Support for Constraints in SmartFrog

Andrew Farrell, Patrick Goldsack
Automated Infrastructure Laboratory, HP Labs, Bristol, UK.
firstname.lastname@hp.com

Abstract

All IT systems need to be managed to some degree. It would be advantageous to automate such management as much as possible, particularly when the complexity of systems makes the management task difficult for human operators. In this report, we describe work on an initial offering of "Support for Constraints in SmartFrog".

SmartFrog is a model-based framework which manages the deployment lifecycle of (possibly distributed) groups of Java objects. By augmenting SmartFrog with a facility for constraints specification, we facilitate the specification of incomplete models which may then be completed and validated by the tool. This would remove some of the burden in writing models and have the additional benefit of enabling models to be reusable in different contexts.

It is worth noting that the work is still experimental in that we have not yet arrived at a definitive set of constraint extensions to SmartFrog. However, it is of value to report on progress in order that we might receive feedback from people using the offering so that it can be improved as appropriate. In the concluding section of this report, we set our offering in the wider context of our work which is to provide an approach to autonomic configuration management of systems.

Keywords: SmartFrog, Constraints, Configuration Management

Introduction

All IT systems need to be managed to some degree. It would be advantageous to automate such management as much as possible, particularly when the complexity of systems makes the management task difficult for human operators. In the last decade or so, a number of tools for *configuration management* have appeared; the common intent of which has been to ease the systems management task.

One such tool is SmartFrog – an open-source technology developed within the Automated Infrastructure Laboratory of HP Labs, Bristol, UK. SmartFrog is a model-based framework which manages the deployment lifecycle of (possibly distributed) groups of Java objects. As we describe in the *Related Work* section, SmartFrog is quite different from most tools in the configuration management space.

Generally speaking, specifying models for system configuration can be labourious (even if tooled) and is subject to human error. One possible remedy would be to facilitate the specification of incomplete models with constraints, such that models are completed and validated by the tool. This would remove some of the burden in writing models and have the additional benefit of enabling models to be reusable in different contexts. Different parameterisations of such models, that is, different sets of values specified for certain model attributes, would cause constraints specified therein to be evaluated in different ways, leading to differently-completed models.

We propose an initial offering of *support for constraints in SmartFrog*. The work is still experimental in that we have not yet arrived at a definitive set of constraint extensions to SmartFrog. However, it is of value to report on progress in order that we might receive feedback from people using the offering so that it can be appropriately improved.

We have based the constraints extensions to SmartFrog on requirements sourced from studies of approaches mainly in the field of configuration management. Many of these are documented in the Related Work section. We have also talked to groups within HP Labs about their modelling requirements, as well as understanding the need for constraints-based evaluation of models in our wider work. We briefly touch on this latter point in the concluding section of this report.

The rest of this technical report is organised as follows. In the following section, we set the context of the report with a discussion of wider related work. We then present our offering of "support for constraints in SmartFrog". We finish with a brief concluding discussion.

Related Work

Configuration management is the process of tracking and controlling the behaviour of networked machines, be they physical or virtual. It encompasses everything that is necessary to take a collection of networked machines, with no software, and convert this "bare metal" hardware into a single functioning system, according to a given specification.

Ideally, a specification for system configuration should be written in a high-level, declarative language. An example requirement that might be specified could be "configure sufficient web servers to guarantee a response time of X". Such a requirement may be derived directly from a Service-level Agreement signed with a particular customer for provision of a web service. Then, the specification would be compiled automatically into low-level configurations for individual machines. Fault-tolerance should also be accommodated in that (clusters of) machines should as much as possible self-adapt in response to changes in requirements.

However, contemporary approaches to configuration management require the specification of requirements at a much lower-level of abstraction and thus manual translation of high-level requirements is necessary. Such approaches are typically variations on a single theme: to facilitate the specification of desired system state in some declarative model and for the machines of a system to configure themselves according to this model. Some examples of configuration management tools that fit this description in varying hues are: LCFG, Quattor, CfEngine, and BCFG. For illustration, we shall look at the first of these in more detail.

LCFG [1] (Local Configuration System) is a configuration management framework, developed by the University of Edinburgh, for managing large numbers of Unix-based workstations. A central repository holds a number of source files which describe various aspects of configuration such as "parameters specific to student machines", or "parameters specific to Redhat machines". These source files are compiled to individual profiles for specific machines, according to the designation of machines for particular purposes – such as "to be used as a web server". When a profile changes, the corresponding client is prompted by a UDP notification to retrieve the updated profile. Component scripts on the client are responsible for processing the configuration parameters in a profile and taking the necessary actions to migrate the client to the desired state represented by the profile.

Puppet [5] is another configuration management tool. It aims to raise the level of abstraction of language used in specifying desired state, compared with tools such as LCFG. It achieves this through the definition of a resource abstraction layer, which is designed to hide platform-specific details such as the files in which particular pieces of configuration information is stored.

Configuration management, as described here, is often positioned as part of a larger process, when considering the management of IT infrastructures in enterprises, namely, that of IT Governance. One currently popular methodology which can be used towards effective IT Governance is the IT Infrastructure Library (ITIL) [6]. ITIL was by the United Kingdom government and has become the de facto framework for IT Service Management. Asset and configuration management, coupled with change management, are recognised as core and interdependent processes by ITIL. A principal goal of ITIL configuration management is to maintain a comprehensive logical representation of the IT environment, which is called the Configuration Management Database (CMDB).

