The SmartFrog Constraint Extensions

For SmartFrog Version 3.12

Localised for UK English / A4 Paper

1 Introduction 4
2 Foundations of SmartFrog's Constraint Support 7
2.1 Prolog – Programming in Logic 7
2.2 Constraint Solving with Eclipse 9
3 Link Resolution When Using Constraints 11
4 Basic Syntax Overview 13
4.1 Control flow operators 13
4.1.1 ‘,’ ‘;’ 13
4.1.2 ‘->’ ‘-->’ 13
4.2 Assignment Operators 14
4.2.1 ‘=’ 14
4.2.2 ‘is’ 15
4.2.3 member 15
4.2.4 allocate 15
4.2.5 source, add_path 16
4.3 Passive test operators 16
4.3.1 ‘==’ ‘\==’ 17
4.3.2 ‘<’ ‘>’ ‘>=’ ‘=<’ 17
4.4 Active constraint operators 17
4.4.1 ‘subtype’ 17
4.4.2 alldifferent 18
4.4.3 ‘eq’, ‘neq’, ‘lt’, ‘lte’, ‘gt’, ‘gte’, ‘nt’ 19
4.4.4 ‘and’, ‘or’, ‘implies’ 19
5 Further Aspects of SmartFrog’s constraint support 20
5.1 VARs 20
5.1.1 List Example 22
5.1.2 Subtyping examples 23
5.2 Automatic Variables 25
5.3 User Variables 26
5.4 Array Types 29
5.5 Aggregator Types 31
5.5.1 forall/exists 32
5.5.2 nsum/nproduct 34
5.5.3 alldifferent 34
5.5.4 Further Examples 34
5.6 AggregatedConstraint Types 34

5.6.1 Alternative Characterisation of Resource Allocation Example 39
5.6.2 Alternative Characterisation of List Example 39
6 Frequently Asked Questions 42
6.1 Why no support for reals? 42
6.2 What about optimisation problems?.....42
6.3 What happens if Eclipse is not installed, or constraint support not enabled? 42
6.4 What about SmartFrog's support for Assertions and Schemas?.....42
7 References.....43

1 Introduction

This document describes 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 and/or verify them.

To specify a constraint in a SmartFrog description, we 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 range default;
   ...
   barn VAR range default;

   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 Section 5.1.1.

  When declaring a VAR attribute, it is usual practice to specify at least a range. The author may also like to specify a default value. An example might be: foo VAR "[1..10]" 10. This says that foo is a local (integer) variable with range [1..10], and whose value defaults to 10 if it is left unassigned after otherwise evaluating the Constraint that the VAR is attached to.

- 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 may 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.

As a simple example, consider the following SmartFrog description.

```
#include "/org/smartfrog/functions.sf"


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

To make use of the constraint extensions we need to include the file functions.sf which contains definitions of some function types.

foo 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 to be a local variable (VAR) with an integer range [1..3]
- 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 foo constraint.

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

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.

In the given 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 foo, x is thus resolved to 2. Further, z is resolved to 3. The constraint string in foo assigns the value of y to be x*z. As the result of this expression is 6, which is outside the range declared for y, 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 foo 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.

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

2 Foundations of SmartFrog's Constraint Support

The constraint solving support within SmartFrog is provided through the use of the *Eclipse* constraint solving engine (not to be confused with the Java IDE of the same name). The Eclipse constraint engine had been previously developed by Imperial College, London, UK, and has since been developed by Cisco Systems. More information can be found on Eclipse at: http://eclipse.crosscoreop.com/ and in [1].

The subset of the SmartFrog language specific to constraint support provides abstractions from the language of Eclipse. That is to say, it is an *abstract syntax* which is somewhat detached from the Eclipse language.

However, on occasion, it may be necessary to revert to specifying certain aspects of a particular domain being modelled using the Eclipse language directly.

Before presenting the abstract constraint language in full, it is appropriate to provide an overview of certain aspects of the native Eclipse language. This is particularly desirable as the computational model is shared between the abstract and Eclipse languages, as well as certain aspects of syntax.

2.1 Prolog – Programming in Logic

Prolog (*Prog*ramming in *Log*ic) [2] is a special case of *logic programming*, which is the use of logic as a programming language combined with deduction, specifically resolution, as a computational mechanism.

A Prolog program may be viewed from both declarative and procedural perspectives. For this document, it suffices to restrict discussion to a procedural perspective (which we unapologetically abuse from a purist's standpoint to make the discussion fit with our assumed audience of practitioners more accustomed to imperative programming languages, such as Java).

The operator ':-' is used to define a procedure, i.e.,

```
procedure head :- procedure body.
```

In Prolog, procedure names are symbols which begin with lower-case letters. For instance,

```
foo :- bar, bar2.
```

This procedurally may be read as: *one way* to perform foo is to perform bar followed by bar2. In this procedural view, calls made to procedures implicitly evaluate (or yield, or return) true or false values.

We may provide multiple definitions of a procedure, whose bodies constitute alternative ways in which the head may be realised. For example, foo may also be defined according to the following:

```
foo :- baz, baz2.
```

The body of a Prolog definition may be empty. Procedurally, this means that the procedure does no work and trivially evaluates to true. An example is:

```
foo2.
```

This says that foo2 necessarily evaluates to true. It is a fact of the domain.

A goal in Prolog is a conjunction of procedure calls which each have to evaluate to true (in the order specified in the goal), for the goal itself to evaluate to true, or succeed. An example is the following which procedurally says perform foo, followed by foo2.

```
:- foo, foo2
```

In calling a procedure, we replace the call in the goal with the body (or right-hand side) of the procedure definition. For instance, if we start with the single *goal* :- foo, then in calling foo, the goal may evolve to :- bar, bar2, according to the first definition of foo, or to :- baz, baz2, according to the second definition. Say the first is chosen to begin with but the evaluation of bar fails. In this case, the current goal will be unwound to the last *choice-point*, where another selection is made. It is not hard to see from this that the evaluation of a Prolog goal yields a computation tree. In this case, the other available selection is to replace foo with baz, baz2.

The computation of a goal is successful iff the empty goal, written :- true, is yielded through the act of successively refining the goal through replacing calls in the goal with the bodies of procedure definitions.

Say we have the following set of procedure definitions:

```
foo :- bar, bar2.
foo :- baz, baz2.
bar :- bat, bat2.
bar2.
bat2.
baz.
baz2.
```

In Prolog, evaluation of the goal :- foo would proceed thus:

```
:- foo.
:- bar, bar2.
:- bat, bat2, bar2.  No resolution on bat can be performed, backtrack
to last choice-point, which was for foo...
:- baz, baz2.
:- baz2.
:- true.
```

As can be seen, the goal successfully evaluates to :- true.

Procedures may be parameterised, with the arguments of a procedure being terms, which are constants, variables, or compound terms. An example:

```
foo(X) :- foo2(Y,X).
```

Here, X and Y are variables which are both universally quantified at the scope of the procedure definition. As Y occurs in the body only, it may be alternatively viewed as being existentially quantified in the body. Variables in goals are also existentially quantified.

Terms in Prolog are may be viewed as data over which computations are made. A constant in Prolog is a *number*, such as 1, 2, 3 etc, or an *atom*, such as fred, tom, or 'hello'. An atom is any sequence of characters starting with a lower-case letter, or ‘. A compound term is a combination of a *functor* (which syntactically is the same as an atom) with a number of terms

as arguments. An example is the combination of a list '.' functor with a set of terms to make a list of these terms, such as .(1, .(2, 3)), which may be written as [1,2,3] in sugared Prolog syntax. A variable starts with an upper-case letter, or underscore, to indicate as much. Examples are Var, Val, _23, _ (where _ is the anonymous variable, for which multiple occurrences do not constitute the same variable, unlike all other variables).

A call within a Prolog goal may be replaced by the body of a definition iff the head of the definition matches the call according to a process called *unification*. A call unifies with a procedure head iff their names are syntactically identical, they have the same number of parameters, and all parameters themselves term-unify.

Two terms unify (i.e. term-unify) iff:

- One or both are variables, or
- They are both constants, and are syntactically identical, or
- They are both compound terms, with syntactically identical functors, and the same number of arguments, such that their respective arguments also term-unify.

Variables in Prolog are logical variables, meaning that their values may only be progressively more specialised to some term value, through the process of unification. That is to say, they are not mutable variables as are the mainstay in imperative programming. For instance, the variable X might be instantiated to a list of two variables [A,B], say, as X and [A,B] term-unify. Subsequently, we may further refine this term to [1,2], through term-unification. However, [A,B] may not be refined to [1], say, as [A,B] and [1] do not term-unify.

2.2 Constraint Solving with Eclipse

Eclipse is a Prolog-based constraint programming language. Constraint solving in Eclipse fundamentally comes down to evaluating/executing Prolog goals, where certain calls made in the process of such evaluation may have side-effects on both a conceived *constraint store* and the *ranges* of logical variables used within the Prolog goal.

Fundamentally, there are two types of call that may be made from within an Eclipse goal. The first is to *constrain* the range of a logical variable, the second is to work towards instantiating a logical variable to a particular value. The first are *constraint calls*, the second *search calls*. Constraint calls cause the ranges of variables, and other constraints, to be stored within the constraint store. Whenever a logical variable is assigned/instantiated to a value, *constraint propagation* within the constraint store occurs, in that the ranges of all logical variables that are remaining to be instantiated are modified appropriately.

For example, say we have three logical variables X, Y and Z. Let's say that we assign each of these variables the range [1,2,3]. Specifying the range constraint would be a constraint call. (We do not elaborate how we would do this in Eclipse here – see pertinent documentation, because in SmartFrog we use an abstraction for range specification). Let us imagine another constraint call: alldifferent([X,Y,Z]), which prescribes that the eventual values of X, Y and Z are all different, i.e., pair-wise distinct. Having made these constraint calls, the conceived constraint store contains the following information:

- X {1..3}, Y {1..3}, Z {1..3}
- alldifferent([X,Y,Z])

Now, let's say that we perform a search call, using Eclipse's member/2 procedure: member(X, [1, 2, 3]). This procedure will assign/unify its first argument to be a member of the list which is supplied as its second argument. The implementation of the procedure is such that it will pick members in the order of the given list. So, it should assign X to 1. The effects of this assignment on the constraint store are for Eclipse to update the ranges of the remaining logical variables as part of constraint propagation. The conceived constraint store now contains the following information:

- Y {2,3}, Z {2,3}
- alldifferent([Y,Z])

As can be seen the ranges of Y and Z have updated dynamically. If we next evaluate member(Y, [1,2,3]), then assigning Y to 1 may no longer occur. Through backtracking member/2 will eventually assign Y to have the value 2. The effect of constraint propagation will then be to *automatically assign* Z to have the value 3, as this is the single remaining valid value for it to have.

In summary, writing Eclipse goals amounts to writing constraint and search calls, and for the effects of search calls to be propagated to constraints within a conceived constraint store by means of constraint propagation.

3 Link Resolution When Using Constraints

As Constraint is a function type, an instance of a Constraint will be evaluated as part of link resolution. This may happen either when a link reference refers in part to such an instance, or as part of the depth-first traversal of the component description hierarchy, which is undertaken as part of link resolution. In either case, once a constraint instance has been processed, i.e. its constraint strings have been evaluated, it is marked as processed, and is not processed again whenever it is referred to subsequently.

To elaborate, let's revisit the example presented in the Introduction, which we give again here for convenience.

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

When performing link resolution, a depth-first, top-down traversal of the sfConfig hierarchy will be undertaken. This means that we start with the attribute foo in the example. As this is a function type, it is evaluated. What this means is to resolve all of its attributes in turn, and then to apply the function, which for Constraint types means that we evaluate the pertaining constraint string/s.

We resolve all of the attributes of foo in top-down order. The first attribute, x, is a link reference, part of which refers to another constraint type, foo2. This means that the constraint function type foo2 will be evaluated at this point. To evaluate foo2, we do exactly the same. We resolve all of its attributes and then evaluate its constraint string. Resolving a VAR is trivial, it simply yields the VAR itself; and, as such, the value that is assigned to the attribute remains the same. We then proceed to evaluate foo2's single (in this case) constraint string; and, having done so, we mark foo2 as having been evaluated. As a side-effect of this particular evaluation, x will have been assigned; although it need not have been. The value assigned to x in foo2 is copied to the x in foo. (By copying, we mean that the link reference is replaced by a *shared* representation of the value). The remaining attributes in foo are resolved, including z which is another function application. Then, the constraint string is evaluated in foo.

Note that constraint evaluation, as carried out by the underlying constraint engine, may be subject to arbitrary backtracking at any stage. This may have the effect of not only backtracking through the evaluation of the current Constraint, but a number thereof. In this event, all of the assignments made within the link resolution process (both to attributes made as part of constraint evaluation, but also to attributes made as part of other function evaluation, such as according to an arbitrary arithmetic expression, as is the case for foo:z in the example) between the point of failure, which initiated the backtracking, and the point to which link resolution has backtracked, will be undone.

This may be seen in the example. We initially set foo2:x to have the value 2, which also causes foo:x to be similarly assigned. We then perform a function evaluation to arrive at the value of foo:z, which yields 3 for this attribute. On evaluating the constraint string, however, there is a failure to assign y to be 6. Backtracking then occurs, unrolling link resolution to the last choice point. This is where we assigned foo2:x to have the value 2. As an alternative, we could have chosen 1. As part of this backtracking, the assignment to foo:z is also undone.

Thereafter, we assign z and y to both be the value 2. This is the sum of performing link resolution on the foo sub-tree. Link resolution then moves to foo2; but, as this has already been marked as having been processed, it trivially returns. We are then finished link resolving sfConfig.

Note that the constraint strings within a constraint are aggregated into a single goal, in the lexical ordering given by their attribute names. Once the single goal has been evaluated the Constraint type is marked as being done. When the depth-first, top-down link resolution traversal reaches the attribute which specifies that it extends the Constraint, the value of the attribute is set to be the component description that underwrites the Constraint type (rather than the Constraint type itself, which is a reference). Whenever a reference is link resolved which points to this type, the component description is returned also.

4 Basic Syntax Overview

In this section, we present some basic aspects of the syntax of the constraint language supported by SmartFrog. Note that, with some exceptions, any Eclipse syntax may be used within constraint strings (see Introduction) with original Eclipse semantics being maintained. It is not worth enumerating the exceptions here. If an issue with using certain Eclipse functionality within constraint strings becomes evident, then it is reasonably likely that there is some conflict constituting one of these exceptions. In this event, the simplest course of action is to define a new procedure in a separate source file (see Section 4.2.5), and then call this procedure within a constraint string. The original Eclipse semantics are applied *without any exceptions* to additional source files.

However, it is strongly emphasised that we provide a number of abstractions from primitive Eclipse syntax to ease the process of writing SmartFrog descriptions with constraints. This provision is to the extent that, for many intended uses, the syntax presented herein should be sufficient.

We now proceed with basic aspects of syntax, as would be used within SmartFrog constraint strings. We present further syntax provisions in Section 5.

4.1 Control flow operators

4.1.1 ‘,’ ‘;’

The operator ‘,’ conjoins calls within a constraint string (which is an Eclipse goal). Calls are made from left-to-right. That is to say, the constraint string

```
a,b,c
```

prescribes that a is called, followed by b, followed by c. Note that, if any of these calls fail (i.e., evaluates to false), then the entire goal fails.

The operator ‘;’ may be used to prescribe a disjunction, such that just one of its operands need to be called and succeed. That is to say, the constraint string

```
a;b;c
```

prescribes that just one of the calls succeeds. The evaluation is also done left-to-right, and only that which is sufficient for the whole goal to succeed is evaluated. For instance, if a succeeds, then b and c are never tried.

Parentheses may also be used to force appropriate precedence. Notably, ‘,’ binds more tightly than ‘;’.

4.1.2 ‘->’ ‘-->’

For writing if-then-else blocks within constraint strings. ‘-->’ is else-less, containing just a ‘then’ part, whereas ‘->’ contains an else part as well as a then part. The condition of an if-then-else instance is just a standard goal. (It should not make use of the active constraint syntax, presented in Section 4.4).

For ‘-->’, the appropriate syntax is: (*condition --> thenpart*). For ‘->’, the appropriate syntax is: (*condition -> thenpart **;** elsepart*).

There are a few examples in Section 5 that use such syntax. An example is:

```
(sfArrayIndex==0 --> bar=0)
```

which simply says that if the value of sfArrayIndex, within the pertaining description, is zero then assign the bar attribute to have the value zero.

Using '->', this would be written:

```
(sfArrayIndex==0 --> bar=0; true)
```

where **true** is trivially consumed. Its opposite, to no great surprise, is **false**. If you write false in a goal, the current branch of the execution tree will fail, and constraint evaluation will backtrack to the last choice-point.

4.2 Assignment Operators

4.2.1 '='

A binary operator, '=' may be used to term-unify the values of two attributes, or the value of one attribute with some term. It is more straightforward for beginners to restrict themselves to the first of these; that is, to assign literal values to attributes within a constraint description, and then refer to these attributes from within the constraint string. We are then effecting the former when using the '=' operator, namely, term-unifying the values of attributes.

Alternatively, it may be desired to refer to literal values directly within a constraint string. To do so, it is necessary to observe the following.

- SmartFrog Strings, e.g. "foo", "the the", when represented in a constraint string, must either:
  - Have the double-quotes escaped, i.e. \"foo\", \"the the\", or
  - Use single-quotes instead, i.e. 'foo', 'the the', or
  - In the case of single word/token strings, do not need any adornments (i.e. quotes marks!) at all, i.e. foo, but not the the.
- For vectors:
  - When an unquoted single word/token appears in a vector, it is treated within constraint processing as a string, e.g., [foo] would be ["foo"] when written as the value of a SmartFrog description attribute.
- Any token starting with an uppercase letter, or an underscore '_', is a variable. These may appear in constraint strings, for example attr1 = [foo, X, Y].

Note that often we will say that '=' assigns the value of an attribute to some other value. Although, in many cases, it may be convenient to think of it in this way, it should always be remembered that we are performing unification on attribute values. What this means is that:

- Either argument may be instantiated and the other not at all, which is not like imperative-language assignment where the left-hand side takes on the value obtained from evaluating the right-hand side
- Both arguments may be wholly uninstantiated, i.e. no refinement has taken place of their respective original VAR forms, which means that we are marking that both attributes will take on the same value when one of them is eventually instantiated, through unification with an instantiated term
- Both arguments may be fully-instantiated, which means that we are checking that they have the same value

- Both arguments may have some arbitrary level of instantiation, in which case we are attempting to conflate their respective values through term-unification.

4.2.2 'is'

A binary operator, 'is' is used to assign the value of an attribute to be the result of evaluating some arithmetic expression. For example, within a constraint string, we may write "attr is 2+3". This will have the effect of setting the value of attr within the description to be the value 5.

4.2.3 member

A binary procedure which assigns its first argument to be a member of the list given as the second argument. For example:

```
member(X, [1,2,3])
```

will assign X to be one of 1, 2 or 3. Note that it selects them in the order given and will only select a member from a given position if all assignments to members at previous positions cause failures leading to them being undone.

4.2.4 allocate

The eight-argument procedure **allocate** performs resource allocation. The arguments in order are:

- A list of resource providers. Example: ["host0", "host1", "host2"]
- A list of provider capabilities where member *i* in the list corresponds to the capabilities of member *i* in the list of resource providers. Each member in the list of capabilities is itself a list of arbitrary arity. Each such list must be of the same arity as the others. Capabilities may pertain to anything, for instance memory, processor speed, and so on; but each must be specified as integers. Example: [[4], [3], [3]] – let's say these are memory capabilities, where for simplicity we consider singleton capabilities.
- A list of consumers. Example: ["vm0", "vm1", "vm2", "vm3"]. We consider an example of allocating vms on hosts.
- A list of consumer requirements. Similar to the list of provider capabilities. Each list within the list of consumer requirements must be of the same arity, and of the same arity of each of the lists in the list of provider capabilities. Example: [[3], [3], [2], [2]].
- A list of colocation constraints. Each colocation constraint is a list of consumers which must be allocated to the same resource. Example: [["vm0", "vm1"]].
- A list of noncolocation constraints. Each noncolocation constraint is a list of consumers which must not be allocated to the same resource. Example: [["vm0", "vm2"], ["vm0", "vm3"]].
- A list of hosted constraints. Each hosted constraint is a 2-member list of [consumer, provider] specifying that consumer must be allocated to provider. Example: [["vm0", "host0"], ["vm1", "host0"]].

Let's consider an example:

```
sfConfig extends Constraint {
   providers ["host0", "host1", "host2"];
   provider_caps [[4], [3], [3]];
   consumers ["vm0", "vm1", "vm2", "vm3"];
   consumer_reqs [[2], [2], [3], [3]];
   allocation VAR;
   [sfConstraint] -- "allocate(providers, provider_caps, consumers, consumer_reqs, allocation,
[], [], [])";
}
```

The output from parsing this description is:

```
providers [|"host0", "host1", "host2"|];
provider_caps [|[|4|], [|3|], [|3|]|];
consumers [|"vm0", "vm1", "vm2", "vm3"|];
consumer_reqs [|[|2|], [|2|], [|3|], [|3|]|];
allocation [|"host0", "host0", "host1", "host2"|];
```

As can be seen the allocation VAR has be instantiated with the allocation, which observes the ordering of the consumers list.

4.2.5 source, add_path

The procedure source/1 compiles additional Eclipse sources should the need to define additional Eclipse-based procedures arise. It will look for these sources in the current working directory as default. Additional paths can be added using add_path/1.

Example:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends Constraint {
  [sfConstraint] -- "source('foo.ecl'), foo, add_path('//C/'), source('foo2.ecl'), foo2";
}
```

The constraint string in sfConfig will attempt to compile foo.ecl from the current working directory, and will attempt to compile foo2.ecl from the current working directory firstly, and then from //C/.

The contents of foo.ecl and foo2.ecl are:

```
foo :- writeln("I am foo"), flush(stdout).
```

```
foo2 :- writeln("I am foo2"), flush(stdout).
```

The procedures writeln/1 and flush/1 are built-ins. Users of the SmartFrog constraint extensions would not ordinarily be concerned with these procedures.

The output is as we would expect.

```
I am foo
I am foo2
```

4.3 Passive test operators

These are operators which allow us to test the values of attributes, where all arguments *must be fully instantiated*. They are called passive operators in contrast to the active

constraint operators. An expression making use of a passive operator is evaluated in full once it is reached. This evaluation yields either success or failure, which in the latter case will cause backtracking.

4.3.1 '==' '\=='

Binary operators which test whether the values of two attributes, or some attribute value and a literal, are the same (==), or different (\==).

4.3.2 '<' '>' '>=' '=<'

Inequality operators, where arbitrary arithmetic expressions are permitted, e.g. "attr1 < (2+3)".

4.4 Active constraint operators

When using active constraint operators, in some call expression in a goal, evaluation of the expression is suspended until its constituents are sufficiently instantiated for it to be refined or evaluated. In this occurrence, evaluation of the remainder of the goal continues until such time, at which point the expression may yield success, failure, or may be resuspended awaiting further instantiation. The use of active operators is thus clearly different from that of passive operators.

4.4.1 'subtype'

A binary operator:

```
attr1 subtype [attr2, …, attrn]
attr1 subtype attr2  /**This is shorthand for: attr1 subtype [attr2]**/
```

This active constraint prescribes that the value assigned to attr1 is a Component Description and is a sub-type of the component description that results from an evaluation applied to [attr2, …, attr*n*].

The evaluation performed is to construct a component description starting with the component description (*cd2*, say) that is the value of *attr2*. We then extend this description with the component description (*cd3*, say) that is the value of *attr*3. What this means is that we preserve the ordering of *cd2*, overwriting any common attributes *in situ* (meaning that the original ordering of attributes is preserved) with their respective values in *cd3*, and appending any non-common attributes in *cd3* to the end of the description being constructed. We repeat this process for the remaining attributes specified in the list [attr2, …, attr*n*], in the order prescribed by the list, such that *cdn* is the final description to overwrite common attributes and have its non-common attributes appended to the end of the description being constructed.

Say, we have the following SmartFrog fragment.

```
foo extends {
  bar 1;
  bar2 2;
}

foo2 extends {
  bar2 3;
  bar3 4;
}

foo3 extends {
  bar 5;
  bar4 6;
  bar3 7;
}
```

If the right-hand side of a subtype constraint were [foo, foo2, foo3], then the construction process just described would start with foo and foo2, overwriting common attributes *in situ*, in foo, namely bar2, and appending non-common attributes of foo2. This would yield:

```
extends {
  bar 1;
  bar2 3;
  bar3 4;
}
```

Repeating the process with foo3 yields the following as the constructed description:

```
extends {
  bar 5;
  bar2 3;
  bar3 7;
  bar4 6;
}
```

As can be seen the description that results is a mish-mash of the descriptions that are used in the construction process.

A description foobar, resolved as the value of an attribute attr1, is a sub-type of [attr2, …, attrn], as prescribed thus:

```
attr1 subtype [attr2, …, attrn]
```

iff the super-type is constructed according to the construction process, just described, *and* iff the first *n* attributes of foobar have the same name as the *n* attributes that fully comprise the constructed super-type, where they share the same ordering.

4.4.2 alldifferent

A unary *procedure*:

```
alldifferent(Attributes)
```

where Attributes is a list of attributes, e.g.:

```
alldifferent([foo, foo2, foo3])
```

This prescribes that the values assigned to attributes foo, foo2 and foo3 must all be different, i.e. pair-wise distinct.

4.4.3 ‘eq’, ‘neq’, ‘lt’, ‘lte’, ‘gt’, ‘gte’, ‘nt’

Binary operators which are suspended until their arguments are fully grounded. See Section 5 for some examples.

- eq: tests for equality of its operands (values of attributes, or literals). equal may also be used.
- neq tests for inequality. Also notequal.
- lt – less than. Also lessthan.
- lte – less than equal. Also lessthanequal.
- gt – greater than. Also greaterthan.
- gte – greater than equal. Also greaterthanequal.

4.4.4 ‘and’, ‘or’, ‘implies’

Binary operators which may be used to combine active constraints. These are evaluated whenever their component constraints are sufficiently instantiated. *They may only be used to combine constraints which use those operators presented in Section 4.4.3*. See Section 5 for some examples of using *and*, *or* and *implies*.

- and, prescribes that both operands (eventually) hold true
- or, prescribes that just one of its operands (eventually) holds true
- implies, used for writing conditional constraints.

