Extensions

For SmartFrog Version 3.12

Localized for UK English / A4 Paper

Table of Contents

1. Introduction

This document covers the general model for how constraints may be used within the SmartFrog language. The purpose of adding constraints to the SmartFrog language is to allow descriptions given in the SmartFrog language to be partially specified, and for the constraints to be used to complete them. Indeed, the constraints are ultimately intended to act as both “verifier” and “completer” of descriptions. Thus attributes can be left unbound, and left to be bound by the constraint solver. Alternatively, they can be bound to specific values and the constraints used as predicates to validate the bindings.

To specify a constraint in a Smartfrog description, use the built-in description type: Constraint. Constraints are function types and are evaluated as part of link resolution, as all function types are.

A Constraint will declare what may be considered to be local variables which will be filled in as the Constraint is evaluated. These are declared using the VAR keyword.

The general form of a Constraint type is as follows.

```
foo extends Constraint {
  bar1 VAR;
  ...
  barn VAR;

  bas1 reference;
  ...
  basn reference;

  bat1 value;
  ...
  batn value;

  [sfConstraint] pred1 constraint string;
  ...
  [sfConstraint] predn constraint string;
}
```

The order of attributes within the description is unimportant. In the foregoing:

- bar1 through barn are the local variables filled in by constraint evaluation. All variables must eventually be filled in, although they do not necessarily have to filled in during the evaluation of the particular constraint type that they are attached to. An example of this is given in 2.2.

- bas1 through basn are link references that are resolved prior to evaluating the constraint. *The use of link references facilitates a means of controlling the order in which constraints within a sfConfig description are evaluated.*

  That is to say, the evaluation of a constraint will be deferred until link references contained therein are first resolved, and if any of these references themselves have parts which resolve to constraint types, then these constraints will be evaluated first.

- bat1 through batn are Smartfrog values, either components or simple values.

- pred1 through predn are uninterpreted constraint strings that are passed to a constraint solving engine for evaluation, according to which the VAR attributes will be filled in.

Constraint strings are determined by means of the tag [sfConstraint].

Constraint strings within a constraint type are ordered according to the lexical ordering of their attribute names. This means that if a constraint type is extended, the order of constraint strings in the extended type can be controlled by appropriate use of attribute names. For example, defining a constraint string with attribute name p1a in an extension of a type with constraint strings p1, p2, p3, will mean that the constraint strings are evaluated in the order p1, p1a, p2, p3.

The version of the constraint system is for experimentation – there is no specific constraint language. Instead, the constraints are given as strings within Constraint types which are interpreted by a constraint solver, implemented as a plug-in.

There is a default plug-in which uses the Eclipse constraint solver: http://eclipse.crosscoreop.com/index.html. This treats the constraint strings as prolog clauses to be used in order to fill in the VAR attributes, as well as validate the values of the other attributes with pre-specified values, either by reference or value.

This document does not go into the details of implementing plug-ins, but in essence a plug-in must extend the abstract type:

```
org.smartfrog.sfcore.languages.csf.constraints.CoreSolver
```

It is made known to the Smartfrog parser by means of setting the java property:

```
org.smartfrog.sfcore.languages.csf.constraints.SolverClassName
```

This should be set to the name of the plug-in class. This can either be done on the Java command line using the -D option or, perhaps more easily, it can be set in the default.ini file used by the default SmartFrog command-line scripts to set system properties (for details of this file, see the SmartFrog user manual).

If this property is not set, constraint types are simply not evaluated. This is not an error, so long as no VARs are specified.

The constraint solving Smartfrog logic is currently supported within a branch, and is not part of the main Smartfrog release. It is at:

https://smartfrog.svn.sourceforge.net/svnroot/smartfrog/branches/core-branch-constraints

2. Writing Smartfrog Descriptions with Constraints

For now, authors of Smartfrog descriptions which make use of constraints need to understand how to write Eclipse programs. This will not always be the case. We are moving towards an abstract constraint language which will have its own semantics. It will use Eclipse for evaluation, but will not require understanding of Eclipse.

We do not seek to present the Eclipse language here. Instead interested readers should consult:

Constraint Logic Programming using Eclipse, Krzysztof R. Apt, Mark Wallace, 978-0521866286.

2.1 Simple Example 1

```
#include
"/org/smartfrog/sfcore/languages/sf/constraints/constraints.sf"


sfConfig extends {
  foo1 extends Constraint {
     x foo2:x;
     y VAR;
     z (x + 1);
     [sfConstraint] p1 "y ranges [1..3]";
     [sfConstraint] p2 "y is x*z";
  }
  foo2 extends Constraint {
     x VAR;
     [sfConstraint] -- "(x=2; x=1)";
  }
}
```

As can be seen, we need to include the file constraints.sf which contains some predefined type definitions.

foo1 is a constraint type which contains:

- an attribute x, which is a reference to the value of the attribute of the same name in foo2
- an attribute y, which is declared a local variable (VAR)
- an attribute z, which is a sum expression in regular Smartfrog syntax

x and z are references and must be resolved prior to evaluating the foo1 constraint.

