The SmartFrog Constraint Extensions

For SmartFrog Version 3.12

Localised for UK English / A4 Paper

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;
   ...
   barn VAR range;

   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. An example might be: `foo VAR "[1..10]"`. This says that foo is a local (integer) variable with range `[1..10]`.

- `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:

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

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

Using '->', this would be written:

```
(sfIndex==0 --> bar=0; nil)
```

where `nil` is trivially consumed. Its opposite is `fail`. 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 '='

The 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'

The 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, …, attrn]`.

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 `attr3`. 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,`

`…, attrn]`, 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'

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 'nt'

Used to negate a constraint, and may be used in combination with and/or/implies (see next).

4.4.5 '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 such a variable declaration is:

```
attr VAR range
```

The range attribute is optional; although, the range should always be present if it is anticipated that the constraint solver will perform reasoning of the constraint variable's range. This will be the case if the variable is subject to constraint propagation by virtue of the constraints expressed over it. If it will be inevitably assigned before any constraint propagation is applicable, then a range specification is not necessary.

More details concerning the range specifier:

- It may be simply: `INTEGER` or `BOOLEAN`. In the former case, we may wish to declare an indeterminate integer range. The latter is for Boolean constraint variables.
- Alternatively, a finite integer range or an enumerated range may be specified. 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 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.

As an alternative to the syntax presented for VAR declarations, it is possible to declare constraint variables as attributes which extend `Var`, `AutoVar`, `UserVar`, or `DefaultVar`.

An example:

```
    attr0 extends Var {
       range [1,2,3];
    }
```

This would be identical to:

```
   attr0 VAR [1,2,3];
```

The purpose of the alternative syntax is to provide a means of adding further detail to the declaration of a constraint variable. For example, including an `auto` attribute indicates that the constraint variable is an *automatic* variable.

```
  attr0 extends Var {
       range [1,2,3];
       auto;
  }
```

The pre-specified `AutoVar` prototype is defined thus:

```
  AutoVar extends Var {
       auto;
  }
```

Thus, the foregoing example may be written thus:

```
  attr0 extends AutoVar {
      range [1,2,3];
  }
```

Automatic variables do not need to be explicitly assigned from within a constraint string. Instead, *by default*, when constraint evaluation has otherwise completed for a `Constraint` type any constraint variables declared as automatic are assigned values from their respective ranges (which must have been specified in their declarations).

For the VAR syntax, a variable can be designated as being automatic with the tag: `sfConstraintAutoVar`. So, the previous SmartFrog fragment would be identical to:

```
   [sfConstraintAutoVar] attr0 VAR [1,2,3];
```

However, we may also give the `auto` attribute a value.

```
  attr0 extends Var {
      range [1,2,3];
      auto LAZY Label;
  }
```

The value indicates *when* the constraint variable should be assigned a value. As stated, the default (corresponding to no value being specified) is when the `Constraint` type containing the variable has otherwise finished. However, if a value is specified, the constraint variable will not be assigned a value until link resolution has reached the attribute to which the reference given resolves. Somewhere in the model, there must be an attribute called `Label`, that constitutes the resolution of the (attrib) reference `Label` where resolution is performed

relative to the `attr0` prototype. Moreover, the resolved attribute `Label` *must* extend the pre-speicified type `LabellingPoint`. That is,

```
   Label extends LabellingPoint.
```

An example of the utility of this provision is given in Section 7. See also Section 5.2 for more information on automatic variables.

A constraint variable may also be a *user variable*, as denoted by its declaration extending `UserVar`, or as a result of being tagged with `sfConstraintUserVar`. See also section 5.3.

A constraint variable may also specify a default value, which is assigned when the the `Constraint` type containing it has otherwise finished. In this case, the default value is given by a `default` attribute with value being present. Example:

```
attr0 extends Var {
    range [1,2,3];
    default 1;
  }
```

We also provide the `DefaultVar` type. This may help readability of the model, but does nothing more than define `default` as `TBD`, which only serves to ensure the `default` attribute is given a value by the model author. A default value will either be a number or a string literal encapsulating an atom. Consider the simple example:

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

sfConfig extends {
   foo extends Constraint {
      bar extends DefaultVar {
         range "[1..3]";
         default 3;
      }
   }
}
```

This yields:

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

Also note that `VAR`s 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

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 (simplified), viz.

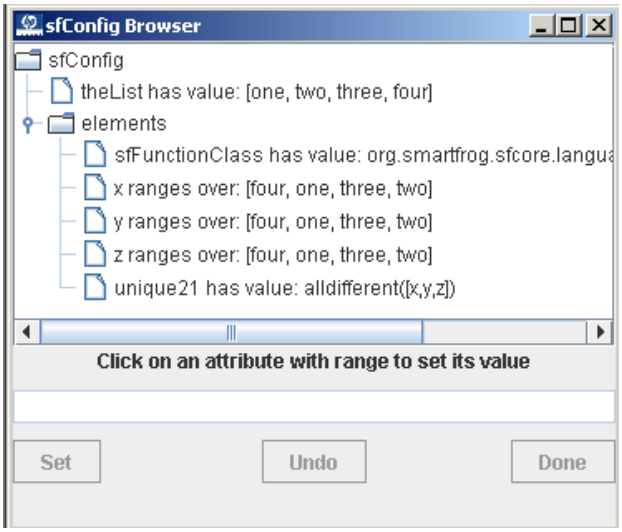
```
#include "/org/smartfrog/functions.sf"

