The SmartFrog Constraint Extensions

For SmartFrog Version 3.11

Localized for UK English / A4 Paper

Table of Contents

Introduction

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

This document covers the general model for how constraints may be used within the SmartFrog language. It documents how plug-ins may be written and loaded, and describes the use of the default plug-in, which uses the constraint satisfaction engine Eclipse (*).

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

(*) Eclipse: http://eclipse.crosscoreop.com/index.html.

Constraints in SmartFrog

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

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

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

1 Variables:

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

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

The semantics of VAR need some explaining.

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

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

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

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

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

Bearing this in mind, consider the following example:

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

```
Example2 extends Example {
    b a;
}

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

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

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

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

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

2 Constraints:

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

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

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

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

```
#cons:nnn#...\#
```

where the “...” indicates the text that will be passed to the plug-in, and nnn indicates any positive integer. This integer is intended for use as an instruction to the solvers regarding the priority of the constraint with respect to others. It is assumed, unless otherwise stated, that the *lower* numbers are solved earlier. Note that the last # is escaped – this allows for #s used within the annotation not to be escaped. This convention may be changed, as appropriate, in future.

To make it easier to write constraints, some pre-defined values are given and are associated with specific keywords. So:

```
#do:m#...\#          is equivalent to
#cons:n#...\#        where n=m+j+1, where j is the highest index
                    attached to a "cons" annotation in the original
                    description
```

Also:

```
#decl#...\#          is equivalent to
```

```
#cons:0#...\#

#cons#...\#          is equivalent to
#cons:1#...\#

#do#...#          is equivalent to
#do:0#...\#
```

Use of #decl# is meant for declaring the range of prolog logical variables, which often would be done early on in constraint solving. Hence, it is prescribed the highest priority. Of course, declarations may be made at any time – just attach an appropriate priority to a #cons# annotation.

Use of #cons# is meant for the specification of further constraints on the range of variables, such as alldifferent([X,Y,Z]), which says that the values of each of the logical variables X,Y,Z must be pair-wise distinct.

Use of #do# is meant for general prolog goals, including search goals which guide the selection process of values for constraint variables.

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

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

Consider the example (using some abstract constraint language)

```
AllDifferent extends {
    a VAR;
    b VAR;
    c VAR;
} #cons# alldifferent([a,b,c]) \#

Ordered extends AllDifferent #cons# a<b, b<c \#
```

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

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

Plug-ins

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

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

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

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

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

During execution of the parser, an additional phase is included in the standard set of phases (between function and assertion) which invokes the plug-in. The plug-in is selected by setting the java property

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

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

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

1. No constraints are solved, any annotations are ignored

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

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

Currently there are two plug-ins implemented, one that uses a vanilla prolog engine, and one that uses Eclipse, which is essentially a variant of prolog augmented with constraint satisfaction. We are seeking only to continue to support the second of these – as the capability of this solver subsumes that of the first.

The Eclipse Plug-in

Eclipse (http://eclipse.crosscoreop.com/index.html) is a Prolog-based constraint satisfaction engine, available under a Mozilla Public Licence. The engine is maintained currently by Cisco Systems.

The Eclipse Plug-in for SmartFrog is implemented as a single class, which has the following genealogy.

```
java.lang.Object
  org.smartfrog.sfcore.languages.csf.constraints.CoreSolver
    org.smartfrog.sfcore.languages.csf.constraints.PrologSolver
      org.smartfrog.sfcore.languages.csf.plugins.EclipseSolver
```

CoreSolver contains logic for instantiating the solver specified for use (see **Plug-ins**). PrologSolver is an abstract class, which may be used as a base class for prolog-based solvers. It implements the method:

```
void solve(CSFComponentDescription cd)
```

which is the sole method of the

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

interface, which any CoreSolver must implement.

PrologSolver also collects the constraint annotations and amalgamates them into a single constraint goal, to be solved by the Eclipse engine; and, gets the results of the various variable bindings, made by the given prolog engine, replacing them appropriately within component descriptions.

EclipseSolver handles the Eclipse-specific aspects of the constraint solving process. We now proceed to describe the basic features of our support for constraints with Eclipse, for SmartFrog, and give a flavour of how things work under-the-hood.

Important information concerning source hierarchy – For now, EclipseSolver lives in trunk/core/components/constraints/... Currently, you can build EclipseSolver using the eclipse.jar jar file provided in the lib directory. For performing constraint solving on .csf descriptions, the Eclipse distribution will need to be downloaded. Shortly, EclipseSolver will be migrated into the SmartFrog core, subject to licensing issues that need to be clarified.

1 Constraint Aggregation

The constraint annotations attached to component descriptions in the sfConfig hierarchy are collected together, and sorted in order of their respective priorities. The annotations are then presented to the Eclipse engine as a single aggregated goal.

2 Compilation and Namespacing

Eclipse prolog sources may be compiled as a part of the base theory of an Eclipse engine instance, using one of these goals (as part of a constraint annotation):

```
source(FN)
source(FN,NS)
```

When either of these goals is reached in executing the aggregated constraint goal (see **Constraint Aggregation**), the named source (FN) is compiled. The predicates defined therein are thus available for the remaining lifetime of the engine instance, unless explicitly retracted. Note that the specification of the source to be compiled (FN), and the namespace (NS), are both strings.

FN is a relative name; and, the default path used in resolving it is the current working directory. Ideally, the following property will have been previously set, and, if so, it is this that determines the path with which FN should be resolved.

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

This can be set on the command-line with the Java -D option, or in the default.ini file used by the default SmartFrog scripts.

Note that Eclipse expects the path to be specified using forward obliques (/). Also, under Windows, the path “C:\” is written “//C/”.

For source/2 (the 2-arity version of source), the NS parameter is a namespace, used for the compiled source. Each predicate defined within the compiled source will be renamed. The string given by NS will be prefixed to the front of the name of each such predicate, together with a double underscore.

Note:

1. All user-defined predicates should be compiled with a namespace. If they are not, there is the possibility of predicate name clashes.

2. No user-defined predicate should explicitly make use of a double underscore

When referring to a predicate, whose definition has been compiled in this way, the fully-qualified name should be used. Say we define the predicate 'member', and compile the file containing it according to the namespace 'foo', then all references to the predicate should be 'foo__member'. Another example of renaming predicates, by namespace prefixing, is provided later.

Namespacing is useful to distinguish predicates defined in different prolog source files, which have common names.

3 State Scoping

State scoping is useful for scoping clausal state which may be asserted in the course of Eclipse goal execution. Two key predicates are sos/0 and eos/0. When 'sos' is processed, as part of executing the aggregated constraint goal, a state level counter is increased. When 'eos' is processed, the state level counter is decreased. If any clausal state is asserted in between, it is asserted to “exist at” the current state level. When 'eos' is processed, all “scoped” clausal state, asserted for the current level, is retracted.

Scoped clausal state may be asserted using the following predicates:

```
assert_state(C) – As Eclipse's assert/1, except asserts C to current
                  state level
asserta_state(C) – As Eclipse's asserta/1, except ...
assertz_state(C) – As Eclipse's asserta/1, except ...
```

```
replace_state_level(C) – Retracts all scoped state at the current state
                         level which matches the functor and arity of C,
                         and then asserts C to the current state level
```

```
retract_all_state – Retracts all scoped clausal state, ie that asserted
                    using assert_state/asserta_state/assertz_state
retract_all_state(C)  - Retracts all scoped clausal state matching C
retract_all_state_level(C) -  Retracts all scoped clausal state
                              matching C for current state level
retract_all_state_level -  Retracts all scoped clausal state for
                           level
retract_state(C) – Retract a piece of clausal state matching C. Tries
                   with current scope level first, and works back
                   through previous levels until find matching.
                   Fails if no match.
```

```
clause_state(C) – Unifies C with matching clausal scoped state. Tries
                  with current scope level first, and works back
                  through previous levels until find matching.
                  Fails if no match.

clause_state_level(C) – Unifies C with matching clausal scope state for
                        current state level. Fails if no match.
```

An example is the following.