As the foo2 part of the foo2:x reference is resolved to another constraint, this constraint must be evaluated first. Evaluating foo2 means passing the anonymous constraint string "(x=2; x=1)" to the Eclipse plug-in.

In constraint strings, attributes are referred to simply by using their name. **Note,** for now, attribute names within Constraint types may **not** start in upper-case. Also, attribute names should not clash with any Eclipse predicates used with the constraint strings of a type, as these predicate names will be obscured by the attribute names. Built-in keywords defined for constraint strings, such as ranges in the example, will obscure attribute names; thus, authors should also avoid using these.

In this constraint string, x is first bound to 2. If further constraint solving within the sfConfig hierarchy fails, backtracking may occur to the extent that this

assignment is undone. This will be the case in this example, as we shall see. In this eventuality, another possibility is provided; that is to assign x to 1.

The result of processing foo2 initially is to set x to 2. In foo1, x is thus resolved to 2. Further, z is resolved to 3. In constraint string p1, we restrict the range of attribute y to [1..3]. The string uses the keyword ranges. This is not yet available. This document will report soon on its support. The syntax in the meantime is the Eclipse syntax: y #:: [1..3].

p2 assigns the value of y to be x*z. As the result of this expression is 6, which is outside this range, link resolution is backtracked. This includes the assignment of z to 3, and the assignment of x to 2. The single constraint string in foo2 makes a dispensation for x to be assigned to 1 as an alternative. This means that z in foo1 is assigned to 2, and y assigned to 2 by the evaluation of constraint string p2.

The final sfConfig after link resolution is as follows, as dumped by sfParse is given.

```
extends {
  foo1 extends DATA {
    x 1;
    y 2;
    z 2;
  }
  foo2 extends DATA {
    x 1;
  }
}
```

2.2 Another Example

```
#include
"/org/smartfrog/sfcore/languages/sf/constraints/constraints.sf"

ListElement extends Constraint {
   list theList;
   [sfConstraint] -- "member(el, list)";
}

sfConfig extends {

   theList genList:theList;

   x extends ListElement { el elements:x; }
   y extends ListElement { el elements:y; }
   z extends ListElement { el elements:z; }

   elements extends Constraint {
      x VAR;
      y VAR;
      z VAR;
      list theList;
      [sfConstraint] -- "[x,y,z] ranges list, alldifferent([x,y,z])";
   }

   genList extends Constraint {
      theList ["one", VAR, "three", VAR];
      aList [VAR, "two", VAR, "four"];

      [sfConstraint] -- "theList = aList";
   }
}
```

After type resolution (which happens prior to link resolution), the sfConfig definition is as follows (dropping sfFunctionClass attributes):

```
sfConfig extends {
   theList genList:theList;

   x extends {
      el elements:x;
      list theList;
      [sfConstraint] -- "member(el, list)";
   }
   y extends {
      el elements:y;
      list theList;
      [sfConstraint] -- "member(el, list)";
   }
   z extends {
      el elements:z;
      list theList;
      [sfConstraint] -- "member(el, list)";
   }

   elements extends {
      x VAR;
      y VAR;
      z VAR;
      list theList;
      [sfConstraint] -- "[x,y,z] ranges list, alldifferent([x,y,z])";
   }


   genList extends {
      theList ["one", VAR, "three", VAR];
      aList [VAR, "two", VAR, "four"];

      [sfConstraint] -- "theList = aList";
   }
}
```

Component descriptions in link resolution are evaluated top-down, depth-first. As such, theList is first resolved. It is a reference which in resolution causes the genList constraint type to be evaluated. The effect of evaluating this constraint type is to assign [“one”, “two”, “three”, “four”] to “theList”.

Inspecting x, we see that it depends on the consraint type elements; as a result elements is evaluated first.

The constraint string says that the ranges of x,y and z are the “theList”, and that the values of x, y and z must be different. This means from the point of view of sfConfig:x / y / z, the respective values of el must be different. This constraint type is an example of a type, for which, when evaluation has finished, there remain unbound VARs. This is acceptable (and often useful) as long as these VARs get bound at some stage, by means of evaluating some other constraint type.

Such binding occurs, when we come to evaluate the constraint string for type x. We pick a member of “theList” for attribute el. Carrying this out will not only bind el in x, it will also bind x in elements. Evaluation of y and z occur similarly. A dump of sfParse on the given source description is now given.

```
extends {
  x extends DATA {
    list [|"one", "two", "three", "four"|];
    el "one";
  }
  y extends DATA {
    list [|"one", "two", "three", "four"|];
    el "two";
  }
  z extends DATA {
    list [|"one", "two", "three", "four"|];
    el "three";
  }
  elements extends DATA {
    x "one";
    y "two";
```

```
    z "three";
    list [|"one", "two", "three", "four"|];
  }
  theList [|"one", "two", "three", "four"|];
  genList extends DATA {
    theList [|"one", "two", "three", "four"|];
    aList [|"one", "two", "three", "four"|];
  }
}
```