ListVar extends UserVar {
   range theList;
}

sfConfig extends {
   theList ["one","two","three","four"];

   elements extends Constraint {
      x extends ListVar;
      y extends ListVar;
      z extends ListVar;
      [sfConstraint] -- "alldifferent([x,y,z])";
   }
}
```

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 set the value of `elements:x` to be `"one"`. We select this attribute and then type `"one"` in the text field above the trio of buttons. After pressing 'Set', the following window results. Note that 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. Also if s/he incorrectly enters a value (i.e., not in range) it lets her/him know. Once the user selects values for `elements:y` and `elements:z` attributes, and presses 'Done', the GUI window will disappear and link resolution will continue.

5.4 Array Types

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

An array defines one or more extents which together determine the size of the array. For single-dimensional arrays, an extent must be an integer, *n*, or an enumeration, such as `["fred", "bob", "sue"]`. For multi-dimensional arrays, an extent must be a vector which

may have an arbitrary combination of integers and enumerations. We stick with the single-dimensional case for now.

The following attributes are necessary when using an Array type (and are to be specified in the order shown if using tagged attributes).

- A `prefix` attribute which 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]` for integer extents, or completed with a member of the enumeration for enumerated extents. For example, for an integer extent of 3, 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.

  Note that the prefix string may also take the guise of a Smartfrog path, that is: "foo:foo1:foo2:bar", where the last element of the path (here, bar) is the prefix. The other elements of the path are treated a real relative path (relative to the array type) which determines where the array elements should be placed. Thus, for the given prefix string, we would treat "foo:foo1:foo2" as an ATTRIB reference, and once resolved place array members whose names begin with "bar" within the resolved *component description*.

One or more occurrences of the following pair of attributes:

- An `sfExtent`-tagged attribute giving an extent for the array. The size of the array is the sum of the sizes of the extents. If there is a single extent, it may instead by specified using an `extent` attribute.
- An `sfGenerator`-tagged defines the template value which is copied for every member added to the array type for the extent previously specified. It must be a component description or a function type. If there is a single extent, it may instead by specified using an `extent` attribute.

If there is a single extent and generator, they may instead by specified using `extent` and `generator` attributes.

Note that the component description or function type defining the generator **must** extend the type: `ArrayGenerator`, which is defined in `org/smartfrog/functions.sf`.

An example of this is shown next. The SmartFrog description:

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

sfConfig extends {
  foo0 extends Array {
      prefix "foo";
      [sfExtent] -- 3;
      [sfGenerator] -- extends ArrayGenerator {
         bar "hello world";
      }
      [sfExtent] -- ["fred","bob","sue"];
      [sfGenerator] -- extends ArrayGenerator {
         baz "hello people";
```

```
      }
  }
}
```

yields the following output from `sfParse`. Note that sfIndex gives the unprefixed index of array members, whereas `sfTag` gives the full index name.

```
baz extends DATA  {
...snip...
foo0 extends  {
  bar "hello world";
  sfIndex 0;
  sfTag "foo0";
}
foo1 extends  {
  bar "hello world";
  sfIndex 1;
  sfTag "foo1";
}
foo2 extends  {
  bar "hello world";
  sfIndex 2;
  sfTag "foo2";
}
foofred extends  {
  baz "hello people";
  sfIndex "fred";
  sfTag "foofred";
}
foobob extends  {
  baz "hello people";
  sfIndex "bob";
  sfTag "foobob";
}
foosue extends  {
  baz "hello people";
  sfIndex "sue";
  sfTag "foosue";
}
```

5.4.1 Multi-dimensional Arrays

Multi-dimensional arrays (up to an arbitrary dimension) may be defined, thus:

```
sfExtent [9,9]
sfExtent [["bob", "fred", "sue"], ["bob", "fred", "sue"]]
```

`sfIndex0`, `sfIndex1`, … are attached as attributes to members giving the index of the member in the 0th, 1st etc dimension. See the Sudoko example, in Section 7 for use of sfIndex0 etc.

5.5 Aggregator Types

Aggregator types allow us to aggregate/collate 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:

- Optional: `context` attribute, which itself is a DATA description providing further attribute values to use in propositions that determine which array members get collated.
- `array` attribute – a lazy reference specifying the array type to be used
- `prefix` attribute – (resolves to) a string giving the prefix of the array members
- `path` attribute – a lazy reference specifying the path in each array member to the attribute that we seek
- Optional further `pred` attribute – a lazy reference specifying a proposition which determines whether (the value of) the specified attribute is included in the aggregation for particular array members. For the proposition, we may use any attribute value (at arbitrary depth) within the array member, as well as the attributes specified in the `context` data description. The Sudoko example in Section 7 gives further elaboration to the use of `pred` and `context` attributes.