```
foo extends {}#do# assert_state(p(1)),
                   assert_state(p(2)),
               assert_state(q(1)),
               sos,
               assert_state(q(2)),
               assert_state(q(3)),
               retract_state(q(_)),
               retract_state(p(_)),
               eos\#
```

The (aggregated) constraint is evaluated as scope level 0 to begin with. p(1), p(2), q(1) are asserted to scope level 0. q(2), q(3) are asserted to scope level 1. retract_state(q(_)) will retract q(2) – it retracts in the closest scope. retract_state(p(_)) will retract p(1).

Finally, eos will retract remaining scope state for level 1, so q(3) will also be retracted. Just p(2) remains asserted.

Further notes:

- sos and eos calls are inserted at the start and end (respectively) of constraint annotations, which means that constraint annotations effect self-contained islands of clausal state.
- When sourcing an Eclipse source, with namespace renaming, the following renaming of predicates will also take place:

assert/1 asserta/1 assertz/1	assert_state/1 asserta_state/1 assertz_state/1
retract/1 retract_all/1	retract_state/1 retract_all_state/1
clause/1	clause_state/1

- In order to access the original predicates, in the event that a source is namespaced-renamed, the following predicates may be used; where, the predicates in the first column are renamed to those in the second:

ecl_assert/1 ecl_asserta/1 ecl_assertz/1	assert/1 asserta/1 assertz/1

ecl_retract/1 ecl_retract_all/1	retract/1 retract_all/1
ecl_clause/1	clause/1

4 Getting and setting attribute state

It is not necessary to use VAR declarations for attributes within component descriptions, unless they are required for link referencing, as the attribute list is not treated as being static by the constraint implementation; the set of attributes of a description may be added to as constraint evaluation progresses.

Firstly, we need to make some introductory points:

- The logical variables used within any individual constraint annotation are local, viz. scoped, to that annotation. That is, variables sharing the same name in different annotations refer to different logical variables. In the implementation of the Eclipse plug-in, a preprocessor is used to rename variables to assure distinctness.

- A number of predicates are made available for getting and setting state. The single-argument versions of these predicates are: sfset/1, sfget/1 and sfattr/1. In all of them, the single argument is the name of the attribute being set/got (in the context of the pertinent component description).

  Aside: Attribute names may be unprefixed, local names; or, they may have a reference part prefixed to them. In the latter case, they must be written as strings. Examples are:

  - Local names: attr, attr2, “attr”, “attr2”...
  - Prefixed names: “foo:attr”, “foo:foo2:attr2”, ...

  A preprocessor will rewrite the use of these single-argument predicates as three-argument predicates, the first argument being a handle to the component description, the second argument being the attribute name (as above), and the third argument being a logical variable.

  In the case of sfget/1, this is the variable to which the value of the attribute is bound. In the case of sfset/1, this is the already-bound variable from which the value to be set is extracted. **Importantly,** whenever the same attribute is used in occurrences of sfget/1, sfset/1, sfattr/1 in a single constraint annotation, the implicit logical variable in each occurrence will be the **same** logical variable. More details will be provided on this as we go along.

- When the same attribute is referred to in different constraint annotations, possibly attached to different component descriptions, there is wiring that exists 'under-the-hood' to connect their use. If we use any of the attribute manipulation predicates, described in this section, the following applies. We use the following example for discussion.

```
ListElement extends {
}#do:1# member(sfattr(element), sfget(list))\#

System extends {
   x extends ListElement { list theList; }
   y extends ListElement { list theList; }
   z extends ListElement { list theList; }
}#cons:1# lib(sd),
          &::(sfget("x:element"), sfget(theList)),
          &::(sfget("y:element"), sfget(theList)),
          &::(sfget("z:element"), sfget(theList)),
```

