The SmartFrog Constraint Extensions

For SmartFrog Version 3.06

Localized for UK English / A4 Paper

Table of Contents

INTRODUCTION 3
CONSTRAINTS IN SMARTFROG 4
1 Variables: *4*
2 Constraints: *5*
PLUG-INS 7
THE PROLOG ABSTRACT PLUG-IN 8

Introduction

This document describes some extensions provided to allow experimentation with constraint resolution technologies in conjunction with the SmartFrog language. These extensions are not intended to provide the definitive "SmartFrog Constraint System", but are designed to allow a wide variety of different technologies to be used for this purpose. Consequently, the extensions themselves provide very limited syntactic or semantic support, and mostly the support is in the form of free textual annotations that are then interpreted by an appropriate plug-in for the specific constraint solver in use.

The document covers the general model for how constraints are envisaged being used, how plug-ins may be written and loaded, and it covers the first such plug-in – the Prolog plug-in – and how this should be used. It should be noted, however, that the Prolog plug-in is abstract in that it does not actually contain a Prolog engine, but is capable of easily being specialised to use such an engine.

The extensions appear as a "new" language – the csf language, files of which are denoted by the extension .csf – which provides the extensions. If these extensions are not used, the language is completely equivalent to the core sf language.

Constraints in SmartFrog

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 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.

The core to the extensions is the introduction of a new experimental language, the csf language. This is a true extension of the existing sf language and if the new features are not used, the csf language has the same semantics as the sf language. Eventually, as the constraint system settles down, it might be envisioned that it replaces the existing function and assertion aspects of the language – replacing it with a uniform constraint model. This is for the future.

The csf language introduces two new concepts: the variable and the constraint.

1 Variables:

A variable is introduced by a new keyword “VAR” which denotes an unbound attribute value. This is much as TBD does in the sf language, and indeed one possibility was to supersede the previous use of TBD, however it was decided to make the new language a proper extension of the old.

Syntactically a VAR may be used anywhere a basic value (e.g. an Integer) may be used – for example as the value of an attribute or in a list. Note, however, that as functions are evaluated before constraints they should be avoided in such expressions unless the function can cope with a VAR parameter. Note that the Vector function is one that can cope with a VAR parameter, so VAR may be used within a list.

The semantics of VAR need some explaining.

1. Every syntactic occurance of the keyword VAR introduces a new variable – i.e. it may be bound to a different value from other variables.

2. Copying a Component Description containing a VAR as the value of an attribute, creates a new variable.

3. An attribute that links to another which is defined as a VAR, shares that VAR. Consequently, the constraints that apply to both of these attributes constraint the value that may be bound to that variable. These attributes both take the same value when the variable is bound.

Each variable is given an index – a unique integer. Whenever a variable is printed, it is shown as “VAR/n” where the “n” indicates this index. Using this mechanism, the sharing that exists between the attributes can be seen.

In fact, the parser will also accept the syntax with the index, but its use is not recommended – it is simply there for completeness.

Bearing this in mind, consider the following example:

```
Example extends Prim {
    a VAR;
    b VAR;
}
```

```
Example2 extends Example {
    b a;
}

sfConfig extends {
    x extends Example;
    y extends Example { a x:a; };
    z extends Examlpe2;
}
```

Running this through the sfParser will result in the following (equivalent up to the actual indices that are generated):

```
extends {
   x extends {
       a VAR/0;
       b VAR/1;
   }
   y extends {
       a VAR/0;
       b VAR/2;
   }
   z extends {
       a VAR/3;
       b VAR/3;
   }
}
```

The sharing and copying should be clear from the indices that are printed.

Note that any VARs that are left unbound by the constraint resolution are considered erroneous and errors are reported.

2 Constraints:

These are the second of the syntactic features added to the csf language.

This version of the constraint system is for experimentation – there is no specific constraint language. Instead, the constraints are given as textual annotations which are interpreted by one or more plug-in constraint solvers. There is no expectation that every plug-in will be able to interpret any textual annotation, but each will have its own “internal” syntax for these annotations.

For example, the Prolog plug-in treats the annotations as prolog clauses to be used in a search for variable bindings.

Constraints may be added to component descriptions, using a form of the SmartFrog long-string syntax – the delimiters of which are

```
#suchThat#...#
```

where the “...” indicates the text that will be passed to the plug-in.

Each component description may have a list of these provided, and the intended interpretation is that of the conjunction of all the constraints in the list – however, in reality the plug-ins can interpret lists of constraints in any way that they desire.

An additional feature is that constraints are propagated through extension. Consequently, descriptions can be created that are “typed” with constraints relating its attributes and which, when extended can also add additional constraints.

Consider the example (using some abstract constraint language)

```
AllDiffernt extends {
    a VAR;
    b VAR;
    c VAR;
} #suchThat#
```

```
     a != b and a != c and b != c
  #

Ordered extends AllDifferent #suchThat#
   a <= b and b <= c
#
```

Ordered has both constraints, and thus will ensure that the three attributes are strictly ordered.

Beyond this abstract description, all the details of the semantics of a constraint resolution depends upon the plug-in that is used.

Plug-ins

The way in which constraints are solved to bind variables is defined by a plug-in. These plug-ins are given the description, with its variables and constraint strings, and are then expected to replace the variables within the description with values that satisfy the constraints, replorting failures to find suitable solutions or if the solutions leave some variables unbound.

The reason for this architecture is that there are a large range of constraint systems, each with different semamtics and search strategies for finding solutions. Furthermore, some of the best solvers are commercial products and there is no way that the SmartFrog system can be made to depend on one of these.