5 Further Aspects of SmartFrog's constraint support

5.1 VARs

In a SmartFrog Constraint type, the value part of an attribute may be specified as a VAR. This indicates that it meant to be filled in at some point during constraint evaluation; notably, not necessarily during the evaluation of the constraint of which it is a member. Alternatively, it may be filled in during the evaluation of another constraint type.

The general form for a variable declaration is:

```
attr VAR range default
```

Both the range and default are optional; although, it is recommended that the range always be present when:

- It has the possibility of being instantiated/assigned during the evaluation of the constraint to which it is attached, and
- It is intended that it be assigned to a string or number literal

More details concerning the range specifier:

- Only integer and enumerated ranges of non-integer values (hereafter, we will just write enumerated ranges, for simplicity) are allowed. Mixing of ranges is not allowed. An enumerated range must consist of members which follow the syntax rules for Eclipse atoms, namely they must be sequences of characters starting with a lower-case letter or a single quote-mark '. An example is: [a,b,c]. Within SmartFrog vectors, such members must be specified as strings, i.e. ["a", "b", "c"].
- It must be present if any active constraints are specified for the pertaining variable.
- It may be either a string giving the range *in situ*, or a reference which, when resolved, yields the range.
    - For an *in situ* definition, the string will specify the range within [] delimiters, where members and/or sub-ranges of the range may be specified, separated by commas. A sub-range specifies an interval, whereby lower- and upper-end values of the interval are composed using two dots ('..'). Examples:
        - "[1,2,3]" -- integer range
        - "[1..100, 1000]" – integer range. Note the use of the double-dots.
        - "[fred, bob, sue]" – enumerated range
    - When range is a reference, it must resolve to an appropriate range, which may be a string, as in the in situ case. It may also be a vector, specifying a fully-elaborated (i.e. no sub-ranges) integer or enumerated range. The capability of specifying a vector is allowed for convenience, notwithstanding it not being supported for in situ definitions. Examples:

```
foo extends Constraint {
    attr0 VAR bar;   /*Specifying by reference allows the same range to*/
    attr1 VAR bar;   /*be reused*/
    bar [1,2,3];     /*Vector with explicitly specified members.  Here
integers*/

    attr2 VAR bar2;
    bar2 "[1,2,3]";   /*As a string*/

    attr3 VAR bar3;
    bar3 VAR "[1..100, 1000]"   /*Must be as a string if using sub-ranges*/

    attr3 VAR bar3;
    bar3 ["fred", "bob", "sue"];  /*When specifying an enumerated range of*/
                                  /*atoms as a vector we must write members*/
                                  /*as strings*/

    attr4 VAR bar4;
    bar4 "[fred, bob, sue]"       /*Alternatively, just write as a string,*/
                                  /*then no need*/
}
```

- It is possible to assign the value of an attribute to be that of another attribute within a constraint string, where the value being assigned is in fact a component description. In order to write ranges where the possible values to be assigned are component descriptions, an enumeration of the attribute names (within the context of the Constraint being evaluated) to which these descriptions are attached is used for the range. See Section 5.1.2 for an example.
- Enumerated ranges must not contain members which are compound terms, or which are atoms beginning with sf, or 'sf. Generally speaking, sf is a reserved prefix in SmartFrog.

Note that there is a pre-processing stage that occurs when aggregating the constraint strings in a Constraint type for evaluation by the constraint engine. During this pre-processing stage, any atoms, used for example within unification/assignment call expressions – see Section 4.2.1, that are attributes within the context of the Constraint type being evaluated are interpreted as being those attributes and are not interpreted as being literals specifying the attribute names as their values. This means that one should avoid using attribute names/ values in enumerated ranges that clash.

A default value will either be a number or a string literal encapsulating an atom. Default values are applied to all VAR-specified attributes within a Constraint type which are not assigned a value in the course of evaluating the constraint, either by virtue of a constraint string, or in being an automatic variable (see Section 5.2). That is, automatic variable assignment takes precedence over default value assignment; tagging a variable as automatic as well as giving it a default value is nonsensical. Default value assignment takes precedence over user variable assignment (see Section 5.3), however. Similarly, tagging a variable as a user variable, as well as specifying a default value for it is nonsensical.

Consider the simple example:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
   foo extends Constraint {
      bar VAR "[1..3]" 3;
   }
}
```

This yields:

```
foo extends DATA {
       bar 3;
}
```

Also note that VARs may be specified within lists when used in constraint strings, such as [“one”, VAR, “three”, VAR], as done in the next example.

5.1.1 List Example

In the following example, we assign elements from a list to three different attributes, with the caveat that all three values must be different, i.e., pair-wise distinct. According to the arrangement of link references between the various Constraint types, in the example, the genList Constraint gets evaluated first.

```
#include "/org/smartfrog/functions.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 theList;
      y VAR theList;
      z VAR theList;
      [sfConstraint] -- "alldifferent([x,y,z])";
   }

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

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

This constraint simply performs term-unification on [“one”, VAR, “three”, VAR] and [VAR, “two”, VAR, “four”], meaning that the list [“one”, “two”, “three”, “four”] is assigned to both genList:theList and genList:aList.

Next to be evaluated, according to the ordering observed in link resolution, is the elements constraint. This simply prescribes that the values assigned to its local attributes elements:x, elements:y and elements:z are to be all different.

Notably, x:el (resp, y:el, z:el) refers through a link to the attribute elements:x (resp. elements:y, elements:z). This means that, from the perspective of link resolution, they are the same variable. In evaluating the constraint string of the Constraint type x, a member of genList:theList is selected, and assigned to x:el. In evaluating Constraint y, the chosen value for y:el must be different from that assigned to x:el, because of the alldifferent constraint. The same applies for z:el, whose value must be different from both x:el and y:el.

When we run this example through the SmartFrog parser, we get the following output, where we note that the values set for x:el, y:el, z:el are indeed all different.

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

Note, as mentioned previously, a VAR does not have to be assigned during evaluation of the Constraint type to which it is attached. We can see this in this example where the alldifferent constraint is made within the elements Constraint for its contained variables, x, y, and z; but, instantiation of these does not happen until the Constraint types x, y, and z are evaluated.

5.1.2 Subtyping examples

The following example is interesting because it shows the subtype constraint at work. In the example, we assign foo to have a value which is a component description. As noted previously, the range declaration for foo (in being a VAR) is an enumeration of the local attribute names that may be used, when resolved, to be candidate component descriptions.

It is important to note that enumerated ranges must not be a mix of attribute and non-attribute names (within the context of the Constraint type). Moreover, authors *must avoid* using attribute names within enumerated ranges if they do not want them to be interpreted as being attribute names pertaining to component descriptions, where the component description itself is used in assigning values and not the name.

As a general rule of them the set of attribute names in a Constraint's context and the values used in enumerated ranges must be pair-wise distinct, unless it is intended that the said behaviour should be effected.

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  foo1 extends Constraint {
     foo2 extends {
        bar "32";
        bar2 "33";
     }
     foo3 extends foo2 {
        bar2 "48";
        bar3 "51";
     }
     foo4 extends {
        bar4 "73";
     }
     foo VAR "[foo3, foo4]";
     [sfConstraint] -- "foo = foo4; foo = foo3";
  }
}
```

In the constraint string, foo is initially set to foo4. As this attribute exists in the Constraint's (immediate) context, foo is instead set to the resolution of foo4. This is extremely useful for assigning component descriptions from with constraint strings.

The result from the SmartFrog parser is as follows, where foo takes on the value of foo4.

```
foo1 extends DATA  {
  foo2 extends  {
    bar "32";
    bar2 "33";
  }
  foo3 extends  {
    bar "32";
    bar2 "48";
    bar3 "51";
  }
  foo4 extends  {
    bar4 "73";
  }
  foo extends  {
    bar4 "73";
  }
}
```

Now, as an alternative, let's insist that the value of foo be a subtype of foo2. This is coded thus:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  foo1 extends Constraint {
     foo2 extends {
        bar "32";
        bar2 "33";
     }
     foo3 extends foo2 {
        bar2 "48";
        bar3 "51";
     }
     foo4 extends {
        bar4 "73";
     }
     foo VAR "[foo4, foo3]";
     [sfConstraint] -- "foo subtype foo2, (foo = foo4; foo = foo3)";
  }
}
```

Here, having specified that foo be a subtype of foo2, we make a choice between assigning foo to be foo4 or foo3. foo4 will be tried first, but this assignment will be unrolled because foo4 is not a sub-type of foo2. However, attempting to assign foo to foo3 will succeed. This is shown in the output, where foo takes on the value of foo3.

```
foo1 extends DATA  {
  foo2 extends  {
    bar "32";
    bar2 "33";
  }
  foo3 extends  {
    bar "32";
    bar2 "48";
    bar3 "51";
  }
  foo4 extends  {
    bar4 "73";
  }
  foo extends  {
    bar "32";
    bar2 "48";
    bar3 "51";
  }
}
```

5.2 Automatic Variables

Automatic variables do not need to be explicitly assigned from within a constraint string. Instead, when constraint evaluation has otherwise completed for a Constraint type any attributes (necessarily VARs) which are tagged with [sfConstraintAutoVar] are assigned values from their respective ranges (which must have been specified in the VAR declaration).