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 {
     prefix "boo";
     extent 3;
     generator extends Constraint, ArrayGenerator {
         bar VAR "[0,1]";
         //sfIndex added in Array processing
         [sfConstraint] -- "(sfIndex==0 -> bar=0; bar=1)";
     }
  }

  foo2 extends Aggregator {
        array LAZY foo1;
        prefix "boo";
        path LAZY bar;
        foo3 extends concat;
  }
}
```

In the presented SmartFrog description, the aggregator collects values from the array `foo1`. The array members are those attributes which have the prefix "`boo`", and we are interested in values of the `bar` attribute within these array members. In this example, in order to get values of bar, the SmartFrog parser will evaluate the containing constraint type for each member of the array.

The constraint string says that if the value of `sfIndex` 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 `sfIndex` 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  {
  ...snip...
  boo0 extends DATA  {
    bar 0;
    sfIndex 0;
    sfTag "boo0";
  }
  boo1 extends DATA  {
    bar 1;
    sfIndex 1;
    sfTag "boo1";
  }
  boo2 extends DATA  {
    bar 1;
    sfIndex 2;
    sfTag "boo2";
  }
}
foo2 extends DATA  {
  ...snip...
  foo3 "011";
}
```

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

5.5.1 nsum/nproduct

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

5.5.2 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.

5.6 Return Values in Constraints

Typically, once a `Constraint` type has been evaluated it is converted to a Component Description. Alternatively, a `Constraint` may specify a "return value" which is instead yielded. This is realised by tagging an attribute in the Constraint with `sfReturn`.

An example:

```
foo extends Constraint {
      input [1,2,3];
      [sfReturn] size VAR;
```

```
      [sfConstraint] -- "length(input, size)";
   }
```

On evaluation, foo will be assigned to 3 which is the size of the input list.

5.7 Aggregations in Constraints

Values (including `VAR`s) may be aggregated from members of array types for use in constraint strings. Whenever an aggregated `VAR` is subsequently instantiated to a value in the constraint, its new value is written back to its original point of declaration.

In a constraint, we may specify a number of such aggregations. An aggregation consists of attributes which must occur in the following order uninterrupted. Aggregations may be interspersed with attributes which do not pertain to aggregations, such as `sfReturn`- and `sfConstraint`-tagged attributes, or general constraint attributes.

- Optional `context` attribute – see Aggregator type, 5.5.
- `array` attribute giving the source array. See 5.5.
- `prefix` attribute giving the prefix of array members. See 5.5.
- `path` attribute giving the path to the attribute whose value is to be collated. See 5.5.

Plus, one or more occurrences of attributes that are data descriptions which extend `AggregateSpecifier`. These will specify at least a `pred` attribute, as well as a possible `unify` attribute. The `pred` attribute is a proposition to discern whether array members should be included in the aggregation, see 5.5. The `unify` attribute if present gives a value with which the aggregation should be unified (see Su Doku example presented in Section 7 for more details). The result of aggregation is assigned to the attribute specifying the `AggregateSpecifier` data description.

Examples are given in Section 7.

5.8 Cardinality Constraints over Arrays

For cardinality constraints, we support a number of function types used to express them over arrays. As can be seen, all of them could be expressed using combinations of `MoreThan` and `FewerThan`, for instance. This is not the only possible consolidation. We provide a range of constructs, notwithstanding some redundancy between them, for convenience.

- `Forall`$_{\texttt{x in Array}}$ `p` – all members `x` of `Array` must satisfy proposition `p` – same as `MoreThan(n-1)`, where `n` is the size of `Array`
- `Exists`$_{\texttt{x in Array}}$ `p` – at least one member `x` must satisfy `p` – same as `MoreThan(0)`
- `MoreThan(N)`$_{\texttt{x in Array}}$ `p` – more than `N` members `x` must satisfy `p`
- `FewerThan(N)`$_{\texttt{x in Array}}$ `p` – fewer than `N` members `x` must satisfy `p`
- `Exactly(N)`$_{\texttt{x in Array}}$ `p` – exactly `N` members `x` must satisfy `p` – same as `MoreThan(N-1) AND FewerThan(N+1)`

For cardinality constraints, the proposition `p` over which a constraint is expressed may be one of the foregoing cardinality function types or one of the following *basic proposition* types:

- `AndProposition(p0,...pn)` – all propositions `p0,...pn` are satisfied in the evaluation context (which is an array member)
- `OrProposition(p0,...pn)` – at least one proposition `p0,...pn` is satisfied in the evaluation context
- `NotProposition(p)` – `p` is not satisfied in the evaluation context
- `IfProposition(p0, p1)` – if `p1` is satisfied in the evaluation context then so is `p0`. (If `p1` is not satisfied in the evaluation context then the proposition is satisfied irrespective of the value of `p0`).
- `IffProposition(p0, p1)` – if `p0` is satisfied in the evaluation context then so is `p1`. If `p0` is not satisfied in the evaluation context then neither is `p1`.
- `Atomic(p)` – `p` is an atomic proposition; that is, it is a boolean expression over the values of attributes relative to the array member evaluation context.

It is possible to specify cardinality constraints to an arbitrary level of nesting, where individual cardinality constraints may refer to different arrays. A cardinality constraint `c0` which contains a cardinality constraint `c1` prescribes the set of evaluation contexts for `c1`, as is the case in quantified formulas in predicate logic. For instance, the cardinality constraint:

`ForAll`$_{\texttt{x in foo}}$`(Exists`$_{\texttt{y in foo2}}$` ( LAZY x:val == LAZY y:bal))`

says that for each member `x` of `foo`, `val` in `x` is equal to the value of `bal` in some member `y` of `foo2`.

