Modelling System Administration Problems with CSPs

John A. Hewson and Paul Anderson

School of Informatics, University of Edinburgh, United Kingdom
`john.hewson@ed.ac.uk`, `dcspaul@ed.ac.uk`

Abstract System administrators increasingly use declarative, object-oriented languages to configure their systems. Introducing constraints to such a language and automatically generating a valid system configuration is an area of active research. We describe our work towards creating ConfSolve, an object-oriented configuration language which can describe constraints over valid configurations, solution of which is provided by compilation into a CSP. We evaluate our solution against a simple virtual machine allocation problem, with promising results for automating Infrastructure as a Service (IaaS) systems.

1 Introduction

Configuration of large computing installations is increasingly performed by automated tools, which make use of declarative, object-oriented languages [4,7]. These tools replace low-level shell scripts describing how to achieve a system state, with a high-level declarative model of the goal state of a system. Such tools have proven popular amongst system administrators configuring large sites, and are used to configure workstations, servers, and routing hardware. Infrastructure is increasingly virtualised, allowing more of it to be dynamically reconfigured to provision new services, reduce consumed resources, or respond to a hardware failure. Yet much of this work is done by hand, which is inefficient and error-prone.

Recently there has been interest in extending such tools to include variables which are not given an explicit value by a system administrator, but instead have constraints specified over them. Research into implementing basic versions of such systems has made use of constraints solved via SAT [8,5], the ECLiPSe [4] constraint logic programming system, and an adapted form of CSP [2].

There is a need for a general-purpose system configuration language which can be used to model a broad range of configuration problems involving constraints, in an easy-to-use manner. As the current state-of-the-art system configuration languages are object-oriented, we believe that any such configuration language must also be. It is desirable to compile constraints written in such a language into a lower-level description of the problem, for which CSP is a natural candidate.

We have developed a proof-of-concept system configuration language, in which constraints over valid solutions are described, and valid concrete configurations generated, via a CSP solver. Our goal is to create a language general enough to describe a large range of configuration problems, in a manner natural to a system administrator, and simpler than writing a model in a constraint programming language. This paper describes our initial work towards that goal.

2 The ConfSolve Compiler

The architecture of the ConfSolve compilation process is shown in figure 1. First, a system configuration specification written in ConfSolve is compiled into the standard constraint programming language MiniZinc [6]. It is then solved using the Gecode [3] constraint solver, the output of which is parsed back into the ConfSolve compiler, and combined with the original ConfSolve model to produce structured text output similar to JavaScript Object Notation (JSON). We use the structured text to generate configuration files for the system configuration tool Puppet [7], and the visualisation tool GraphViz. By generating configuration files for Puppet, we avoid the need for ConfSolve to perform system configuration tasks itself, so we need concern ourselves only with the compilation process.

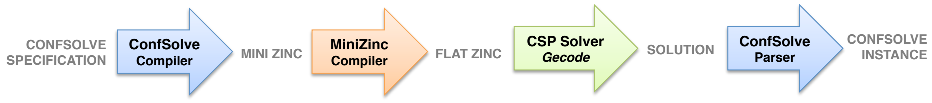


Figure 1. The architecture of ConfSolve. A ConfSolve instance is a structured text file representing a single solution.

2.1 The ConfSolve Language

ConfSolve aims to bridge the gap between existing object-oriented system configuration languages and solver input languages such as MiniZinc. It is designed to be as familiar to system administrators as possible, providing object-orientation with classes, inheritance, reference types, and enumerations, as well as a more Java-like syntax, though this is still evolving. The remainder of the section introduces the ConfSolve language by way of an example. A full grammar is given in figure 2; we intend to provide a complete formal description in a later paper.