Client Automation Enterprise using RADIA technology [7] is the current HP solution to automated change and configuration management of machines in an enterprise IT context. The state of the IT environment is maintained in the CMDB. Dynamic policies govern desired states for machines, and changes are made to the configurations of clients accordingly. These changes, orchestrated through ITIL-based change management, are reflected in the CMDB. Clients have Application Manager installed on them which allows a server to deploy mandatory applications and data to them according to application management profiles which specify their intended use, e.g. web server, application server, and so on. Application Manager handles the discovery, deployment, configuration, repair, update, and removal of applications and data from clients as prescribed by policy.

None of these technologies cover quite the same space as SmartFrog. In the majority of these systems there is little or no cross-client coordination and they treat each client as an independent entity. In that sense they are weaker than SmartFrog with its ability to coordinate and configure across a range of nodes and carry out autonomic actions. SmartFrog makes it easy for components to perform peer-to-peer interactions with each other according to a number of mechanisms built into SmartFrog:

- Service Location Protocols, which allow SmartFrog components to automatically discover each other without the need for any explicit configuration to link them.
- Partition Membership Protocols, which allow a number of SmartFrog components to declare themselves as part of a group and have the framework automatically elect a leader within the group. This is typically used to provide fail-over for critical services such as DHCP. A leader will run a configuration policy engine which will determine which servers in a group, say, should run a DHCP daemon. If one of these fails, then the leader can reassign the daemon to another member of the group.

However within the space where client-server based tools such as LCFG are targeted, largely in desktop and device management, they do provide much more value out-of-the-box along with a simpler more direct configuration model. Moreover, there is a complementarity between these tools and SmartFrog, as evidenced by the joint initiative between SmartFrog and LCFG [8]. As motivated in [8], as infrastructures get larger and more complex, some degree of autonomous reconfiguration is essential, so that individual nodes (and clusters) can make small adjustments to their configuration in response to their environment, without the overhead of feeding changes back via a central repository; an example being to elect a replacement for a failed DHCP server amongst themselves. As well as fault-tolerance, autonomous reconfiguration could be applied to load-balancing.

As described previously, in current configuration management tools there is a gap between the level of abstraction at which IT administrators would ideally wish to work, such as "configure sufficient machines as web servers to satisfy demand", and that supported by current configuration management tools. There are some recent research contributions [9,10,11] that have sought to close this gap. They allow the specification of rules or constraints which map abstract specifications of requirements into specific allocations of machines to roles, which can then be enacted by tools such as LCFG and CfEngine. SmartFrog already obviates the need to explicitly specify some aspects of configuration, such as the roles that individual machines play, through the partition membership protocols described previously. However, this only removes the need to specify a fairly limited subset of configuration aspects, when compared with the sorts of abstract descriptions that can be expressed using these constraints-based approaches. For SmartFrog, we retain the use of membership protocols for this semi-autonomous client-side aspect of configuration but look to support other more abstract methods of configuration specification which are in keeping with the described constraint-based languages. We recommend the use of the support for constraints in SmartFrog, described in this report, for this purpose. We give additional insight into the support provided by Cauldron/Quartermaster for constraints-annotated modelling in the following section.

In common with these other constraint-based approaches to resolving abstract configuration descriptions, SmartFrog does not include any ontological support for describing IT artefacts. A fundamental reason for this is that there are a number of initiatives, such as CIM and SML, which have already sort to address this point. We do not see the need to replicate this work. Something that is missing, and which could prove useful to provide at some stage is a translation mechanism which converts CIM [12] and SML [13] models to SmartFrog descriptions (including constraints, to account for SML's Schematron [14] rules which are used to constrain the values of attributes within models). Moreover, configuration management tools such as LCFG provide some vocabulary for more primitive aspects of machine configuration. As we see the purpose of SmartFrog as being a tool that is used in a complementary sense with tools such as LCFG, we would defer to these other tools to provide such an ontological context. The question is then of integration, which can be resolved on a per-case basis.

Support for Constraints in SmartFrog

We propose an initial offering of support for constraints in SmartFrog. It is based on requirements sourced from studying approaches mainly in the field of configuration management, some of which are reviewed in the previous section. Examples include: Cauldron/Quartermaster [10,15], as well as various approaches to constraint-based configuration management of IT infrastructure [9,10,11]. We have also informed our work by talking to groups within HP Labs, and are now in a position to road-test our offering. This will be done through a combination of testing it in the course of our work and getting others to use SF's constraints support.

It is worth setting out what we are *not* doing. We are not implementing our own constraints solver – we are primarily interested in identifying useful concepts to support, not in implementation. Instead, we intend to leverage existing one(s), but have not as yet committed to the use of any. Which solver(s) we use at any time depends on conclusions we draw with respect to useful concepts to support, and how these are best supported. We currently favour the ECLIPSE Constraint Logic Programming (CLP) solver [16].

Moreover, we are not overwhelmingly concerned with the efficiency of solving currently, although we do attempt to implement things as efficiently as possible. Our greater concern is arriving at a language for constraint specification in SmartFrog, and getting it right. Finally, we are currently just addressing the case of solving constraint problems (that is, generating completions of constraint-annotated SmartFrog models) rather than finding optimal solutions according to some optimality criteria. The latter would be optimisation, which is something we may address in time but we choose to leave for the time being.

Generally, a constraint problem is characterised by: a set of domain variables that need to be assigned, and a set of constraints over the assignments. [17] provides an informative overview of the applicability of linear or integer programming and CLP-based constraint solving to constraint problems thus characterised.