An example:

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


sfConfig extends {
  boo0 extends MoreThan {
     [sfTag] -- "x";
     [sfSource] -- LAZY boo1;
     [sfPrefix] -- "foo";
     [sfCard] -- 1;
     [sfProp] -- extends Atomic {
         [sfProp] -- (LAZY x:help == 1);
     }
  }

  boo1 extends Array {
     prefix "foo";
     extent 3;
     generator extends ArrayGenerator, Constraint {
           help extends AutoVar {
                 range "[0..1]";
           }
     }
  }
}
```

Cardinality constraints may appear anywhere in a model and are evaluated whenever a Constraint type is evaluated (until we have individually established their validity). Here we say simply that the values assigned automatically to `help` within `foo`-prefixed members of the array `boo1` must be such that more than one of them has the value one. As the engine will assign values to variables in ascending order from their ranges, and backtracks the last-assigned value first, the proposition will be satisfied once the engine assigns `boo1:foo2:help` to one having previously backtracked and assigned `boo1:foo1:help` to one and having assigned `boo1:foo0:help` to zero which is not backtracked. The output confirms as much:

```
boo0 extends  {
...snip...
}
boo1 extends DATA  {
  foo0 extends DATA  {
    help 0;
    sfIndex 0;
    sfTag "foo0";
  }
  foo1 extends DATA  {
    help 1;
    sfIndex 1;
    sfTag "foo1";
  }
  foo2 extends DATA  {
    help 1;
    sfIndex 2;
    sfTag "foo2";
  }
}
```

6 Dynamic Policies and Lazy Constraints

Constraints are typically evaluated at parse-time; but there is no reason why they can not be evaluated also at run-time, as *lazy constraints*. We include a `LazyConstraint` function type for convenience.

```
eval extends LazyConstraint {
      cpu LAZY si_observed:cpu;
      mem LAZY si_observed:mem;
      net LAZY si_observed:net;
      cpu_alloc LAZY si_observed:cpu_alloc;
      mem_alloc LAZY si_observed:mem_alloc;
      net_alloc LAZY si_observed:net_alloc;
      cpu_new VAR;
      mem_new VAR;
      net_new VAR;
      [sfConstraint] -- "compute_allocation(cpu, cpu_alloc, cpu_new,
                                            mem, mem_alloc, mem_new,
                                            net, net_alloc, net_new)";
}
```

Note the use of `LAZY` attribute values for references as appropriate. (This is necessary where we want the current value referred to by the `LAZY` reference to be sourced on constraint evaluation rather than having it resolved a priori at parse-time).

Moreover, we define the notion of a `DynamicPolicyEvaluation` (DPE) type which is only evaluated at run-time in being designated as lazy. A DPE may cause arbitrary changes to a model every time it is evaluated subject to evaluation guards being satisfied.

The form of DPE is:

```
foo extends DynamicPolicyEvaluation {
   guard lazy boolean expression;

    … arbitrary function evaluation …

   effects extends ApplyEffects{
       Effects statements
   }
}
```

When a DPE is evaluated, the `guard` lazy boolean expression is evaluated. This is a guard conditions which needs to evaluate to true for the DPE to be evaluated any further. If the guard condition holds, then some arbitrary function evaluation may occur. These will typically be functions such as `LazyConstraint`s. The results of evaluating the functions are assigned to the attributes specifying them. The results may then be used in the subsequent effects statements, specifed by the `effects` attribute. This attribute must be a data description containing either: `path`, `key`, `update` attributes, *or* in turn a number of data descriptions each containing: `path`, `key`, `update` attributes. A `path` attribute gives the target description (or Prim) of the effect update, a `key` attribute gives the name of the attribute to be updated, and `update` gives the new value for this attribute.

Note that constraints themselves (lazy or not) may also make modifications to the model. So, we have DPEs and constraint types which can do this. Some of the differences between them are not immediately obvious; the approach will be clarified by a changed approach in a future release.

A constraint may refer to a constraint variable which has previously been registered with the constraint engine. When the constraint is evaluated, it may instantiate/assign the `VAR` to some value (which may be fully-instantiated, i.e. is an integer, enumerated, or some other literal, or partially-instantiated, i.e. is some term such as a list containing new `VAR`s). This instantiation will be written back to the original point of declaration of the `VAR`. Thus, the evaluation of constraints may have side-effects which reach outside the `Constraint` types being evaluated. However, it is not possible to reassign the value of an attribute within a constraint, it is only possible to instantiate variables. That is, attributes referred to within a constraint are treated by the constraint as immutable entities, once fully instantiated.

The effects terms within DPEs allow for the values of attributes to be arbitrarily changed. That is, attributes are treated as mutable entities by DPEs. The distinction described between constraints and DPEs represents a potential source of confusion, and is far from desirable. As stated, this will be resolved.

An example:

```
handlestarvation extends DynamicPolicyEvaluation {
   guard (LAZY foo:up && LAZY foo:cpu);

   eval extends LazyConstraint {
      cpu LAZY foo:cpu;
      cpu alloc foo:cpu alloc;
      cpu_new VAR;
      [sfConstraint] -- "compute_allocation(cpu, cpu_alloc, cpu_new)";
   }

   effects extends DATA {
      changeCPU extends DATA {
         path LAZY foo;
         key "cpu_alloc";
         update LAZY eval:cpu_new;
      }

      changeCPUFlag extends DATA {
         path LAZY foo;
         key "cpu";
         update false;
      }
   }
}
```

In the example, the DPE is only evaluated if the guard is satisfied. That is, if `foo:up` is true as is `foo:cpu`. The example DPE assigns new values to attributes which have already been assigned, i.e., `foo:pu_alloc`, and `foo:cpu`. As stated, constraints themselves do not support mutable attributes.

7 Sudoko Example

This example is a simple is a 9x9 Su Doku puzzle. Su Doku is abbreviation of the Japanese: *Suuji wa dokushin ni kagiru* (数字は独身に限る), which roughly translates as *the digits are limited to one occurrence*. In Su Doku, there is a grid of squares. In the 9x9 version, the grid is further partitioned into 9 3x3 squares, as shown in Figure 2. We have nine sets of symbols (here, numbers) which we use to label the squares of the grid. We have a set of nine ones, a set of nine twos, a set of nine threes, and so on up to a set of nine nines.

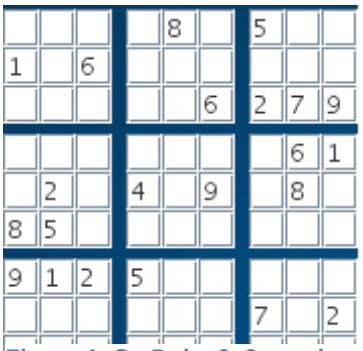


Figure 1: Su Doku 9x9 puzzle

The constraints here are simple. There are 27 in all, that is nine lots of three constraints: for *i* in {0, ..., 8}, no symbol may appear more than once in the i^{th} row, column or 3x3 square. In Figure 3, we show the squares involved in the set of three constraints for the 0^{th} row, column, and 3x3 square.

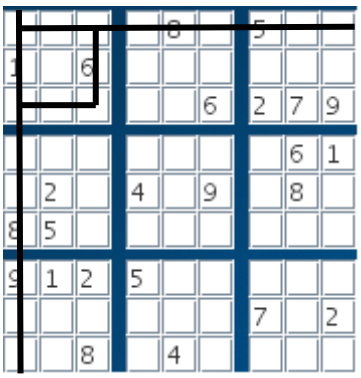


Figure 2: Squares involved in constraints for 0th-indexed row, column, and 3x3 square

Su Doku problems will have some pre-labelled squares, otherwise it would be trivial to construct a solution. Strictly speaking, Su Doku problems should be specified with initial labellings which engender unique solutions. Clearly, it is possible that some initial grids will have no solution or many solutions, even if they satisfy the described constraints for Su Doku.

The SmartFrog model that we have implemented to solve 9x9 Su Doku problems is as follows.

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

SuDoku9 extends {

   puzzle TBD;

   Board extends Array {
      prefix "square";
      extent [9,9];
      generator extends Constraint, ArrayGenerator {
          val extends Var {
             range "[1..9]";
             auto LAZY PrintSolution;
          }
      }
    }

   PreValues extends Constraint {
      [sfSource] puzzleArray LAZY Board;
      [sfPrefix] memberPrefix "square";
      [sfPath]   pathToSquare LAZY val;
      [sfUpdate] aggregatedSquares puzzle;
   }

   Constraints extends Array {
      prefix "cons";
      extent 9;
```

```
      generator extends Constraint, ArrayGenerator {
         [sfContext] index sfIndex;  //local sfIndex
         [sfSource]  puzzleArray LAZY Board;
         [sfPrefix]  memberPrefix "square";
         [sfPath]    pathToSquare LAZY val;
         [sfPred]    predToChooseVerticalSlice (index == LAZY sfIndex0);
         [sfUpdate]  verticalSlice;
         [sfPred]    predToChooseHorizontalSlice (index == LAZY sfIndex1);
         [sfUpdate]  horizontalSlice;
         [sfPred]    predToChooseMiniSquare
                (index == (((LAZY sfIndex1 / 3)*3) + (LAZY sfIndex0 /
3)));
         [sfUpdate]  miniSquare;
         [sfConstraint] -- "alldifferent(horizontalSlice),
                           alldifferent(verticalSlice),
                           alldifferent(miniSquare)";
      }
   }

   PrintSolution extends PrettyPrint {
      [sfSource] puzzleArray LAZY Board;
      [sfPrefix] memberPrefix "square";
      [sfPath]   pathToSquare LAZY val;
      [sfUpdate] output;
   }

   WriteBack extends PolicyEvaluation {
      effects extends {
         puzzle PrintSolution:output;
      }
   }
}

sfConfig extends SuDoku9 {
   puzzle [
            [[4,0],8], [[6,0],5],
            [[0,1],1], [[2,1],6],
            [[5,2],6], [[6,2],2], [[7,2],7], [[8,2],9],
            [[7,3],6], [[8,3],1],
            [[1,4],2], [[3,4],4], [[5,4],9], [[7,4],8],
            [[0,5],8], [[1,5],5],
            [[0,6],9], [[1,6],1], [[2,6],2], [[3,6],5],
            [[6,7],7], [[8,7],2],
            [[2,8],8], [[4,8],4]
   ];
}
```

A SmartFrog [18] model is a hierarchy of attribute sets called prototypes having the distinguished attribute `sfConfig` as its root. Each attribute may have a value which is a literal (String, Integer, etc), extends a prototype, or which is a reference to an attribute defined elsewhere. In the example, `sfConfig` extends `Sudoku9` which means that it takes on all of the attributes of `Sudoku9`, overwriting the puzzle attribute.