An example system administration task which could benefit from constraint solving is the allocation of virtual machines onto an array of homogenous servers. This set-up is typical of enterprises which have adopted IaaS (infrastructure as a service), or cloud computing datacenters. A virtual machine is referred to as a *role* and a physical machine as a *machine*; the goal is to allocate each role to

```
           ⟨spec⟩ ::= ⟨declaration⟩*
    ⟨declaration⟩ ::= ⟨class⟩
                    | ⟨enum⟩
                    | ⟨variable⟩
          ⟨class⟩ ::= class ⟨identifier⟩ (extends ⟨simple type⟩)? { ⟨member⟩* }
           ⟨enum⟩ ::= enum ⟨identifier⟩ { ⟨identifier⟩ (, ⟨identifier⟩)* }
         ⟨member⟩ ::= ⟨variable⟩
                    | ⟨constraint⟩
       ⟨variable⟩ ::= var ⟨identifier⟩ as ⟨type⟩ ⟨terminator⟩

           ⟨type⟩ ::= ⟨simple type⟩
                    | ⟨ref type⟩
    ⟨simple type⟩ ::= ⟨identifier⟩
                    | ⟨set type⟩
       ⟨set type⟩ ::= ⟨identfier⟩ ⟨set bounds⟩
     ⟨set bounds⟩ ::= [ (⟨int literal⟩ ..)? ⟨int literal⟩ ]
       ⟨ref type⟩ ::= ref ⟨simple type⟩
                    | ref ( ⟨set type⟩ )

     ⟨constraint⟩ ::= where ⟨expression⟩ ⟨terminator⟩

          ⟨block⟩ ::= { (⟨expression⟩ ⟨terminator⟩)* }
     ⟨expression⟩ ::= ⟨relational expr⟩
                    | ⟨arithmetic expr⟩
                    | ( ⟨expression⟩ )
      ⟨atom expr⟩ ::= ⟨identifier⟩
                    | ⟨member expr⟩
    ⟨member expr⟩ ::= ⟨atom expr⟩ . ⟨identifier⟩
⟨relational expr⟩ ::= true | false
                    | ⟨atom expr⟩
                    | !⟨relational expr⟩
                    | ⟨expression⟩ ⟨boolean op⟩ ⟨expression⟩
                    | ⟨expression⟩ ⟨relational op⟩ ⟨expression⟩
                    | if ( ⟨expression⟩ ) ⟨expression block⟩
                    | foreach ( ⟨identfier⟩ in ⟨atom expr⟩ (where ⟨relational expr⟩)? ) ⟨block⟩
⟨arithmetic expr⟩ ::= ⟨number⟩
                    | ⟨atom expr⟩
                    | -⟨arithmetic expr⟩
                    | ⟨arithmetic expr⟩ ⟨arithmetic op⟩ ⟨arithmetic expr⟩
                    | sum ( ⟨identfier⟩ in ⟨atom expr⟩ (where ⟨relational expr⟩)? ) ⟨block⟩
                    | card ( ⟨atom expr⟩ )
  ⟨relational op⟩ ::= == | != | < | > | <= | >= | in | subset | union | intersection
     ⟨boolean op⟩ ::= && | || | -> | <->
  ⟨arithmetic op⟩ ::= + | - | / | * | % | ^

     ⟨identifier⟩ ::= (⟨letter⟩ | _ ) (⟨letter⟩ | _ |⟨digit⟩)
         ⟨number⟩ ::= -? ⟨digit⟩+
     ⟨terminator⟩ ::= ; | ⟨new line⟩
```

Figure 2. Grammar for the ConfSolve language. Whitespace is ignored, except when evaluating the ⟨terminator⟩ rule.

a machine, so that the sum of the CPU, RAM, and disk space, required by each role does not exceed that provided by the host machine.

In example 1a, we define a `Machine` class to represent the physical machines in the system; these are homogenous with each having 16 units of CPU (1 unit = 0.5 CPU), 16GB RAM, and 2TB of disk space. A machine may be on either a public or a private network. Virtual machines extend the `Role` class, which is available in the two different sizes, `LargeRole` and `SmallRole`. These define the amount of CPU and RAM that the virtual machine requires, and place an upper bound on the size of the disk space which it may require. The `Role` class contains a variable `host` which is declared to be of type `ref Machine`; this is an object reference to be resolved by the solver.