As each of these solvers can deal with a different range of constraint types, so it is not even possible to provide a single fixed constraint syntax without in some sense limiting this syntax to the least common subset – hence the decision to make the constraints uninterpreted strings so far as the SmartFrog system is concerned.

This doecument does not go into the details of implementing plug-ins, but in essense a plug-in must implement the interface:

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

During execution of the parser, an additional phase is included in the standard set of phases (between function and assertion) which invokes the plug-in. The plug-in is selected by 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, this can be set in the default.ini file used by the default SmartFrog command-line scritps to set system properties (for details of this file, see the SmartFrog user manual).

If this property is not set, the default solver is used and this has the following semantics:

1. No constraints are solved, any annotations are ignored

2. The description is checked to see if there are any unbound variables and if so, reports an error.

Consequently, if the constraint and variable syntax is not used, and if the default plug-in is used, the net semantcs is that of the basic sf language.

Currently there is a single plug-in implemented, with others on the way. This plug-in uses a Prolog language engine. This is described abstractly below.

The Prolog Abstract Plug-in

The Prolog abstract plug-in is a parent class for any prolog plug-in – it carries out many of the tasks requried to interpet the strings and variables as Prolog entities and to create an overall Prolog query to be solved. It becomes easy, therefore, to integrate a range of different Prolog engines such as JLog (a portable Java GPL prolog), SWIProlog (an fast and extensive LGPL C++ implementation) or perhaps Sicstus Prolog (a commercial product).

Currently only the first of these is implemented as an actual plug-in. Because the JLog code is GPL, it cannot be distributed as part of the SmartFrog core (indeed it is not part of the core, merely a plug-in). The plug-in code (also GPL as it depends on JLog) and a redistribution of Jlog, is available from the SmartFrog web-site (www.smartfrog.org). Thanks are due to the JLog team for approving of this integration and redistribution. The web page for this open source project is http://www.sourceforge.net/projects/jlogic.

1 The Prolog theory

Prolog comes with a number of built-in rules, the specific set will depend on the implementation used. However, it is possible that for a specific purpose other rules will need to be provided, for example defining rules about a specific deployment context. In order to support this, the Prolog plug-in supports the concept of a theory file – a file that is read (consulted, in Prolog-speak) before the constraints are solved.

This file is defined by the system property

```
opt.smartfrog.sfcore.languages.csf.constraints.prologTheoryFile
```

This can be set on the command-line with the Java -D option, or in the default.ini file used by the default SmartFrog scripts. If the property is not set, the default theory file

```
/org/smartfrog/sfcore/languages/csf/constraints/prologTheory.prolog
```

is read. This file contains a few trivial examples of the definition of rules. This file will be extended over time (suggested rule definitions for inclusion are welcome), and so should be consulted to see what is provided.

2 Writing constraints

Constraints are simply Prolog queries, and are written as such and are only parsed when given to the Prolog processor – neither SmartFrog nor the plug-tries to parse this text. However, the plug-in does pre-process the constraints in the following way:

1. It identifies references to the Component Description to which the constraint is attached. These references are surrounded by the '@' character (if the character '@' is required, this should be written '@@').

2. Each reference is dereferenced in the context of the Component Description and there is one of three possibilities:

   - the reference is to a basic value of a type that can be represented in Prolog – the value is unparsed in the place of the reference using Prolog syntax;

   - the reference is to a value that cannot be represented within Prolog – this is considered an error;

   - the reference is to a VAR, which is unparsed as a Prolog unbound variable.

3. The constraints are concatenated using the "," operator (conjunction) so that bindings from one constraint flow over to the next, and so that back-tracking traverses back-and-forth over the constraints.

4. If a solution is found, the variable bindings are extracted from the query result and the whole descritpion is traversed and the variables replaced with their value. If any variables remain unbound, this is considered an error.

The values that are mapped into Prolog are all the numeric types (though possibly only as Integer and Float, or even less resolution), strings (represented as Atoms, not character lists), booleans, the SFNull value and lists (vectors). Component Descriptions are not currently represented as Prolog values and so may not appear in queries.

Note that as the constraints are concatenated with no additional annotation by the conjunction operator, overall parsing should be considered (e.g. bracketting of operators, the occurance of the clause terminator '.', etc). These may produce some unexpected and hard to understand errors!

3 Constraint ordering

The plug-in makes no effort to intelligiently order the constraint query clauses. This means that at times the search may be very inefficient, or may not find a solution where one exists. Later releases of the plug-in may make a better effort to find a solution by ordering clauses according to some measure – such as the number of unbound variables, or perhaps by providing an annotation to hint to the plug-in as to whether the constraint is should be early or late in the overall search. Again, suggestions are appreciated...

4 Examples

Here are a number of small examples to illustrate the use of the plug-in. They assume the definition of some list rules in the Prolog theory file.

```
Ordered extends {
    a VAR;
    b VAR;
    c VAR;
} #suchThat# @a@ < @b@, @b@ < @c@#

sfConfig extends Ordered #suchThat#
       subset([@a@, @b@, @c@], [1,7,3,9,5])
#;
```

The above example will bind the attributes a, b and c to an ordered set of integers selected from the list of integers [1,7,3,9,5].

```
ListElement extends {
    element VAR;
    list VAR;
} #suchThat# member(@element@, @list@) #

System extends {
    theList VAR;
    x extends ListElement {list theList;}
    y extends ListElement {list theList;}
    z extends ListElement {list theList;}
} #suchThat# allDifferent([@x:element@, @y:element@, @z:element@]) #

sfConfig extends System {
    theList ["one", VAR, "three", VAR];
    aList [VAR, "two", "three", "four"];
} #suchThat# @aList@ = @theList@ #
```

The above example defines the elements of x, y and z to be different elements of the list which results from the unification of the two lists defined in sfConfig.