```
         alldifferent(sfget("x:element"), sfget("y:element"),
                                          sfget("z:element"))\#
sfConfig extends System {
   theList ["one", VAR, "three", VAR];
   aList [VAR, "two", VAR, "four"];
}#cons# sfattr(theList)=sfattr(aList)\#
```

Note, as will be described properly below, the use of sfattr/1 is equivalent to doing an sfget/1 (at the start of the annotation), where the value obtained is bound (if set) to a logical variable; followed by doing an sfset/1 (at the end of the annotation) where the value that has been bound (if not previously bound) *to the same logical variable*, in the course of executing the annotation, is written back in the pertaining description for the given attribute.

In the example, once the preprocessor has done its job, the annotation using the alldifferent/1 predicate will look as follows (as indicated in bold font):

```
sfConfig extends System {
   x extends{
      list ["one", VAR/1, "three", VAR/2];
   }#do:1# sos, sfget(Cxt, element, E), sfget(Cxt, list, L),
           member(E, L), sfset(Cxt, element, E), eos\#

   y extends{
      list ["one", VAR/1, "three", VAR/2];
   }#do:1# sos, sfget(Cxt, element, E), sfget(Cxt, list, L),
           member(E, L), sfset(Cxt, element, E), eos\#

   z extends{
      list ["one", VAR/1, "three", VAR/2];
   }#do:1# sos, sfget(Cxt, element, E), sfget(Cxt, list, L),
           member(E, L), sfset(Cxt, element, E), eos\#

   theList ["one", VAR/1, "three", VAR/2];
   aList [VAR/3, "two", VAR/4, "four"];

}#cons# sos, sfget(Cxt, theList, L), sfget(Cxt, aList, A), L=A,
        sfset(Cxt, theList, L), sfset(Cxt, aList, A), eos\#
 #cons:1#sos, lib(sd),
        sfget(Cxt, "x:element", X), sfget(Cxt, "y:element", Y),
        sfget(Cxt, "z:element", Z), sfget(Cxt, theList, L),
         &::(X,L), &::(Y,L), &::(Z,L), alldifferent([X,Y,Z]),
        eos\#
```

Note also the transformation in the ListElements, say for x, from:

```
x extends{
    list ["one", VAR/1, "three", VAR/2];
}#do:1# member(sfattr(element), sfget(list))\#
```

To:

```
x extends{
    list ["one", VAR/1, "three", VAR/2];
}#do:1# sos, sfget(Cxt, element, E), sfget(Cxt, list, L),
        member(E, L), sfset(Cxt, element, E), eos\#
```

Notably, the wiring that is made 'under-the-hood' unifies the occurrence of E in the annotation for description x, with X in the second annotation for sfConfig. This means that when we reach member(E, L), E is constrained to have the range [“one”, “two”, “three”, “four”]. Moreover, for the other ListElements, propagation of the alldifferent/1 constraint means that, when choosing a value for “y:element”, its range will be [“two”, “three”, “four”] (assuming that “one” was bound to "x:element"), and that, when choosing a value for

"z:element", its range will be [“three”, “four”] (assuming that “two” was bound to "y:element").

To set attributes, the sfset/3 predicate may be used. The predicate takes a context – pertaining to the component description to which the given constraint annotation is attached – as its first argument. It also takes the attribute to be set as the second argument, and the intended value as the third.

```
sfset(Cxt, Attr, Value) – Sets Attr to have Value in Cxt
```

If the attribute does not already exist, then the attribute is added to the component description. If the attribute does already exist, then the old value and the new value must be unifiable; that is, the new value must be the same value as the old (modulo variable renaming), or must be more specific. For example, an attribute may have the value: [1,2,VAR]. Setting it to have the value [1,2,3] would be permitted, but attempting to set it to be [2,1,3] would fail.

Note that in atoms that appear in a constraint annotation, use of the three-argument version of this predicate is redundant. Instead, the following one- and two-argument versions should be used.