Example 1a ConfSolve type declarations

```
enum Network { Public, Private }

class Machine {
  var cpu as int
  var memory as int       // MB
  var disk as int         // GB
  var network as Network;

  where cpu == 16
  where memory == 16384
  where disk == 2048
  where network = Network.Public
}

class Role {
  var host as ref Machine
  var disk as int
  var cpu as int
  var memory as int
  var network as Network
}

class SmallRole extends Role {
    where cpu == 1
    where memory == 768
    where disk <= 20
}

class LargeRole extends Role {
   where cpu == 4
   where memory == 3584
   where disk <= 490
}
```

In example 1b, we declare global variables and place constraints over them, namely a set of two machine objects, and two role objects. To allow a constraint to be written over all roles, the variable `roles`—a set of references to roles—is introduced. Finally, a `foreach` constraint ensures that for each set of roles hosted on a given machine, the sum of the consumed resources does not exceed the resources provided by the host machine.

Example 1b ConfSolve instance and constraint declarations

```
var machines as Machine[2]

var sql_server as LargeRole
where sql_server.disk == 412

var web_server as SmallRole
where web_server.disk == 15
where web_server.network == Network.Public

var roles as ref Role[2]

where foreach (m in machines) {
  sum (r in roles where r.host == m) {
    r.cpu
  } <= m.cpu

  sum (r in roles where r.host == m) {
    r.memory
  } <= m.memory

  sum (r in roles where r.host == m) {
    r.disk
  } <= m.disk
}
```

2.2 Compiling to MiniZinc

The compilation process from ConfSolve to MiniZinc consists of flattening the object and reference types, unrolling `foreach` constraints, and replacing enums with integer constants. The flattening process is still under development and will be explained fully in a later paper, so instead we provide an overview of the MiniZinc generated for the example problem given in this paper. Flattening of object references in ConfSolve introduces a large number of constraints and variables into the corresponding MiniZinc, as does the unrolling of `foreach` constraints.

Variables are flattened according to their nesting in the global scope; starting at the top-level of the global scope, and proceeding downwards. Assignment of constants is automatically determined in the case of equality where the left hand side is a variable and the right hand side contains no variables. For each machine in the `machines` set, MiniZinc of the following form is generated:

```
int : machines_1_cpu = 16;
int : machines_1_memory = 16384;
int : machines_1_disk = 2048;
```

This is the code for the object `machines[1]`, the first element of the `machines` set. We do not use MiniZinc's set type because it does not allow the nesting of sets within sets, likewise for MiniZinc's array type. In this example there is no nesting, so an array could be used, but we have left this as a compiler optimisation to be implemented later.

Each role (`role_1`, `role_2`), is flattened in a similar manner. As MiniZinc does not have an enumeration type, we substitute `Network` values for integer constants:

```
int : sql__server_disk = 412;
int : sql__server_cpu = 4;
int : sql__server_memory = 3584;
var 0..1 : sql__server_network;
```

The underscore character is escaped as a double underscore to avoid name collisions, for example if there were also a global variable `sql` with the attribute `server`.

Object references are encoded as integers. Each object is given a unique integer identifier by the compiler, which is used to determine reference equality. The set of references `roles`, and the `host` variables of the two roles are encoded as follows: (where 1,2 are the identifiers for the two machines, and 3,4 are identifiers for the two roles)

```
var set of {3, 4} : roles;
var {1, 2} : sql__server_host;
var {1, 2} : web__server_host;
```

The `foreach` and `sum` constraints are unrolled, because they are able to quantify over sets of reference types—in this case both `roles` and `machines`—which MiniZinc lacks. Unrolling creates a single expression, which for the the foreach/sum constraint over `cpu` is of the form:

```
(bool2int(sql__server_host = 1) * bool2int(3 in roles) * sql__server_cpu) +
(bool2int(web__server_host = 1) * bool2int(4 in roles) * web__server_cpu)
<= machines_1_cpu
/\
(bool2int(sql__server_host = 2) * bool2int(3 in roles) * sql__server_cpu) +
(bool2int(web__server_host = 2) * bool2int(4 in roles) * web__server_cpu)
<= machines_2_cpu
```

Here, the expression `bool2int(sql__server_host = 1)` is used to determine reference equality, which corresponds to the ConfSovle sum filter `where r.host == m`. The expression `constraint bool2int(3 in roles)` ensures that the `roles` set contains a reference to the object in question—in this example, the constraint is redundant, but our compiler does not yet identify this.