An example, following on from the previous section:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  foo1 extends Constraint {
     foo2 extends {
        bar "32";
        bar2 "33";
     }
     foo3 extends foo2 {
        bar2 "48";
        bar3 "51";
     }
     foo4 extends {
        bar4 "73";
     }
     [sfConstraintAutoVar] foo VAR "[foo3, foo4]";
     [sfConstraint] -- "foo subtype foo2";
  }
}
```

Instead of explicitly assigning foo, we mark it as an automatic variable. The constraint string is evaluated first, which simply prescribes that foo be a sub-type of foo2. Then, the single automatic variable will be presented to the constraint engine for assignment. At this point, the engine may only select foo3, because of the sub-typing constraint, which it proceeds to do. The output is the same as above.

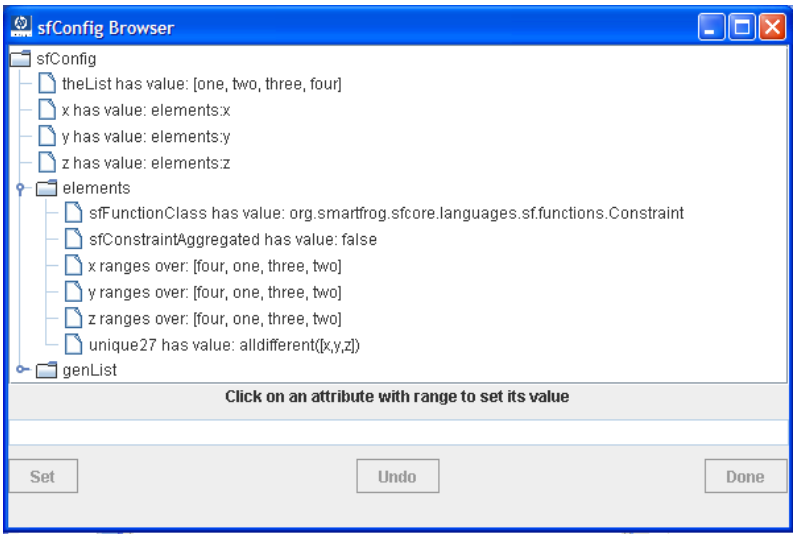
5.3 User Variables

User variables follow a similar theme to automatic variables in that they are variables which are filled if left unassigned during constraint evaluation. This time, however, a GUI pops up for the user to fill in values. User variables are tagged with [sfConstraintUserVar], and ranges for them *must* be specified. Note that user variables are entered once automatic variables have been filled in.

We present a rehash of the list example from previously, viz.

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
   theList genList:theList;

   x elements:x;
   y elements:y;
   z elements:z;

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

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

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

Note the absence of using the member/2 procedure to assign values to attributes. When running sfParse, a window will pop up, thus:

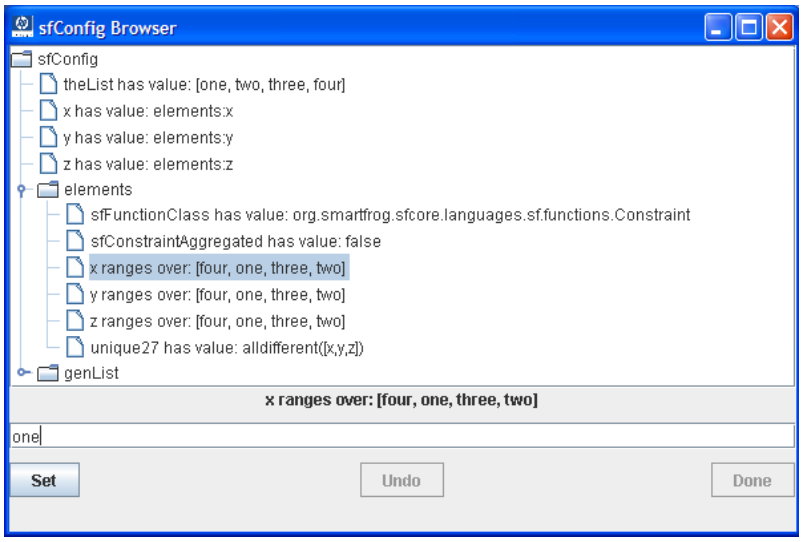


The path leading to the Constraint type (here, elements) is automatically opened up exposing its attributes. All of the user variables that may be set have their ranges specified. In the example, these user variables are elements:x, elements:y and elements:z.

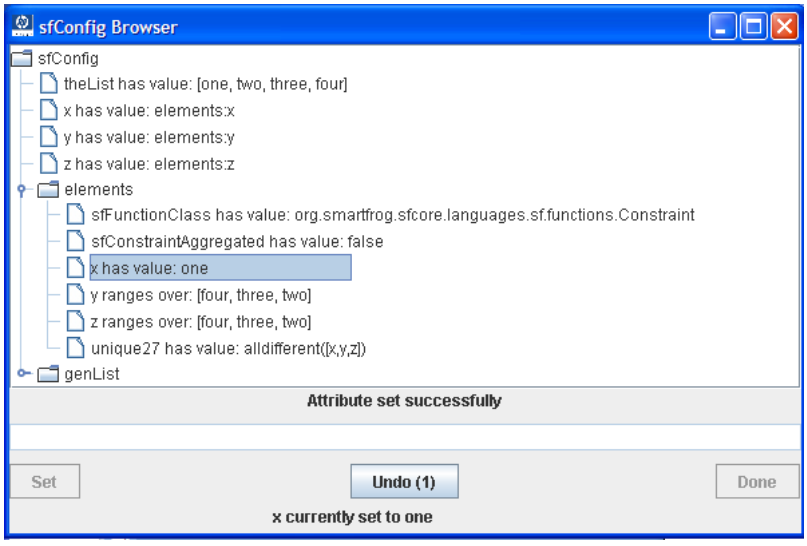
Note that:

- The Set button allows the user to set the value of a user variable, once s/he has selected it in the tree representation and has entered a value in the text field
- The Undo button allows s/he to undo user variables that have been set by her/him in this Constraint type
- The Done button allows s/he to save her/his selections once all user variables in the Constraint type have been assigned
- Ranges are updated dynamically through constraint propagation, as we shall see. This is a really neat feature.

Let's click on elements:x. We see that the Set button becomes enabled:



Let's set the value to be "one". The ranges of elements:y and elements:z update automatically, and the Undo button (labelled 1 to indicate one undo possible) becomes enabled.



At this stage the user could undo this change, but we won't show that. Also if s/he incorrectly enters a value (i.e., not in range) it lets her/him know. Let's set elements:y, and see that the range of elements:z shrinks yet further.

Finally, let's set elements:z. Notice that Done becomes enabled, as we may save these attribute settings and exit.

We may click Done or Undo. We'll click Done and look at the output from sfParse.

```
theList [|"one", "two", "three", "four"|];
x "one";
y "three";
z "four";
elements extends DATA  {
  [ sfConstraintUserVar ] x "one";
  [ sfConstraintUserVar ] y "three";
  [ sfConstraintUserVar ] z "four";
}
genList extends DATA  {
  theList [|"one", "two", "three", "four"|];
  aList [|"one", "two", "three", "four"|];
}
```

We see that the attributes we have set have been saved in the description hierarchy.

5.4 Array Types

Array is another function type. It simply adds a number of copies of a template value, which must be a component description or some function type, to the context of its underlying component description.

In org/smartfrog/functions.sf, the Array function type is defined as follows.

When using an Array type the following attributes are necessary. There is an exception to them all being necessary which is elaborated below.

- sfArrayExtent is the extent of the array. This must be an integer, specifying the *immutable* length of the array, or a vector defining an enumeration of indices. An examples of the latter is [“fred”, “bob”, “sue”].
- sfArrayPrefix defines the attribute prefix of the array members that are added to the Array’s underlying component description. Each attribute name is completed with an appropriate index into the range [0..*n-1*] where *n* is sfArrayExtent for integer extents, or completed with a member of the enumeration for enumerated extents. For example, for the default array type, the members of the array have the names “foo0”, “foo1” and “foo2”. In the case of [“fred”, “bob”, “sue”], the members would be “foofred”, “foobob” and “foosue”, assuming the same prefix.
- sfArrayGenerator defines the template value which is copied for every member added to the array type. It must be a component description or a function type.

  Of particular note is that the sfArrayGenerator is not resolved as part of processing the Array function type. This is different to all other function types, where all attributes are resolved. We do not resolve sfArrayGenerator, as it is appropriate that any processing that this may incur is not done while expanding the Array, but at some later time.

The SmartFrog description:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  baz extends Array {
      sfArrayGenerator extends {
          bar “hello world”;
      }
      sfArrayExtent 3;
      sfArrayPrefix “foo”;
  }
}
```

yields the following output from sfParse. Note that sfArrayIndex gives the unprefixed index of array members, whereas sfArrayTag gives the full index name.

```
baz extends DATA  {
  sfArrayPrefix "foo";
  foo0 extends  {
    bar "hello world";
    sfArrayIndex 0;
    sfArrayTag "foo0";
  }
  foo1 extends  {
    bar "hello world";
    sfArrayIndex 1;
    sfArrayTag "foo1";
  }
  foo2 extends  {
    bar "hello world";
    sfArrayIndex 2;
    sfArrayTag "foo2";
  }
  sfArrayExtent 3;
}
```

As an altenative to the use of single sfArrayGenerator and sfArrayExtent attributes, we may use tags to indicate extents and generators, and allow more than one pair. This means that we can compose an array which has members of more than one generator type. Some restrictions are prescribed, viz.