Integer programming is well-suited to problems whose constraints naturally fit into the equational mould – a set of domain variables are inter-related in ways that are naturally captured as a set of equations over these variables. However, they are particularly inefficient (in terms of the number of domain variables required for the representation of problems) when this is not the case. There are many classes of problems for which constraints over the "natural" decision variables of the domain cannot be represented in this way. CLP-based approaches have the advantage for these classes of problems of providing support for the specification of constraints directly over these decision variables.

An example is the classic *n*-queens problem. (We characterise a similar example to this later using SmartFrog – Su Doku). In the *n*-queens problem, there are *n* queens and an *n*x*n* grid (see Figure 1, for *n*=8). All queens have to be placed on the board in distinct squares, and no two queens can occupy the same diagonal, row or column. In a CLP-based characterisation of this problem, we would have *n* domain variables, one for each queen and row. The range of the domain variable would be [1..*n*] giving the column occupied by the queen. The constraints are simply specified thus:

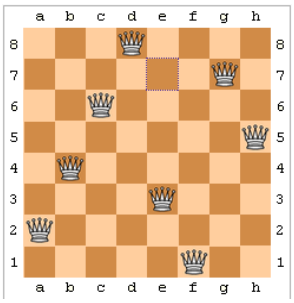

Figure 1: n-Queens board (n=8)

```
for all queens i, for all queens j
(value(i) == value(j) OR value(i) + i == value(j) +j OR
                         value(i) - i == value(j) - j)
IMPLIES i=j
```

That is, if the column value for any two queens is the same OR if the row + column value for any two queens is the same (meaning that they lie on a common top-right to bottom-left diagonal) OR if the row – column value for any two queens is the same (a top-left to bottom-right diagonal) then the queens in question must be one and the same.

If we were to characterise this problem as a regular integer programming problem, we would need a binary *decision* variable for each square (i.e. *n*x*n* variables), over which would be specified linear formulations capturing the appropriate constraints. As documented in [17], these constraints are less natural than in the CLP account.

As a general principle, we are interested in employing a constraint programming approach which as much as possible captures problems in a natural way. The types of problems that we anticipate will be typically characterised by constraints which do not naturally fit into the equational model of integer programming. Moreover, the approach that we would consider to be the best fit for the current semantics of SmartFrog is one which supports progressive instantiation of a constraint problem with backtracking. CLP-based approaches are well-suited to this requirement, unlike those based on integer programming.

In CLP-based constraint solving, we follow a *declare, constrain, search* pattern for expressing problems, viz.

- *Declare the domain variables and their ranges*
- *Constrain their possible assignments*
- *Prescribe search strategies which bias variable assignments*

In the following section, we present a simple example which follows this pattern.

Simple Su Doku Example

Our first example is a simple 9x9 Su Doku puzzle. (This is not an example relating to configuration management, unless your management problem is based on Su Doko!) 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 nine 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.

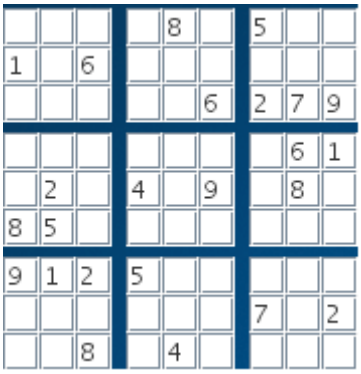

Figure 2: Su Doku 9x9 puzzle

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.

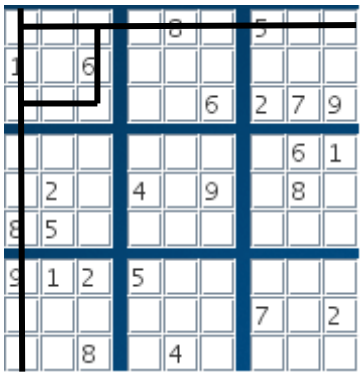

Figure 3: Squares involved in constraints for 0th-indexed row, column, and 3x3 square

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 Label;
          }
      }
    }

   PreValues extends Constraint {
      array LAZY Board;
      prefix "square";
      path   LAZY val;
      update extends AggregateSpecifier {
         unify puzzle;
      }
   }

   Constraints extends Array {
      prefix "cons";
      extent 9;
      generator extends Constraint, ArrayGenerator {
         context extends DATA { index sfIndex; }  //local sfIndex
         array LAZY Board;
         prefix "square";
         path LAZY val;
         ver extends AggregateSpecifier { pred (index == LAZY sfIndex0); }
         hor extends AggregateSpecifier { pred (index == LAZY sfIndex1); }
         sq extends AggregateSpecifier {
            pred (index == (((LAZY sfIndex1 / 3)*3)  + (LAZY sfIndex0 / 3)));
         }
         [sfConstraint] -- "alldifferent(hor),
                            alldifferent(ver), alldifferent(sq)";
      }
   }

   Label extends LabellingPoint;

   PrintSolution extends PrettyPrint {
      array LAZY Board;
      prefix "square";
      path LAZY val;
      output extends AggregateSpecifier;
   }

   WriteBack extends ApplyEffects {
      key "puzzle";
      update 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), which 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.