2.3 Generating Puppet

Once the generated MiniZinc has been solved by Gecode, the solution is parsed back into the ConfSolve compiler, which maps the flat variable assignments back to an object-oriented model, and outputs this in structured format similar to JSON. Example 2 shows the solution output for the example problem; we can see that the `host` variables have been assigned references to the two physical machines.

Example 2 A ConfSolve solution for the example problem

```
roles: {sql_server, web_server};

machines[1]: Machine {
    cpu: 16;
    memory: 16384;
    disk: 2048;
    network: Public;
}

machines[2]: Machine {
    cpu: 16;
    memory: 16384;
    disk: 2048;
    network: Public;
}

sql_server: LargeRole {
    disk: 412;
    cpu: 4;
    memory: 3584;
    network: Public;
    host: machines[1];
}

web_server: SmallRole {
    disk: 15;
    cpu: 1;
    memory: 768;
    network: Public;
    host: machines[1];
}
```

The Puppet configuration tool consists of a central server, and a client running on each machine. The clients periodically request their declarative configuration from the server, which is converted into a sequence of imperative configuration steps on the client, extensible via a plug-in system. There are a number of widely adopted system configuration tools in existence, each with its own input language. Rather than tie ConfSolve to a particular system, the structured text output in example 2 is instead parsed by a separate program and converted into the appropriate format.

The generated Puppet code is shown in example 3. Each `node` is a physical machine, with an assigned set of roles. The roles `SqlServer` and `WebServer` are defined by the administrator in other files; there is no need for these static elements of the configuration to be encoded in ConfSolve. Ultimately the ConfSolve language could be extended to support all the features of Puppet or *vice versa*, but this significant undertaking is not necessary to conduct our research.

Example 3 Puppet code generated from ConfSolve solution file

```
node machine_1 {
  include SqlServer
}

node machine_2 {
  include WebServer
}
```

3 Evaluation

In order to determine the size of problem which our model representation can scale to, we expanded the example in this paper by increasing both the number of physical and virtual machines in the system. An enterprise using IaaS may wish to configure tens of machines in one go, or several hundred—it is these systems which are of the most interest to us. A cloud environment is much larger; as many as 10,000 virtual machines may be configured at once, a task better suited to a carefully tailored model.

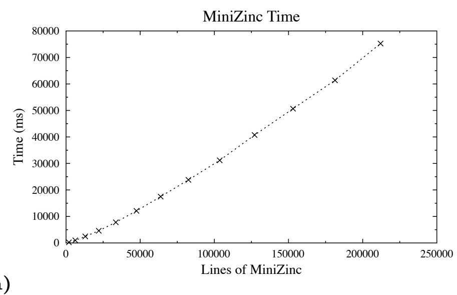
The ratio of physical to virtual machines was maintained at 1:4, and every virtual machine is of size `LargeRole`. The evaluation was performed on a machine with a 2GHz Intel Core i7 processor, using MiniZinc 2.2 and Gecode 3.5.0.

The time taken for `mzn2fzn` to convert our generated MiniZinc into the FlatZinc is shown in figure 3a. The time is compared with the number of lines of MiniZinc generated, because the `foreach` and `sum` expressions are unrolled by the ConfSolve compiler, resulting in an exponential increase in the model size. The time taken for `mzn2fzn` processing increases linearly with the problem size, but takes considerably longer than we had anticipated. Figure 3c shows the time taken for Gecode to solve the 520 virtual machines problem is just under 5000 milliseconds, but it took `mzn2fzn` over 75 seconds to compile the MiniZinc, using over 1700MB of memory in the process (see figure 3b).

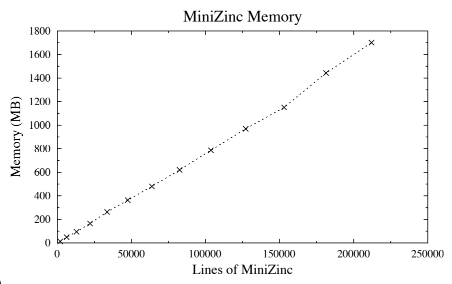
As our test machine had 2GB of free memory, the largest problem we solved was 520 virtual machines, allocated across 130 physical machines. Figure 3c show that solving took just under 5 seconds, consuming all 2GB of free memory (see figure 3d).