A model is converted from its raw form to an internal representation in a parsing stage. A phase of parsing is to perform link resolution which resolves attribute references (by ultimately copying values) including evaluating functions. A couple of function types of

particular interest in this example are `Constraint` and `Array` as can be seen in the model above.

We start by defining the board. This is part of the *declare* aspect of declare, constrain, search.

```
Board extends Array {
   prefix "square";
   extent [9,9];
   generator extends Constraint, ArrayGenerator {
       val extends Var {
          range "[1..9]";
          auto LAZY Label;
       }
   }
 }
```

The `Board` attribute (which `sfConfig` contains) is a 9x9 array, as denoted by the `extent` attribute. An array type will add array members to some prototype (which defaults to the Array's prototype). The members have names which are prefixed with the String value denoted by the `prefix` attribute, here "square". This array type will thus add members: `square_0_0`, `square_0_1`, `...`, `square_1_0`, `square_1_1`, `...`, `square_9_9` to the Array prototype. The `generator` attribute indicates the prototype or function type to be used for array members. In this case, it is a `Constraint` function type. When evaluated in link resolution, each instance of this Constraint function will register a `val` variable having an inclusive Integer range of `1..9` with the constraint solver. Observe that the local variable declaration is made by extending the type `Var`. In doing so, we specify an `auto` attribute meaning that the variable will be automatically assigned a value from its range (where the assignment is subject to backtracking); and this assignment will occur once link resolution has subsequently reached the attribute `PrintSolution` (which link resolution determines is an attribute of `sfConfig`).

We also specify a number of pre-specified values for the Su Doku problem, as follows.

```
PreValues extends Constraint {
   [sfSource] puzzleArray LAZY Board;
   [sfPrefix] memberPrefix "square";
   [sfPath]   pathToSquare LAZY val;
   [sfUpdate] aggregatedSquares puzzle;
}
```

This is an aggregating `Constraint` type, which means that it takes values of attributes from various members of an array and aggregates them into a list for processing as part of the constraint evaluation[1]. The `sfSource`-tagged attribute gives the array source

[1] Note that for the various pre-specified prototypes supported by SmartFrog's constraint extensions, we use a combination of distinguished attribute names and distinguished attribute tags. Typically, any one constraint prototype will use either a set of distinguished attribute names, such as `Array` which uses prefix, extent, and

for aggregation. The `sfPrefix`-tagged attribute gives the String prefix for array members. The `sfPath`-tagged attribute gives the path into array members of the common attribute whose values are to be aggregated. In this case, the common attribute is the `val` attribute described previously. The values of this attribute are taken from each square of the 9x9 board, and assigned to the `sfUpdate`-tagged attribute, here `squares`.

An added twist is the following. If the `sfUpdate`-tagged attribute already has a value, then we attempt a form of unification between the aggregated values and the attribute's value. The `squares` attribute refers to the value of `puzzle`, which is defined in `sfConfig`. Its value will be added to the model prior to parsing, as a result of a user pressing the "solve" button. On this happening, the GUI logic writes the values of the squares that have been given a value to the `puzzle` attribute.

The form of the value of the puzzle attribute is: `[ [loc, val], ...]`. The aggregated values from the `val` attributes will take the form `[val, ...]`. The unification step looks at the `loc` values in the value of the `puzzle` attribute and reconciles them with the array members associated with the aggregated values. That is, for any `[loc, val]` pair, we look at the aggregated value, `val'`, got from the array member denoted by `loc`, and attempt a unification on `val` and `val'`. If they unify (e.g., a variable unifies with any literal to yield the literal), then the result of unification replaces the aggregated value for the array member in question.

We have an aggregated list of constraint variables corresponding to the `val` attribute from members of the Board array, i.e. `[VAR1, VAR2, VAR3, ...]`. Associated with this list are implicit locations, viz. `[ [[0,0], VAR1], [[0,1], VAR2], [[0,2],VAR3], ...]`. The attribute puzzle will be a list of board values, corresponding to the pre-specified values, e.g. `[ [[4,0], 8], [[6,0], 5], ...]`. For locations, which are common to both lists, the pre-specifed values get unified with the variables aggregated in the list of constraint variables.

Finally, this unification is submitted to the constraint solver. There is no constraint string specified – when this is the case, the attributes are still submitted to the constraint solver so that any unification that may have taken place is accounted for.

In the second aspect (constrain), we specify a number of constraints for values assigned to the `val` attribute.

```
Constraints extends Array {
   prefix "cons";
   extent 9;
   generator extends Constraint, ArrayGenerator {
      [sfContext] index sfIndex;  //local sfIndex
      [sfSource]  puzzleArray LAZY Board;
      [sfPrefix]  memberPrefix "square";
      [sfPath]    pathToSquare LAZY val;
      [sfPred]    predToChooseVerticalSlice (index == LAZY sfIndex0);
      [sfUpdate]  verticalSlice;
```

---

generator, or a set of distinguished attribute tags, which are more commonly used within extensions of `Constraint`. Sometimes, a combination is allowed. We have tried to adopt the most natural convention for each constraint type.

```
        [sfPred]    predToChooseHorizontalSlice (index == LAZY sfIndex1);
        [sfUpdate]  horizontalSlice;
        [sfPred]    predToChooseMiniSquare
                (index == (((LAZY sfIndex1 / 3)*3) + (LAZY sfIndex0 /
3)));
        [sfUpdate]  miniSquare;
        [sfConstraint] -- "alldifferent(horizontalSlice),
                           alldifferent(verticalSlice),
                           alldifferent(miniSquare)";
    }
  }
```

In all there are 27 constraints, three for each index *i* between zero and eight (inclusively). For each set of three, one constraint pertains to the i^{th} row (`horizontalSlice`), another to the i^{th} column (`verticalSlice`), and a third to the i^{th} 3x3 square (`miniSquare`). Each constraint says that the squares in each column, row, 3x3 square, must be assigned `alldifferent` values.