A `Constraint` type declares *local variables* to be filled in as the constraint is evaluated. These are declared using the `VAR` keyword. The general form of `Constraint` is as follows:

```
foo extends Constraint {

   bar1 VAR range;
   ... barn VAR range;

   ... other attributes ...

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

The `bar1,…,barn` attributes declare local constraint variables along with range information. The `pred1,…,predn` attributes specify constraint strings that prescribe constraints over values of these local variables, and specify search strategy (by virtue of binding variables in a particular order).

There are variations on the presented syntax that are also supported. A local variable may also be defined as an extension of one of the prototypes: `Var`, `AutoVar`, `UserVar`, or `DefaultVar`. We can see an example of this in the presented model, which we shall now explain one piece at a time. (See [19] also for more information.) 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 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` type 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 `Label` (which is an attribute of `sfConfig`).

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

```
PreValues extends Constraint {
   array LAZY Board;
   prefix "square";
   path   LAZY val;
   update extends AggregateSpecifier {
      unify 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. The `array` attribute gives the array source for aggregation. The `prefix` attribute gives the String prefix for array members. The `path` 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 array members for each square of the 9x9 board, and assigned to the `update` attribute. The `update` attribute starts off as a prototype extending `AggregateSpecifier`. This pre-defined prototype may be extended with assignments to attributes `pred` and/or `unify`. The first of these, `pred`, specifies a proposition that guards which values get aggregated. The second of these, `unify`, specifies a value with which we should attempt a form of unification with the aggregated values. In the model excerpt, the attribute refers to the value of `puzzle`, which is defined in `sfConfig`.

In more detail, 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 {
      context extends DATA { index sfIndex; }  //local sfIndex
      array LAZY Board;
      prefix "square";
      path LAZY val;
      ver extends AggregateSpecifier { pred (index == LAZY sfIndex0); }
      hor extends AggregateSpecifier { pred (index == LAZY sfIndex1); }
      sq extends AggregateSpecifier {
         pred (index == (((LAZY sfIndex1 / 3)*3)  + (LAZY sfIndex0 / 3)));
      }
      [sfConstraint] -- "alldifferent(hor),
                         alldifferent(ver), alldifferent(sq)";
   }
}
```

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 (`hor`), another to the i^{th} column (`ver`), and a third to the i^{th} 3x3 square (`sq`). The constraint string prescribes that the squares in each column, row, and 3x3 square, 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 previously for the `Board` array into three lists: `hor`, `ver`, and `sq`. The `array` attribute specifies the `Board` array from which to aggregate values for processing by the `Constraint` type. The `prefix` attribute gives the common prefix of the array members of interest. We are interested in `square`-prefixed members of the `Board` array. The `path` attribute specifies the path into array members of the attribute of interest. We are interested in `val`. The three `AggregateSpecifier` prototype attributes serve to constrain which occurrences of `val` get aggregated, namely those which satisfy the proposition given by the `pred` attributes. The aggregated values are assigned to the `hor`, `ver`, and `sq` attributes.

The `context` prototype serves to provide additional attribute values to use in evaluating the `pred`-specified propositions. These 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 attribute specified in the `context` prototype 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 `pred` propositions, we refer to `sfIndex0` which is the index of the candidate `Board` array member in the 0^{th} dimension, and `sfIndex1` which is the index of the candidate `Board` 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 `ver`. For the next one, it is the 1^{st} (or vertical) index that is of interest; the aggregated values are assigned to `hor`. For the last proposition, we are interested in those members of `Board` whose square is in the `index`th 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 `Board` square variables to be automatic. Recall, we instructed this to occur when link resolution reaches the `Label` attribute, which occurs next in the model. Notice that it extends a special type: `LabellingPoint`, which has been specifically provided for specifying anchor points for labelling automatic variables.

```
Label extends LabellingPoint;
```

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 {
   array LAZY Board;
   prefix "square";
   path LAZY val;
   output extends AggregateSpecifier;
}
```

`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 ApplyEffects {
   key "puzzle";
   update 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 (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 [20]. See Figure 4.

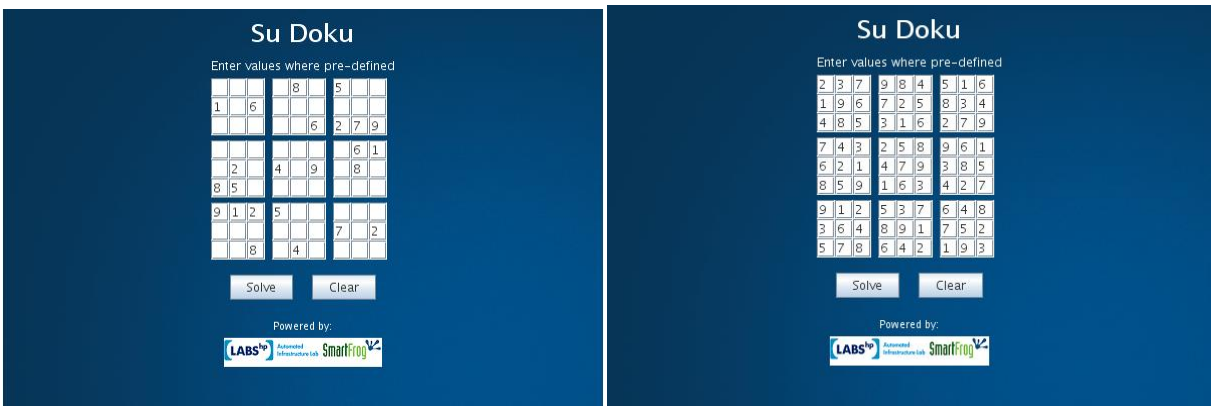

Figure 4: Screeshots from Su Doku Solver, Deployed on Web

Detailed Account of Constraints Support

Let us consider in some greater detail the support for constraints in SmartFrog. From our studies, we have so far determined the need for:

- Integer, boolean and enumerated-type domain variables. For simplicity, we choose not to provide support for real or rational domain variables for the time being. We will revisit this issue at some point. An enumerated type is characterised by the members of its range being explicitly enumerated. `["fred", "bob", "sue"]` is an example of an enumerated type.
- Cardinality constraints over arrays of objects of same type, for example:
  - All values of `foo:bar` in array members have different values
  - More than one instance of `foo:bar` has the value `1`
- Constraints evaluated individually for each member of such an array. For example:
  - If my index in array is `<5` then set my `foo:num` to `10` else to `20`
- Boolean, arithmetic, and enumerated-type constraints which include specific arbitrary attribute values as domain variables:
  - Boolean constraints, using: `nt`, `or`, `and`, `if`, `iff`. For instance, for boolean domain variables, `X` and `Y`, the constraint string `(X or Y), X=false` yields `Y=true` (through constraint propagation)
  - Integer constraints using: `eq`, `neq`, `lt`, `lte`, `gt`, `gte`. For instance, for integer domain variables, `X` and `Y` with ranges `[1..3]`, the constraint string `X*Y gt 8` yields `X=3, Y=3` (through constraint propagation)
  - Enumerated-type equality and disequality constraints using: `eq`, `neq`. For instance, for enumerated-type domain variables, `X` and `Y` with ranges `["fred", "bob"]`, the constraint string `X neq Y, X=fred` yields `Y=bob` (through constraint propagation)
- Sub-type constraints, for example: `foo` must be a sub-type of `bar`
- Association constraints, for instance in the allocation of VMs to hosts, the host specified in a VM's record must have the VM in its list of hosted VMs; and *vice versa*.
- Default values for variables which do not get assigned during the course of constraint evaluation.
- User-defined variables which require user input to provide assignments.
- Automatic variables which are automatically assigned values from their ranges, instead of requiring assignments to be coded into models as part of constraint strings. We have seen an illustration of automatic variables already in the Su Doko example.

The initial offering supports all of these requirements, albeit some are more road-tested than others. For cardinality constraints, we support a number of function types used to express them over arrays, 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 `p0` is satisfied in the evaluation context then so is `p1`. (If `p0` is not satisfied in the evaluation context then the proposition is satisfied irrespective of the value of `p1`).
- `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 {
        tag "x";
        array LAZY boo1;
        prefix "foo";
        card 1;
        pred extends Atomic {
                  pred (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 their validity is individually established). 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";
  }
}
```

A principal feature of Cauldron/Quartermaster [10,15] is the ability to specify *forall*- and *exists*-type cardinality constraints, which are called `gand` (for 'group and') and `gor` (for 'group or'). Also provided is `gadd`, for 'group add'. Using these, it is possible to specify association and 'all different' constraints for instance. We present some examples, comparing Cauldron/Quartermaster and SmartFrog.

- For *a group of servers, the cost of each server is constrained to be less than $1750*. For both Cauldron/Quartermaster and SmartFrog model fragments shown, there is a constraint specified for each server which prescribes its maximum cost, derived from the sum of application and computer costs.

  Cauldron/Quartermaster:

```
Server {
   cost: float;
   comp: Computer;
   app: Application;
   satisfy cost == comp.purchaseCost + app.licenceFee;
   satisfy cost <= 1750;
}
```

  SmartFrog with constraints:

```
Server extends Constraint {
   cost VAR INTEGER;
   comp extends Computer;
   app extends Application;

   purchaseCost comp:purchaseCost;
   licenceFee app:licenceFee;

   [sfConstraint] -- "cost lt 1750,
                      cost = purchaseCost + licenceFee";
}
```

  This constraint could also be written using the `ForAll` cardinality constraint construct in SmartFrog, and using a `gand` (which effects a 'for all') in Cauldron.

- *The sum cost of the servers is less than $10000*.

  Cauldron/Quartermaster:

```
satisfy gadd(I, serv, serv[i].cost <= 10000)
```

  SmartFrog with constraints:

```
ServersCost extends Constraint {
   array LAZY Servers;
   prefix "server";
   path LAZY cost;
   costlist extends AggregateSpecifier{}
   [sfConstraint] -- "sum(costlist, Cost), Cost lt 10000";
}
```

  In the foregoing, we write attribute specifiers to aggregate the `cost` attribute from members of an array of servers. We see from the previous model fragment that this attribute is declared as a `VAR`, and it is these `VAR`s that are aggregated. The `sfSource`-tagged attribute

gives the source array, `Servers`; the `sfPrefix`-tagged attribute gives the prefix of array members, `server`; the `sfPath`-tagged attribute gives the path into an array member of the attribute value to be aggregated, which is simply the `cost` attribute; and the `sfUpdate`-tagged attribute gives the attribute, `costlist`, to which to assign the result of aggregation.

Note that `sum` is an *active* constraint, which means that it may be specified *a priori*, and as assignments are made to individual cost attributes, the ramifications of the assignments are propagated through the range of the variable `Cost`. Effectively, the lower bound of the range of `Cost` will move up with every (positive) assignment. Note, however, that we also latterly limit the value that may be assigned to `Cost` by specifying an upper-bound ($10000). The net effect of the constraint string is that, as assignments are made to the aggregated cost attributes, the lower bound will move up. If it moves above the upper bound, then a violation of the upper bound is necessarily entailed; and constraint solving will backtrack. Backtracking thus happens as soon as the `Cost` upper bound can no longer be satisfied; it is not necessary to wait until all `cost` attributes have been assigned in order to figure this out.

- In the allocation of VMs to hosts, the host specified in a VM's record must have the VM in its list of hosted VMs; and *vice versa*.

Cauldron/Quartermaster:

In `Host`, the first `satisfy` constraint is an 'all different' constraint. The second constraint says that for each VM specified in the set of VMs of a host, the VM's host attribute must refer to the given host. In `VM`, the `satisfy` constraint says that the specified host must in turn specify the VM somewhere in its list of VMs.