```
sfset(Attr, Value) – Sets Attr to have Value
sfset(Attr) – Sets Attr. Replaced by common logical variable, plus an
              occurrence of sfset/3 (if not already present)
```

These versions of sfset (/1, /2) may be used only in constraint annotations, attached to component descriptions, and not in predicate definitions provided in auxiliary theory files. For this latter purpose, the three-argument version must be used.

The said preprocessor will insert the component description Cxt into uses of sfset/1 and sfset/2. The preprocessing of sfset/1 is interesting. It is replaced by a logical variable, which, as previously noted, is the same as that used for all other occurrences of sfset/1, sfget/1 and sfattr/1 used in a constraint annotation. The generated occurrence of sfset/3 is moved to the end of the constraint, but if such an occurrence already exists then it is discarded.

A simple example is that of the “allocate/2” goal (which comes from a use-case concerned with allocating virtual machines to hosts). For its second argument, allocate (see embolded fragment) takes a logical variable, which will be bound to the allocation result. (Note, allocate is sourced as in the 'foo' namespace, so we use foo__allocate to refer to it.)

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

sfConfig extends {
  Test1 extends {
    producers [["h0",[4]],["h1",[3]],["h2",[3]]];
    consumers [["v0",[3]], ["v1",[3]], ["v2",[2]], ["v3",[2]]];
  }#do:1#foo__allocate(sfcxt,sfset(ass))\#
}#do#source("alloc.ecl","foo")\#
```

After preprocessing the annotation becomes:

```
#do:1#sos, foo__allocate(Cxt, Ass), sfset(Cxt, ass, Ass), eos\#
```

That is, the occurrence of sfset/1 is replaced by a logical variable Ass, and a copy of sfset/3 is tagged to the end of the annotation.

Note, as an important aside, the use of sfcxt. Whenever sfcxt is used in a constraint annotation, the preprocessor replaces it with a handle to the actual component description to which the annotation is attached. This is essential when authoring predicates which are stored in auxiliary theory files. For example, the definition of allocate/2 (in part) is:

```
allocate(Cxt, Ass):-
        sfget(Cxt, producers, P),
        writeln(P), flush(stdout),
        (var(P) ->
            writeln("Must be producers in an allocation problem"),
            flush(stdout),
            fail;
            preprocess_producers(P)), ...
```

Here, allocate/2 uses the Cxt supplied as an input argument in all uses of sfget/3 and sfset/3, as can be seen:

```
        sfget(Cxt, producers, P),
```

In predicates defined in auxiliary theory files, the only attribute manipulation predicates that may be used are sfget/3 and sfset/3.

To get the values of attributes, as already alluded to, use:

```
sfget(Cxt, Attr, Value) – Gets value of Attr in Cxt, and binds to
                          Value. Value is left unbound, if no such
                          value.
sfget(Attr, Value) – Gets value of Attr and binds Value. Value is ...
sfget(Attr) – Gets Attr. Replaced by common logical variable, plus an
              occurrence of sfget/3 (if not already present)
```

An occurrence of the one-argument version, sfget/1, is replaced by a logical variable (common to all occurrences of sfget/1, sfset/1, sfattr/1 for that attribute in the annotation); a three-argument version is inserted at the start of the annotation, if not already there for the pertaining attribute.

An example, from above, is:

```
#cons:1# use(sd),
         &::(sfget("x:element"), sfget(theList)),
         &::(sfget("y:element"), sfget(theList)),
         &::(sfget("z:element"), sfget(theList)),
          alldifferent(sfget("x:element"), sfget("y:element"),
                                           sfget("z:element"))\#
```

which becomes:

```
 #cons:1# sos, use(sd),
        sfget(Cxt, "x:element", X), sfget(Cxt, "y:element", Y),
        sfget(Cxt, "z:element", Z), sfget(Cxt, theList, L),
        &::(X,L), &::(Y,L), &::(Z,L), alldifferent([X,Y,Z]),
        eos\#
```

Finally, there is a predicate sfattr/1, which is also replaced by a logical variable common to all occurrences of sfget/1, sfset/1, sfattr/1, for the named attribute, in the constraint annotation. The preprocessor also generates an sfget/3, and inserts it at the start of the annotation, if not already present for the attribute; and generates an occurrence of sfset/3, and inserts it at the end of the annotation, if not already present for the attribute.

An example is:

```
#do:1# member(sfattr(element), sfget(list))\#
```

which becomes:

```
#do:1# sos, sfget(Cxt, element, E), sfget(Cxt, list, L),
          member(E, L), sfset(Cxt, element, E), eos\#
```