In more detail, we declare an array of extent nine. Each member is a `Constraint` type which aggregates the variables declared for the `Board` array (in the first part) into three lists: `horizontalSlice`, `verticalSlice`, and `miniSquare`. The `sfSource`-tagged attribute specifies the array from which to aggregate values for processing by the `Constraint` type. The `sfPrefix`-tagged attribute gives the common prefix of the array members of interest. We are interested in `square`-prefixed members of the `Board` array. The `sfPath`-tagged attribute specifies the path into array members of the attribute of interest. We are interested in `val`. The three `sfPred`-tagged attributes serve to constrain which occurrences of `val` get aggregated, namely those which satisfy the proposition given by the `sfPred`s. The aggregated values are assigned to the subsequent `sfUpdate`-tagged attribute.

`sfContext`-tagged attributes provide additional attribute values to use in `sfPred`-tagged propositions. `sfPred`-tagged propositions are evaluated in the context of the candidate array members, whose values may or may not be selected for an aggregation. When evaluating a proposition against an array member, each `sfContext`-tagged attribute has its (key, value) written at the top-level into the array member. It is then possible for these values to be used in evaluating a proposition. Here, `index` is written into array members of `Board` with the value `sfIndex` which is the index of the array member specifying the constraint type in the `Constraints` array. In the `sfPred`-tagged propositions, we refer to `sfIndex0` which is the index of the candidate `Board` array member in the 0^{th} dimension, and `sfIndex1` the index of the candidate Boards array member in the 1^{st} dimension.

The first proposition says that we aggregate `val` for those members of `Board` whose 0^{th} (or horizontal) index is the same as the index into `Constraints`. The aggregation is assigned to `verticalSlice`. For the next one, it is the 1st (or vertical) index that is of interest; the aggregated values are assigned to `horizontalSlice`. For the last proposition, we are interested in those members of `Board` whose square is in the `indexth` 3x3 square of the board.

The final aspect of the Su Doku example (search) gets the constraint engine to label those squares that are not necessarily entailed through the union of all of the `alldifferent`

constraints. For these squares, assignment to a value from a square's range may be followed by backtracking if the assignment leads to a dead-end. The specified labelling is brute force in nature – we do not use any heuristics which bias variables or values to try first. For this example, such heuristics would not make a significant difference to the complexity of solving puzzles. The constraint engine works effectively to minimise the amount the amount of search and labelling necessary. It propagates the effects of assignments (including the assignments of squares for pre-specified values) immediately through the ranges of variables, meaning that variables ranges and thus labelling options quickly become contracted. Labelling occurs by virtue of declaring the variables to-be-labelled as automatic variables. Recall, we instructed this to occur when link resolution reaches the `PrintSolution` attribute, which occurs next in the model.

In the course of solving the Su Doku puzzle, the constraint engine will have prescribed the writing of square values back to the `val` attributes within `Board` array members. However, the result of solving the puzzle will not have been written back to the `puzzle` attribute. This would be desirable in order to easily extract the result.

```
PrintSolution extends PrettyPrint {
   [sfSource] puzzleArray LAZY Board;
   [sfPrefix] memberPrefix "square";
   [sfPath]   pathToSquare LAZY val;
   [sfUpdate] output;
}
```

`PrintSolution` extends `PrettyPrint` which is a function type which aggregates values in the same way that an aggregated constraint does, but instead of aggregating the values as a list of the form: `[val, val, ...]`, it aggregates them as a list of the form: `[[loc, val], [loc, val], ...]`, which is an appropriate form to write back to the `puzzle` attribute, for which we wish to include location information. An example would be `[[[0,0],2], [[1,0], 3], ...]`.

Finally, we need to write this list back to the puzzle attribute. As far as constraints are concerned, attributes are immutable in the sense that once they have been assigned a value they may not be assigned again. However, we define the `PolicyEvaluation` function type whose purpose is to evaluate fragments of policy and which may have side-effects on the model. `PolicyEvaluation` types treat attributes as mutable entities.

```
WriteBack extends PolicyEvaluation {
   effects extends {
      puzzle PrintSolution:output;
   }
}
```

The side-effect that we specify here is to copy the output from `PrintSolution` to the `puzzle` attribute. Finally, we specify some pre-specified values:

```
sfConfig extends SuDoku9 {
   puzzle [
             [[4,0],8], [[6,0],5],
             [[0,1],1], [[2,1],6],
             [[5,2],6], [[6,2],2], [[7,2],7], [[8,2],9],
             [[7,3],6], [[8,3],1],
             [[1,4],2], [[3,4],4], [[5,4],9], [[7,4],8],
             [[0,5],8], [[1,5],5],
             [[0,6],9], [[1,6],1], [[2,6],2], [[3,6],5],
             [[6,7],7], [[8,7],2],
             [[2,8],8], [[4,8],4]
   ];
}
```