- Attributes with arbitrary names may be tagged [sfArrayExtent] and [sfArrayGenerator], but they must appear in the extended Array function type in an alternating fashion starting with an [sfArrayGenerator] tagged attribute. There may be intermediate attributes without being tagged as either extent or generator.
- All extents must be integers or all extents must be enumerated ranges. The types of extents may not be mixed.

A valid example (used later on in this report) is:

```
hosts extends Array {
      [sfArrayGenerator] -- extends HostType0;
      [sfArrayExtent] -- 1;
      [sfArrayGenerator] -- extends HostType1;
      [sfArrayExtent] -- 2;
      sfArrayPrefix "host";
}
```

In this example, the SmartFrog parser will generate an array of three members, one of which extends HostType0, while the other two members extend HostType1.

5.5 Aggregator Types

Aggregator types allow us to collect together the values of the same arbitrarily-nested attribute within (indexed) members of an array type. In order to specify the attribute in question, an author necessarily specifies the sfAggregatorPath attribute. This is a string whose contents are like a reference, although not exactly. An example would be “foo:bar”. The first part of the quasi-reference is actually the array type. The remainder of it, in this case just bar (although, we could have bar:bar2:bar3, say) is the path within each array member to the attribute that we seek.

Having collected the values of the common attribute, the aggregator type adds them as arguments to any function type specified as the value of an attribute within the (top-level context of the) Aggregator. Then, it resolves these function types.

Note that although Aggregator is implemented in the SmartFrog core as a function type, it resolves its immediate attributes after carrying out its associated function. This is in contrast to all other function types which resolve their attributes prior to carrying out their respective functions.

An example of an Aggregator at play is now presented.

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  foo1 extends Array {
     sfArrayGenerator extends Constraint {
         bar VAR "[0,1]";
         //sfArrayIndex added in Array processing
         [sfConstraint] -- "(sfArrayIndex==0 -> bar=0; bar=1)";
     }
  }

  foo2 extends Aggregator {
        sfAggregatorPath "foo1:bar";
        foo3 extends concat;
  }
}
```

In the presented SmartFrog description, the aggregator collects values from the array foo1. It seeks attribute values which are resolved from the reference fragment “bar”, with respect to members of the array foo1. In order to get the respective values of bar, the SmartFrog parser will evaluate the respective Constraint type, corresponding to sfArrayGenerator, of each member of the foo1 array.

The constraint string says that if the value of sfArrayIndex is zero then so is bar else bar is 1. The number of array members is 3 (from the default set in the definition of Array). The respective values of sfArrayIndex will be 0, 1 and 2. Thus, the values that the aggregator will collect will be 0, 1 and 1. In processing foo2, having collected these values, the SmartFrog parser will add them as arguments to any extant function types. The single one present is concat. This is evaluated with its new arguments and resolves to the string “011”, as we can see from the following output.

```
foo1 extends DATA  {
  sfArrayExtent 3;
  sfArrayPrefix "foo";
  foo0 extends DATA  {
    bar 0;
    index 0;
```

```
    sfArrayIndex 0;
    sfArrayTag "foo0";
  }
  foo1 extends DATA  {
    bar 1;
    index 1;
    sfArrayIndex 1;
    sfArrayTag "foo1";
  }
  foo2 extends DATA  {
    bar 1;
    index 2;
    sfArrayIndex 2;
    sfArrayTag "foo2";
  }
}
foo2 extends DATA  {
  sfAggregatorPath "foo1:bar";
  foo3 "011";
}
```

There are a number of additional built-in function types provided as a result of developing the constraint support.

5.5.1 forall/exists

forall and exists are particularly useful functions types. They respectively check whether all or at least one of their arguments conform to some criterion. The criterion takes on the form of a Boolean operator, which is specified by the sfAEOperator attribute. Also supplied must be a left or right argument, by virtue of sfAELeftArg or sfAERightArg, respectively.

Each argument of forall (resp. exists) is individually evaluated against this left or right argument, using the specified operator. If all (resp. at least one) of the evaluations yield the value true, then the forall/exists function type yields true, otherwise it yields false.

An example:

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
  foo1 extends Array {
     sfArrayGenerator extends Constraint {
         //sfArrayIndex added in Array processing
         bar VAR "[0,1]";
         [sfConstraint] -- "(sfArrayIndex==0 -> bar=0; bar=1)";
     }
  }

  foo2 extends Aggregator {
        sfAggregatorPath "foo1:bar";
        foo3 extends forall {
           sfAEOperator extends EQ;
           sfAELeftArg 1;
        }
  }
}
```

Here, we check that the value assigned to bar for (indexed) members of the array foo1 is always 1. As this is false, foo3 should resolve to the value false, which can be seen:

```
foo1 extends DATA  {
  sfArrayExtent 3;
  sfArrayPrefix "foo";
  foo0 extends DATA  {
    bar 0;
    index 0;
```

```
    sfArrayIndex 0;
    sfArrayTag "foo0";
  }
  foo1 extends DATA  {
    bar 1;
    index 1;
    sfArrayIndex 1;
    sfArrayTag "foo1";
  }
  foo2 extends DATA  {
    bar 1;
    index 2;
    sfArrayIndex 2;
    sfArrayTag "foo2";
  }
}
foo2 extends DATA  {
  sfAggregatorPath "foo1:bar";
  foo3 false;
}
```

If we change “forall” to “exists”, then foo3 now evaluates to true, as there is at least one instance where bar for (indexed) members of foo1 is 1:

```
foo1 extends DATA  {
  sfArrayExtent 3;
  sfArrayPrefix "foo";
  …snip…
}
foo2 extends DATA  {
  sfAggregatorPath "foo1:bar";
  foo3 true;
}
```

Back to the forall example, we could also validate the resolved value to be what we expect.

```
sfConfig extends {
  foo1 extends Array {
     sfArrayGenerator extends Constraint {
         bar VAR "[0,1]";
         //sfArrayIndex added in Array processing
         [sfConstraint] -- "(sfArrayIndex==0 -> bar=0; bar=1)";
     }
  }

  foo2 extends Aggregator {
        sfAggregatorPath "foo1:bar";
        foo3 extends forall {
           sfAEOperator extends EQ;
           sfAELeftArg 1;
        }
  }
  foo4 extends Constraint {
     result foo2:foo3;
     [sfConstraint] -- "result==true";
  }
}
```

This description will fail to pass, where the verbose output betrays the problem (bold added for emphasis):

```
Solving: false==true
org.smartfrog.sfcore.languages.sf.constraints.EclipseSolver$SmartFrogEclipseRuntimeException:
Unable to solve constraints. General failure.
…
```

Note that, strictly speaking, forall is unnecessary, as the same effect can be achieved within a constraint string commonly attached to each individual member of an array. However, the provision of exists is useful, as its constraining effect can not be achieved within individual member constraint strings. forall is provided as a function type for completeness, and as an alternative.

5.5.2 nsum/nproduct

nsum and nproduct respectively perform an n-way sum and product on their arguments.

5.5.3 alldifferent

alldifferent verifies that all of its arguments are different. Use of this function type is an alternative to using alldifferent/1 (from Section 4.4.2) in a constraint string. alldifferent as a function type may only be used when its arguments have already been instantiated. This is in contrast to the alldifferent/1 procedure that may be used in constraint strings, where the members of its single argument list do not need to have been previously instantiated. It is an active constraint in its constraint string form.

5.5.4 Further Examples

We note some work done as part of the Quartermaster project at HP Labs [3]. Part of this project was concerned with automatic resource configuration. It has been helpful to look at their work as examples to motivate and test our work on constraints.

In one of the example domain descriptions in [3], two interesting constraints are specified, viz.

- satisfy (numCPU == (+i: 0<=i<numOfvPars: vpars[i].numCPU))
- satisfy(forall i:0<=i<nPartition: partition[i].numOfvPars==8)

The first of these specifies a constraint on the sum of a common attribute value within members of an array. If we were to represent such an example, we would use nsum to sum the values of the common attribute, and then use a constraint string to verify that it equals the desired value. Here's a snippet of what such a representation may look like.

```
totCPUs extends Aggregator {
     sfAggregatorPath "vpars:numCPU:val";
     total extends nsum;
}
totCPUsConstraint extends Constraint {
   vParsTotal totCPUs:total;
   partTotal numCPU:val;
   [sfConstraint] -- "vParsTotal==partTotal";
}
```

The second constraint specifies that for all members of an array the value of a particular attribute must be the same across all members. We have seen a representation of such a constraint in SmartFrog, using the constraint extensions, just above.