```
Host {
  vms: (ref VM)[..n];
  satisfy gand(i, vms, gand(j, vms, vms[i] == vms[j] ==> i==j));
  satisfy gand(i, vms, vms[i].host == this);
}

VM {
  host: ref Host;
  satisfy gor(i, host.vms, host.vms[i] == this);
}
```

SmartFrog with constraints:

For our discussion, we choose to show a complete example model for SmartFrog. The part which enforces the association constraint between hosts and vms is shown in bold. The approach taken in Cauldron/Quartermaster is to declaratively prescribe the association constraint. Then, a theorem prover [15] is used to ensure that any candidate model satisfies the association constraints captured by the `satisfy` statements. In SmartFrog's support for constraints, we do not start with a candidate model; rather, we complete models incrementally. In this *modus operandi*, it is appropriate that once a variable is assigned (for instance) that any appropriate consistency enforcement (such as that prescribed by an association constraint) is explicitly instructed by the model. In this sense, it is closer to imperative enforcement than declarative enforcement.

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

Host extends ArrayGenerator {
   resources extends Constraint {
      host_type extends AutoVar {
       range ["ht0", "ht1"];
      }
      memory extends AutoVar {
       range [3, 4];
      }
      idx PARENT:sfIndex;

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

   caps [resources:memory];
   hostvms [];

}

VM extends Constraint, ArrayGenerator {
   resources extends Constraint {
      vm_type extends AutoVar {
       range ["vmt0", "vmt1"];
      }
      memory extends AutoVar {
       range [2, 3];
      }
      idx PARENT:sfIndex;

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

   reqs [resources:memory];

   allocated extends Var {
       postActions extends {

          -- extends DynamicPolicyEvaluation {
               host extends RawRef {
                     reference ("hosts" ++ ":" ++ sfVarAssignment);
               }

               appendToHostList extends LazyConstraint {
                     vm LAZY sfTag;
                     hostvms LAZY host:hostvms;
                     newhostvms VAR;
                     [sfConstraint] --
                              "appendlist(hostvms,[vm],newhostvms)";
               }
```

```
                effects extends ApplyEffects {
                      path LAZY host;
                      key "hostvms";
                      update LAZY appendToHostList:newhostvms;
                }
            }
        }
    }
}

sfConfig extends ResourceAllocator {
   providerList:array LAZY hosts;
   providerList:prefix "host";
   providerList:path LAZY sfTag;

   providerCapsList:array LAZY hosts;
   providerCapsList:prefix "host";
   providerCapsList:path LAZY caps;

   consumerList:array LAZY vms;
   consumerList:prefix "vm";
   consumerList:path LAZY sfTag;

   consumerReqsList:array LAZY vms;
   consumerReqsList:prefix "vm";
   consumerReqsList:path LAZY reqs;

   allocationList:array LAZY vms;
   allocationList:prefix "vm";
   allocationList:path LAZY allocated;

   hosts extends Array {
      prefix "host";
      extent 3;
      generator extends Host;
   }

   vms extends Array {
      prefix "vm";
      extent 4;
      generator extends VM;
   }
}
```

In the given model, we declare an array of `hosts` and an array of `vms`. Hosts have three or four GBs of memory. VMs have memory requirements of two or three GBs. Some constraint solving takes place for hosts and VMs to respectively determine their memory capabilities and requirements. For hosts, the following constraint strings prescribe `host0` to have four GBs of memory. `host1` and `host2`, on the other hand, are prescribed to have just three GBs each.

```
   idx eq 0 implies memory eq 4 and host_type eq ht0
   idx gt 0 implies memory eq 3 and host_type eq ht1
```

Similarly, for VMs, `vm0` and `vm1` are prescribed to have a memory requirement for two GBs each, and `vm2` and `vm3` to have a requirement for three GBs each.

The constraint type `ResourceAllocator` performs resource allocation; in this example of vms onto hosts. During the course of evaluating `ResourceAllocator` the SmartFrog parser will make assignments to the `allocated` constraint variable within VMs. Its definition is repeated here.

```
allocated extends Var {
    postActions extends {

        -- extends DynamicPolicyEvaluation {
             host extends RawRef {
                   reference ("hosts" ++ ":" ++ sfVarAssignment);
             }

             appendToHostList extends LazyConstraint {
                   vm LAZY sfTag;
                   hostvms LAZY host:hostvms;
                   newhostvms VAR;
                   [sfConstraint] --
                            "appendlist(hostvms,[vm],newhostvms)";
             }

             effects extends ApplyEffects {
                   path LAZY host;
                   key "hostvms";
                   update LAZY appendToHostList:newhostvms;
             }
        }
    }
}
```

The `allocated` prototype is treated as a constraint variable because it extends `Var`. In this case, we do not specify a range for this variable. It is not necessary to do so because the constraint solver will not reason over its range, rather the variable will simply get assigned by the resource allocation logic. The prototype does however specify some actions (`postActions`) to carry out when the variable gets assigned. In the presented 'actions' statement, we instruct that once a VM has been allocated to a host (that is, once a VM has had its `allocated` constraint variable assigned), the assignment should be reflected in the given host by inserting the name of the VM into the host's `hostvms` list.

Firstly, we get a reference to the host in question. We compose this on-the-fly by concatenating the string '`hosts`' with the *value* of `sfVarAssignement`. The attribute `sfVarAssignement` is inserted into an 'actions' statement when its pertaining constraint variable is assigned; and is given the value to which the variable has been assigned. Say, `allocated` is assigned the value `host0`. Then, `sfVarAssignement` will be assigned this value. In this case, the composed reference is: `LAZY hosts:host0`.