4 Conclusions and Future Work

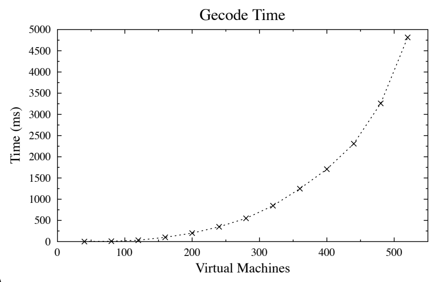
We have developed an object-oriented system configuration language, in which constraints are used to specify valid configurations. The object-oriented code is compiled to MiniZinc, and solved using the Gecode constraint solver. Writing an object-oriented model in ConfSolve is simpler than implementing the corresponding problem directly in MiniZinc. Evaluation of a simple virtual machine allocation problem showed that despite its size, the flattened problem can be solved in a reasonable timeframe for some IaaS configurations, but not for a cloud-sized configuration. For a large-scale configuration where a solve time of an hour would be acceptable, the virtual machine problem is memory-bound. We suspect that the bespoke and heterogeneous nature of IaaS configurations will be of particular interest in future research. MiniZinc to FlatZinc translation acted as a bottleneck, taking considerably longer to execute than the solver



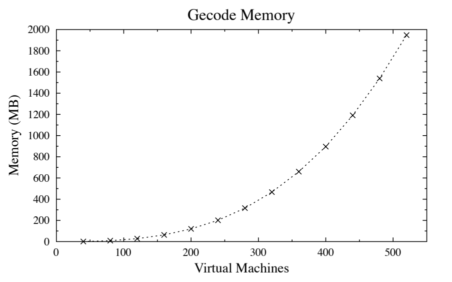
a)



b)



c)



d)

Figure 3. a) time taken for MiniZinc to generate the corresponding FlatZinc, in milliseconds **b)** memory consumed by MiniZinc, in megabytes **b)** time taken for Gecode to find a solution to the FlatZinc encoded problem, in milliseconds **d)** memory consumed by Gecode, in megabytes.

itself, but we have no reason to believe that this problem is fundamental, rather it appears to be an implementation issue in `mzn2fzn`.

In future work, we plan to expand the ConfSolve language, and formally define both it and the compilation process. Our flattening algorithm could readily be enhanced to generate more efficient MiniZinc code: leaving unrolling for MiniZinc to perform where possible, making use of arrays, and using global constraints where possible (such as a bin-packing constraint). It would be desirable to break symmetry in the generated MiniZinc models. We are currently investigating the use of refinement types [1], and optimisation goals.

The example problem, documentation, and cross-platform binaries are available to download at `http://homepages.inf.ed.ac.uk/s0968244/modref2011`.

Acknowledgements. Thanks to Andy Gordon of Microsoft Research for many helpful discussions. This work was funded by Microsoft Research through their European PhD Scholarship Programme.

References

1. Bierman, G., Gordon, A., Langworthy, D.: Semantic Subtyping with an SMT Solver (2010), `http://research.microsoft.com/en-us/um/people/adg/Publications/dminor.pdf`

2. Fleishanderl, G., Friedrich, G., Haselbock, A., Schreiner, H., Stumptner, M.: Configuring large systems using generative constraint satisfaction. IEEE Intelligent Systems and their applications 13(4), 59–68 (1998)
3. Gecode Team: Gecode: Genetic constraint development environment (2006), available from `http://www.gecode.org`
4. Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., Toft, P.: The SmartFrog configuration management framework. SIGOPS Oper. Syst. Rev. 43, 16–25 (January 2009), `http://doi.acm.org/10.1145/1496909.1496915`
5. Narain, S., Levin, G., Malik, S., Kaul, V.: Declarative infrastructure configuration synthesis and debugging. Journal of Network and Systems Management 16(3), 235–258 (2008)
6. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. Principles and Practice of Constraint Programming (CP 2007) pp. 529–543 (2007)
7. Puppet Labs: Puppet (2008), available from `http://www.puppetlabs.com/puppet/`
8. Ramshaw, L., Sahai, A., Saxe, J., Singhal, S.: Cauldron: A policy-based design tool. In: 7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006). pp. 113–122 (2006)