5.6 AggregatedConstraint Types

An AggregatedConstraint type is a cross between an Aggregator and a Constraint. In contrast to Aggregator, instead of collecting the values of a single attribute within members of an array, it aggregates the values of any number of attributes. Instead of adding values individually to arbitrary function types as Aggregator does, AggregatedConstraint collates them into lists of values, one list for every attribute path (within an array) specified. It then

makes these attributes available for constraint solving, which takes places after the collation of attribute values, according to the constraint strings specified for the AggregatedConstraint.

It does more! If any of the values collected are VARs which are subsequently instantiated as part of evaluating any extant constraint strings, the AggregatedConstraint type will write these values back *at their original sources*.

Let's take a breath and look at an example. Notably, this is the same example as that presented in Section 4.2.4, and constitutes a different way of representing the same problem.

```
#include "/org/smartfrog/functions.sf"

Host extends Constraint {
   [sfConstraintAutoVar] host_type VAR host_types;
   host_types ["ht0", "ht1"];

   [sfConstraintAutoVar] memory VAR memory_types;
   memory_types [3, 4];

   caps [memory];

   //sfArrayIndex -- added in Array processing

   [sfConstraint] -- "sfArrayIndex eq 0 implies memory eq 4 and host_type eq ht0";
   [sfConstraint] -- "sfArrayIndex gt 0 implies memory eq 3 and host_type eq ht1";
}


VM extends Constraint {
   [sfConstraintAutoVar] vm_type VAR vm_types;
   vm_types ["vmt0", "vmt1"];

   [sfConstraintAutoVar] memory VAR memory_types;
   memory_types [2, 3];

   reqs [memory];

   allocated VAR;

   //sfArrayIndex -- added in Array processing

   [sfConstraint] -- "sfArrayIndex lt 2 implies memory eq 2 and vm_type eq vmt0";
   [sfConstraint] -- "sfArrayIndex gte 2 implies memory eq 3 and vm_type eq vmt1";
}

sfConfig extends ResourceAllocator {
   providers "hosts:sfArrayTag";
   provider_caps "hosts:caps";
   consumers "vms:sfArrayTag";
   consumer_reqs "vms:reqs";
   allocation "vms:allocated";

   hosts extends Array {
      sfArrayExtent 3;
      sfArrayGenerator extends Host;
      sfArrayPrefix "host";
   }

   vms extends Array {
      sfArrayExtent 4;
      sfArrayGenerator extends VM;
      sfArrayPrefix "vm";
   }
}
```

sfConfig extends ResourceAllocator, which is a built-in type, specified within functions.sf.

```
ResourceAllocator extends AggregatedConstraint{
   sfAggregatedConstraintSource0 providers;
   sfAggregatedConstraintSource1 provider_caps;
   sfAggregatedConstraintSource2 consumers;
   sfAggregatedConstraintSource3 consumer_reqs;
   sfAggregatedConstraintSource4 allocation;
   colocations [];
   nocolocations [];
   hosted [];
   [sfConstraint] -- "allocate(sfAggregatedConstraintVals0, sfAggregatedConstraintVals1,
sfAggregatedConstraintVals2, sfAggregatedConstraintVals3, sfAggregatedConstraintVals4,
colocations, nocolocations, hosted)";
```

As can be seen, it is an AggregatedConstraint. The constraint string calls the built-in procedure, allocate/8, presented in Section 4.2.4. To recap, this procedure allocates consumers with certain requirements on to providers with certain capabilities.

In the definition of ResourceAllocator, we see a number of attributes beginning with sfAggregatedConstraintSource. These are the various attribute paths that we seek to collate. In the case of ResourceAllocator all of them are link references to attributes that the user specifies values for when they extend ResourceAllocator. In the example, we can see the values that are specified for the providers, provider_caps, consumers, consumer_reqs and allocation attributes. For instance, providers is specified to be “hosts:sfArrayTag” meaning that the tags of the members of the hosts array should be collated.

When we make use of the collated attributes in constraint strings, we replace *Source* in sfAggregatedConstraintSource with *Vals*. This is because the AggregatedConstraint type adds these attributes to the AggregatedConstraint type as part of its pre-processing, prior to evaluating any extant constraint strings. The values of these attributes are the various collated lists of values from array members. So, sfAggregatedConstraintVals0 will contain the collation of the various hosts’ array tags, i.e., [“host0”, “host1”, “host2”].

The context for the example is the allocation of virtual machines on hosts. The capabilities and requirements of hosts and machines may be an arbitrary list of integer values. The only constraint is that they are uniform in size. For this example, we consider just memory; thus, the capabilities and requirements lists are single-member.

Let’s provide a narrative of what occurs in processing this description. sfConfig is a ResourceAllocator, which in turn is an AggregatedConstraint. This is a function type, which is evaluated.

Firstly, each of its attributes are resolved, which will cause the hosts and vms arrays to be expanded. There are 3 hosts – “host0”, “host1” and “host2” – and 4 vms – “vm0”, “vm1”, “vm2” and “vm3”.

Then, all of the attributes which start with sfAggregatedConstraintSource are inspected. For instance, at this point, the value of sfAggregatedConstraintSource0 will have been resolved to “hosts:sfArrayTag”. Each of these attributes should be a string literal, specifying which attribute value is to be collated across a given array type.

In the case of sfAggregatedConstraintSource0 the array is hosts, and the attribute is given by the reference fragment sfArrayTag. We obtain the value of sfArrayTag from each of

“host0”, “host1” and “host2”, which are members of the array hosts. In fact sfArrayTag is the name of each member, so the collated list will be [“host0”, “host1”, “host2”].

As part of the process of collating the sfArrayTag attribute, each of the Constraint types that constitute the members of the hosts array are resolved. What this means in real terms is that the constraint types will be evaluated.

The constraint strings in the Host (resp. VM) constraint type constrain how the automatic variables, memory and host_type (resp. vm_type) may be assigned.

For sfAggregatedConstraintSource1, we collate “hosts:caps”. In hosts, the caps member is a vector which is singleton and contains a reference to the memory attribute. From a general standpoint, the caps list could be of any length, collating a number of hosts capabilities.

For sfAggregatedConstraintSource2 and sfAggregatedConstraintSource3, we collate vms and vm requirements. This process is similar to the collation of hosts and capabilities.

The sfAggregatedConstraintSource4 attribute refers to the “allocated” attribute in members of the vms array. This is a VAR and will be filled in as part of the evaluation of the constraint string.

The constraint string calls allocate/8 on the *Vals* counter-parts of the *Source* attributes just described. allocate/8 will fill in sfAggregatedConstraintVals4 with the allocations of vms to hosts. The AggregatedConstraint type will write these values back to the allocated attribute in each member of the vms array.