Following that, we specify a constraint (`appendToHostList`) whose constraint string inserts the vm name into the list of vms hosted by `host0`. The updated list is assigned to the local constraint variable: `newhostvms`. Finally, we write the newly composed list of vms back to `LAZY hosts:host0`. This is prescribed by the `effects` statement. The `path` attribute specifies the target of the write, which resolves to: `LAZY hosts:host0`. The attribute to be written is `hostvms`, as given by `key`. The value to be written is the previously computed `newhostvms`, as given by the `update` attribute.

OUTPUT FROM MODEL

As can be seen, in the following output with emboldened text for the relevant parts, the appropriate association constraint is enforced: each *vm*'s `allocated` host appears in the corresponding host's `hostvms` value, and each `hostvms` value is the smallest list satisfying this constraint.

```
...snip...
hosts extends DATA  {
  host0 extends  {
    resources extends DATA  {
      host_type "ht0";
      memory 4;
      idx 0;
    }
    caps [|4|];
    hostvms [|"vm0", "vm1"|];
    sfIndex 0;
    sfTag "host0";
  }
  host1 extends  {
    resources extends DATA  {
      host_type "ht1";
      memory 3;
      idx 1;
    }
    caps [|3|];
    hostvms [|"vm2"|];
    sfIndex 1;
    sfTag "host1";
  }
  host2 extends  {
    resources extends DATA  {
      host_type "ht1";
      memory 3;
      idx 2;
    }
    caps [|3|];
    hostvms [|"vm3"|];
    sfIndex 2;
    sfTag "host2";
  }
}
vms extends DATA  {
  vm0 extends DATA  {
    resources extends DATA  {
      vm_type "vmt0";
      memory 2;
      idx 0;
    }
    reqs [|2|];
    allocated "host0";
    sfIndex 0;
    sfTag "vm0";
  }
  vm1 extends DATA  {
```

```
    resources extends DATA  {
      vm_type "vmt0";
      memory 2;
      idx 1;
    }
    reqs [|2|];
    allocated "host0";
    sfIndex 1;
    sfTag "vm1";
  }
  vm2 extends DATA  {
    resources extends DATA  {
      vm_type "vmt1";
      memory 3;
      idx 2;
    }
    reqs [|3|];
    allocated "host1";
    sfIndex 2;
    sfTag "vm2";
  }
  vm3 extends DATA  {
    resources extends DATA  {
      vm_type "vmt1";
      memory 3;
      idx 3;
    }
    reqs [|3|];
    allocated "host2";
    sfIndex 3;
    sfTag "vm3";
  }
}
```

As a final example, consider that sometimes it would be desirable to allow users to direct the constraint solving process. That is, instead of specifying search heuristics, we would allow a user to specify how variables should be bound, subject to constraints, and subject to the possibility of backtracking. Consider the following model.

```
#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])";
   }
}
```

Attributes `x`, `y`, and `z` within `elements` are all declared as `user` variables, by virtue of extending `ListVar`, which in turn extends `UserVar`. The effect of such a declaration, for these variables, is that the user will be asked to enter values for them from their ranges (once all other constraint solving within the `Constraint` type, as prescribed by extant constraint strings has been effected). `ListVar` also prescribes a range for these constraint variables: `["one","two","three","four"]`. In the single constraint string, we constrain the assignments of `x`, `y`, and `z` to be mutually different, by virture of `alldifferent([x,y,z])`. Note that `alldifferent` is an *active constraint*, which means that as assignments are made to the domain variables named by the constraint, the ranges of the other yet-to-be-assigned variables shrink accordingly (to reflect the remaining values that may be used in assignment.) Observe from the following screenshots that as soon as we assign the value `one` to `x`, the ranges of `y` and `z` shrink accordingly.

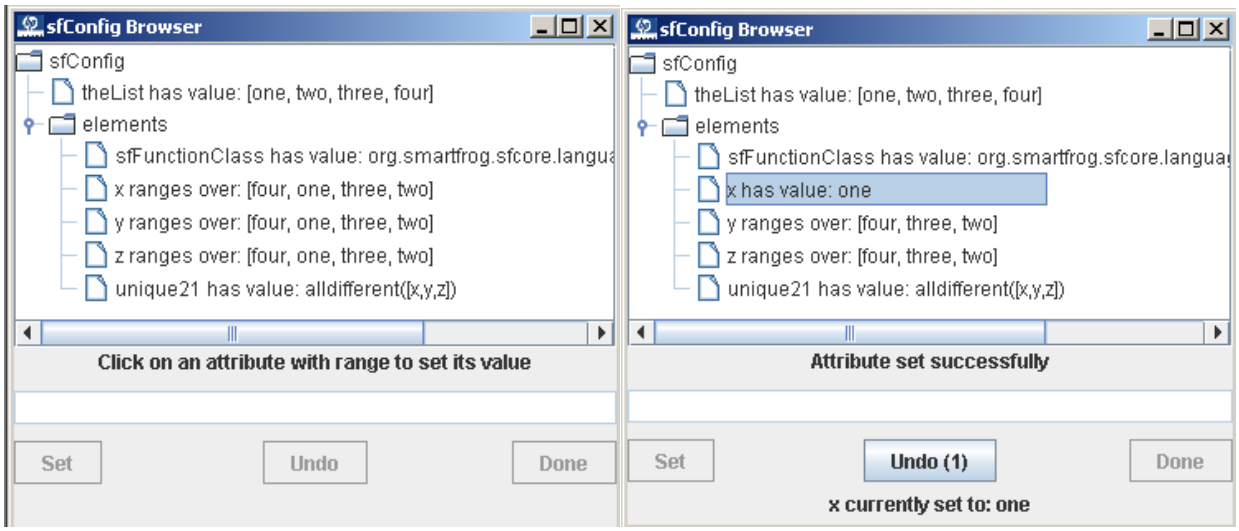

Figure 5: Screenshots from SmartFrog parsing of a Model with User Variables

Conclusions

We have presented a brief overview of an initial offering of 'support for constraints in SmartFrog'. We are motivated by the observation that writing models can be a labourious process and error-prone. To remediate this problem, at least to a degree, we propose support for constraints in SmartFrog so that incomplete models may be automatically completed and validated. We are testing our offering in the context of our wider work, and are making it available to others in order to get feedback which will be used to mature the offering as appropriate.