As we have said, constraint evaluation is done as part of parsing a SmartFrog model. The command `sfParse -v suExample.sf` yields assignment to squares, viz.

```
puzzle [|
[|[|0, 0|], 2|], [|[|1, 0|], 3|], [|[|2, 0|], 7|], [|[|3, 0|], 9|],[|[|4,
0|], 8|], [|[|5, 0|], 4|], [|[|6, 0|], 5|], [|[|7, 0|], 1|], [|[|8, 0|],
6|], [|[|0, 1|], 1|], [|[|1, 1|], 9|], [|[|2, 1|], 6|], [|[|3, 1|], 7|],
[|[|4, 1|], 2|], [|[|5, 1|], 5|], [|[|6, 1|], 8|], [|[|7, 1|], 3|], [|[|8,
1|], 4|], [|[|0, 2|], 4|], [|[|1, 2|], 8|], [|[|2, 2|], 5|], [|[|3, 2|],
3|], [|[|4, 2|], 1|], [|[|5, 2|], 6|], [|[|6, 2|], 2|], [|[|7, 2|], 7|],
[|[|8, 2|], 9|], [|[|0, 3|], 7|], [|[|1, 3|], 4|], [|[|2, 3|], 3|], [|[|3,
3|], 2|], [|[|4, 3|], 5|], [|[|5, 3|], 8|], [|[|6, 3|], 9|], [|[|7, 3|],
6|], [|[|8, 3|], 1|], [|[|0, 4|], 6|], [|[|1, 4|], 2|], [|[|2, 4|], 1|],
[|[|3, 4|], 4|], [|[|4, 4|], 7|], [|[|5, 4|], 9|], [|[|6, 4|], 3|], [|[|7,
4|], 8|], [|[|8, 4|], 5|], [|[|0, 5|], 8|], [|[|1, 5|], 5|], [|[|2, 5|],
9|], [|[|3, 5|], 1|], [|[|4, 5|], 6|], [|[|5, 5|], 3|], [|[|6, 5|], 4|],
[|[|7, 5|], 2|], [|[|8, 5|], 7|], [|[|0, 6|], 9|], [|[|1, 6|], 1|], [|[|2,
6|], 2|], [|[|3, 6|], 5|], [|[|4, 6|], 3|], [|[|5, 6|], 7|], [|[|6, 6|],
6|], [|[|7, 6|], 4|], [|[|8, 6|], 8|], [|[|0, 7|], 3|], [|[|1, 7|], 6|],
[|[|2, 7|], 4|], [|[|3, 7|], 8|], [|[|4, 7|], 9|], [|[|5, 7|], 1|], [|[|6,
7|], 7|], [|[|7, 7|], 5|], [|[|8, 7|], 2|], [|[|0, 8|], 5|], [|[|1, 8|],
7|], [|[|2, 8|], 8|], [|[|3, 8|], 6|], [|[|4, 8|], 4|],[|[|5, 8|], 2|],
[|[|6, 8|], 1|], [|[|7, 8|], 9|], [|[|8, 8|], 3|]|]
```

For this particular example, the following output (taken from part of the parse output) shows us that the constraint engine had to perform 119 labelling assignments before a solution was identified. As this is more than the number of squares, there was certainly some backtracking involved.

```
  Label counts:119
```

The result of constraint solving for this example can also be seen in some screenshots taken of our SmartFrog-based Su Doku solver application, which is also available on the web (http://wiki.smartfrog.org). See Figure 4.

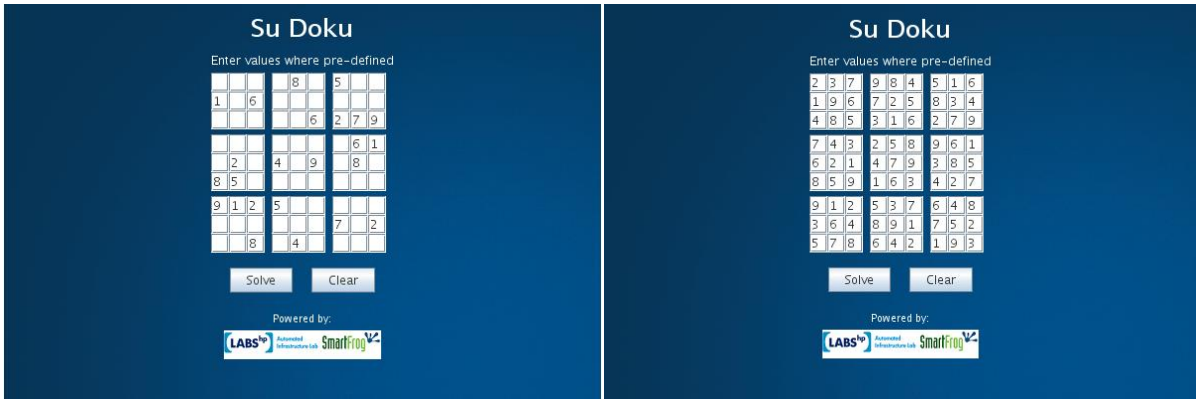


Figure 3: Screeshots from Su Doku Solver, Deployed on Web

8 Frequently Asked Questions

Here are some (at least conceived) frequently asked questions

8.1 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.

8.2 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.

8.3 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).

8.4 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.

9 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.