Let’s see the full output from parsing this description using sfParse, where we can see that the allocated attributes of vms:vm0, vms:vm1, vms:vm2, and vms:vm3 have been appropriately filled in.

```
Provisionally Allocating: vm0 on host: host0
Provisionally Allocating: vm1 on host: host0
Provisionally Allocating: vm2 on host: host1
Provisionally Allocating: vm3 on host: host2
vm0 deployed on host host0
vm1 deployed on host host0
vm2 deployed on host host1
vm3 deployed on host host2
PLACEMENT SUCCESS: Successfully placed VMs.

...snip...

sfAggregatedConstraintSource0 "hosts:sfArrayTag";
sfAggregatedConstraintSource1 "hosts:caps";
sfAggregatedConstraintSource2 "vms:sfArrayTag";
sfAggregatedConstraintSource3 "vms:reqs";
sfAggregatedConstraintSource4 "vms:allocated";
colocations [||];
nocolocations [||];
hosted [||];
providers "hosts:sfArrayTag";
provider_caps "hosts:caps";
consumers "vms:sfArrayTag";
consumer_reqs "vms:reqs";
allocation "vms:allocated";
hosts extends DATA  {
  sfArrayExtent 3;
  sfArrayPrefix "host";
  host0 extends DATA  {
    [ sfConstraintAutoVar ] host_type "ht0";
    host_types [|"ht0", "ht1"|];
```

```
    [ sfConstraintAutoVar ] memory 4;
    memory_types [|3, 4|];
    caps [|4|];
    sfArrayIndex 0;
    sfArrayTag "host0";
  }
  host1 extends DATA  {
    [ sfConstraintAutoVar ] host_type "ht1";
    host_types [|"ht0", "ht1"|];
    [ sfConstraintAutoVar ] memory 3;
    memory_types [|3, 4|];
    caps [|3|];
    sfArrayIndex 1;
    sfArrayTag "host1";
  }
  host2 extends DATA  {
    [ sfConstraintAutoVar ] host_type "ht1";
    host_types [|"ht0", "ht1"|];
    [ sfConstraintAutoVar ] memory 3;
    memory_types [|3, 4|];
    caps [|3|];
    sfArrayIndex 2;
    sfArrayTag "host2";
  }
}
vms extends DATA  {
  sfArrayExtent 4;
  sfArrayPrefix "vm";
  vm0 extends DATA  {
    [ sfConstraintAutoVar ] vm_type "vmt0";
    vm_types [|"vmt0", "vmt1"|];
    [ sfConstraintAutoVar ] memory 2;
    memory_types [|2, 3|];
    reqs [|2|];
    allocated "host0";
    sfArrayIndex 0;
    sfArrayTag "vm0";
  }
  vm1 extends DATA  {
    [ sfConstraintAutoVar ] vm_type "vmt0";
    vm_types [|"vmt0", "vmt1"|];
    [ sfConstraintAutoVar ] memory 2;
    memory_types [|2, 3|];
    reqs [|2|];
    allocated "host0";
    sfArrayIndex 1;
    sfArrayTag "vm1";
  }
  vm2 extends DATA  {
    [ sfConstraintAutoVar ] vm_type "vmt1";
    vm_types [|"vmt0", "vmt1"|];
    [ sfConstraintAutoVar ] memory 3;
    memory_types [|2, 3|];
    reqs [|3|];
    allocated "host1";
    sfArrayIndex 2;
    sfArrayTag "vm2";
  }
  vm3 extends DATA  {
    [ sfConstraintAutoVar ] vm_type "vmt1";
    vm_types [|"vmt0", "vmt1"|];
    [ sfConstraintAutoVar ] memory 3;
    memory_types [|2, 3|];
    reqs [|3|];
    allocated "host2";
    sfArrayIndex 3;
    sfArrayTag "vm3";
  }
}
sfAggregatedConstraintVals0 [|"host0", "host1", "host2"|];
sfAggregatedConstraintVals1 [|[|4|], [|3|], [|3|]|];
sfAggregatedConstraintVals2 [|"vm0", "vm1", "vm2", "vm3"|];
sfAggregatedConstraintVals3 [|[|2|], [|2|], [|3|], [|3|]|];
sfAggregatedConstraintFrees4 "true";
sfAggregatedConstraintVals4 [|"host0", "host0", "host1", "host2"|];
```

5.6.1 Alternative Characterisation of Resource Allocation Example

Using Arrays which have multiple generators and extents, we can explicitly define distinct host types (and vm types) and have instances of these types as members of the same array. This leads to arguably a simpler characterisation of the given example, in that we do not need to attach constraint strings to the foregoing single host generator, Host. At the same time, there is more descriptive structure to this characterisation than in the simple allocation example, presented in Section 4.2.4.

```
#include "/org/smartfrog/functions.sf"

HostTypeBase extends {
   memory 4;
   caps [memory];
}
HostType0 extends HostTypeBase;
HostType1 extends HostTypeBase {
   memory 3;
}

VMTypeBase extends {
  allocated VAR;
  memory 2;
  reqs [memory];
}
VMType0 extends VMTypeBase;
VMType1 extends VMTypeBase {
   memory 3;
}

sfConfig extends ResourceAllocator {
   providers "hosts:sfArrayTag";
   provider_caps "hosts:caps";
   consumers "vms:sfArrayTag";
   consumer_reqs "vms:reqs";
   allocation "vms:allocated";

   hosts extends Array {
      [sfArrayGenerator] -- extends HostType0;
      [sfArrayExtent] -- 1;
      [sfArrayGenerator] -- extends HostType1;
      [sfArrayExtent] -- 2;
      sfArrayPrefix "host";
   }

   vms extends Array {
      [sfArrayGenerator] -- extends VMType0;
      [sfArrayExtent] -- 2;
      [sfArrayGenerator] -- extends VMType1;
      [sfArrayExtent] -- 2;
      sfArrayPrefix "vm";
   }
}
```

5.6.2 Alternative Characterisation of List Example

Using an AggregatedConstraint type, we may also present an alternative version of the list example from Section 5.1.1. Instead of explicitly enumerating an elements array, we may instead define it as an Array type, and then used an AggregatedConstraint type to specify the alldifferent constraint.

```
#include "/org/smartfrog/functions.sf"

sfConfig extends {
   theList genList:theList;

   elements extends Array {
      sfArrayGenerator extends {

         element extends Constraint{
            val VAR theList;
         }

         [sfTemp] -- diff;

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

     }
   }

   diff extends AggregatedConstraint {
     sfAggregatedConstraintSource0 "elements:element:val";
     [sfConstraint] -- "alldifferent(sfAggregatedConstraintVals0)";
   }

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

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

In the preceding, we specify an elements array type, whose members principally define a single value *element:val*. *val* is contained within a Constraint type, as all VARs must be used in the context of a Constraint type. This effectively registers them with the underlying constraint engine. However, before going about setting a value for an *element:val* attribute, by means of member selection on *theList*, we need to enforce the alldifferent constraint, as done in the original version of this example. We prescribe this ordering through a temporary attribute, tagged with [sfTemp], which refers to the diff Constraint. As this appears, in the Array type (which is evaluated top-down as all contexts are) prior to the Constraint performing member selection, it will get resolved first.

The output is as one would expect.

```
theList [|"one", "two", "three", "four"|];
elements extends DATA  {
  sfArrayPrefix "foo";
  foo0 extends  {
    element extends DATA  {
      val "one";
    }
    unique28 extends DATA  {
      list [|"one", "two", "three", "four"|];
      el "one";
    }
    sfArrayIndex 0;
    sfArrayTag "foo0";
  }
  foo1 extends  {
    element extends DATA  {
      val "two";
    }
    unique28 extends DATA  {
      list [|"one", "two", "three", "four"|];
      el "two";
    }
    sfArrayIndex 1;
    sfArrayTag "foo1";
  }
  foo2 extends  {
    element extends DATA  {
      val "three";
    }
    unique28 extends DATA  {
      list [|"one", "two", "three", "four"|];
      el "three";
    }
    sfArrayIndex 2;
    sfArrayTag "foo2";
  }
  sfArrayExtent 3;
}
diff extends DATA  {
  sfAggregatedConstraintSource0 "elements:element:val";
  sfAggregatedConstraintFrees0 "true";
  sfAggregatedConstraintVals0 [|VAR29, VAR30, VAR31|];
}
genList extends DATA  {
  theList [|"one", "two", "three", "four"|];
  aList [|"one", "two", "three", "four"|];
}
```

6 Frequently Asked Questions

Here are some (at least conceived) frequently asked questions

6.1 Why no support for reals?

We support constraint problems whose variables have finite domains (or ranges). Often this is not particularly limiting – many of the constraint problems that we have used as motivational examples have involved finite domains. Moreover, problems that involve reals are often finite domain problems, and integers or enumerated types can be used to represent these.

Problems which require variables with infinite domains represent a whole different category from the sorts of problems for which SmartFrog constraint support has largely been conceived. We envisage that in the vast majority of cases where SmartFrog would be used, restricting variables to have finite domains will not be limiting. In any case, it is possible to revert to native Eclipse support for such problems.

What we mean by reverting to native Eclipse support is for an author to write their own Eclipse-based procedures and to call these from SmartFrog constraint strings.

6.2 What about optimisation problems?

Solving optimisation problems is for the time being only supported at the native Eclipse level; that is, outside of the language presented here for embedding constraints in SmartFrog.

The generic structure of an optimisation problem in Eclipse is:

1. Specify constraints, as well as defining a cost function
2. Pass a search procedure along with the cost function to a minimisation procedure.

Specifying problems in this way is fairly straightforward. If demand becomes evident for optimisation support within the SmartFrog constraint language, then we may revisit the provision such support.

6.3 What happens if Eclipse is not installed, or constraint support not enabled?

In this event, a Constraint type's attributes are still resolved; however, no constraint solving takes place, and the underlying component description is simply returned (as described in Section 3).

6.4 What about SmartFrog's support for Assertions and Schemas?

There is clear overlap between SmartFrog's current support for Assertions and Schemas and its new support for constraints, particularly between Assertions and the constraint support. For the time being, the relationship is not a subsuming one; and for this reason both will remain. For instance, it is possible to do run-time schema and assertion checking, which is not possible with the constraint support – constraints are evaluated solely at parse-time, for now.

Also, it is possible that a SmartFrog user may not want to use the Eclipse-based constraint support. (See 6.3). In this event, some built-in support in the form of assertions and schemas may prove useful to the user.

7 References

[1] “Constraint Logic Programming using Eclipse”. Krzysztof R. Apt, Mark Wallace, 2007, ISBN: 0521866286.

[2] “*The Art of Prolog: Advanced Programming Techniques”.* Leon Sterling and Ehud Shapiro, 1994, ISBN: 0262193388.

[3] “Automated Policy-Based Resource Construction in Utility Computing Environments”. Akhil Sahai et al. HP Labs Tech Report. August 21, 2003. HPL-2003-176.