It is interesting to make a distinction between augmenting SmartFrog with constraints in the context of its rôle as a tool for configuration management, and proposing its augmentation as an option for general-purpose constraint solving. Cauldron/Quartermaster [10,15] makes a similar distinction – Cauldron being a general-purpose object-oriented constraint solver, and Quartermaster its incantation in a configuration management context. We could similarly propose constraints-augmented SmartFrog for the general-purpose formulation of constraints problems. That is, SmartFrog provides an expressive higher-level syntax for constraint specification compared with Constraint Logic Programming languages such as Eclipse, making available concepts such as template definitions and inheritance which help to ease the process of domain modelling. In the context of configuration management, we largely support all of the facilities provided by Quartermaster through our constraint extensions to SmartFrog.

It is notable that our work emphasis is somewhat different to that of Cauldron/Quartermaster. Our offering is part of a wider effort towards facilitating autonomic control, or self-management, in systems. To this end, the constraints-based support within SmartFrog is complementary to other SmartFrog-based work on orchestration. The two form a *hybrid* approach to autonomic configuration management, combining state-based orchestration with manipulation of underlying component topologies according to constraints-based policies. We have applied our hybrid approach in the implementation of a dynamically-provisioned web server architecture. State-based orchestration ensures that no requests are sent to web server instances by a load balancer component until they are up, whereas dynamically-evaluated policies prescribe actions to be taken in response to alarms. An alarm may indicate unacceptably high response delays, and an action to take in response may be to provision an additional server instance. Constraint solving may come into play in how the additional instance is provisioned, for instance. More details will be provided in a forthcoming report. We have also documented the approach as a submission to TechCon 2009 [21].

References

[1] Paul Anderson and Alastair Scobie. LCFG: The Next Generation. In UKUUG Winter Conference. UKUUG, 2002.

[2] Quattor – System Administration Toolsuite, at: http://quattor.web.cern.ch/quattor.

[3] N. Desai, A. Lusk, R. Bradshaw, and R. Evard. BCFG: A Configuration Management Tool for Heterogeneous Environments.
In CLUSTER '03: Proceedings of Fifth IEEE International Conference on Cluster Computing, pages 500–503, Dec. 2003.

[4] http://www.cfengine.org.

[5] Luke Kanies. PUPPET: Next generation Configuration Management. jLOGIN: Feb 2006. Available at:http://www.usenix.org/publications/login/2006-02/pdfs/kanies.pdf.

[6] Office of Government Commerce (OGC). The Official Introduction to the ITIL Service Lifecycle, ISBN:9780113310616, TSO, 2007.

[7] HP Client Automation Enterprise (based on RADIA technology), at: http://www.hp.com.

[8] Paul Anderson, Patrick Goldsack, and Jim Paterson. SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control.
In LISA '03: Proceedings of the 17th USENIX conference on System administration, pages 213–222, Berkeley, CA, USA, 2003.

[9] Thomas Delaet and Wouter Joosen. PoDIM: A Language for High-level Configuration Management.
In LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference, pages 1–13, Berkeley, CA, USA, 2007.

[10] A. Sahai, S. Singhal, R. Joshi, V. Machiraju. Automated Policy-based Resource Configuration in Utility Computing Environments. HP Labs Technical Report: HPL-2003-176.

[11] Sanjai Narain. Network Configuration Management via Model Finding.
In LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference, pages 155–168, Berkeley, CA, USA, 2005.

[12] Common Information Model (CIM) Standards, at: http://www.dmtf.org/standards/cim/.

[13] Service Modeling Language (SML) Working Group, at: http://www.w3.org/XML/SML/.

[14] Schematron: A language for making assertions about patterns in XML documents, at: http://www.schematron.com

[15] L. Ramshaw, A. Sahai, J. Saxe, S. Singhal. Cauldron: A Policy-based Design Tool. POLICY 2006, pages 113-122.

[16] Eclipse Constraint Logic Programming Solver, at: http://www.eclipse-clp.org/

[17] Hentenryck, P. V. 2002. Constraint and Integer Programming in OPL. INFORMS J. on Computing 14, 4 (Oct. 2002), 345-372.

[18] The SmartFrog Configuration Management Framework, SIGOPS Operating Systems Review special issue on "Survey of Work from Hewlett-Packard Labs", January 2009, to appear.

[19] The SmartFrog Constraint Extensions, at:
http://smartfrog.svn.sourceforge.net/viewvc/*checkout*/smartfrog/trunk/core/smartfrog/docs/csfExtensions.pdf

[20] Su Doku at SmartFrog wiki: http://wiki.smartfrog.org

[21] Andrew Farrell, Eric Deliot, Patrick Goldsack, Paul Murray, Eric Deliot, Alistair Coles. A Hybrid Approach to Autonomic Configuration Management. Submission to TechCon 2009.