A Declarative Approach to Automated Configuration

John A. Hewson
School of Informatics
University of Edinburgh
`john.hewson@ed.ac.uk`

Paul Anderson
School of Informatics
University of Edinburgh
`dcspaul@ed.ac.uk`

Andrew D. Gordon
Microsoft Research &
School of Informatics
University of Edinburgh
`adg@microsoft.com`

Abstract

System administrators increasingly use declarative, object-oriented languages to configure their systems. Extending such systems with automated analysis and decision making is an area of active research. We introduce ConfSolve, an object-oriented declarative configuration language, in which logical constraints over a system can be specified. Verification, impact analysis or even the generation of valid configurations can then be performed, by translation to a Constraint Satisfaction Problem (CSP), which is solved with an off-the-shelf solver. We present a full definition of our language and its compilation process, and show that our implementation outperforms previous work utilising an SMT solver, while adding new features such as optimisation.

Keywords

configuration management, automation, constraints

1 Introduction

Configuration of large computing installations is increasingly performed by automated tools which make use of declarative, object-oriented languages [1, 2, 3]. These tools replace low-level scripts which describe the steps needed to achieve a given system state, with a high-level declarative model of the goal state of the system. Such tools can be used to configure workstations, servers, or network hardware, and have proven popular among administrators of large or complex sites.

However, such tools do not facilitate automated analyses, such as checking that a configuration is valid, or assigning values automatically given a set of constraints: instead it is left to the administrator to decide which configurations are possible. Indeed, this predominantly manual process is often subject to inefficiencies and errors [4].

There is a need for a general-purpose system configuration language which can be used to model a broad range of configuration problems involving complex declarative constraints, in an easy-to-use manner. Such a tool could be used to make or help make better decisions, in a number of possible scenarios:

1. Verification of a system configuration according to some model.
2. Impact-analysis of configuration changes.
3. Generating valid configurations from a model.
4. Optimising a configuration according to a model.

There has been recent interest in configuration tools which, given a model of a system, are able to perform analyses to aid the system administrator. Recent research into such systems has made use of off-the-shelf boolean satisfiability (SAT) solvers [5, 6] and constraint-logic-programming (CLP) systems [3]. These approaches are promising, but there still remains a need for a rigorously defined high-level modelling language which supports constraints, and provides sufficient expressivity to be of use in system configuration, while scaling to problems of a practical size.

We believe the constraint programming (CP) is a natural candidate for modelling and solving such problems. In this paper we present ConfSolve, a system configuration language, in which constraints over valid solutions may be specified, and valid concrete configurations may be either validated or generated via a constraint satisfaction problem (CSP) solver. ConfSolve aims to be a language general enough to describe a large range of configuration problems in a manner natural to a system administrator, without requiring expertise in a constraint modelling language, or expert help to construct models.

Contributions of the paper

1. Define a constraint-based object-oriented configuration language.
2. Define the translation of the language to a CSP (encoded in MiniZinc [7]).
3. Show that translated models can scale to problems of a useful size.
4. Demonstrate that the language can be used to model problems identified in previous work based on SMT, and scale to significantly larger problems.

Structure of the paper

The remainder of this paper is structured as follows: We introduce the ConfSolve language by means of example, give an overview of the MiniZinc constraint language, present the abstract grammar of ConfSolve, describe its type system, and describe a method for its transformation into MiniZinc. Finally, we provide experimental results demonstrating that our method outperforms previous work, before discussing the implications and directions for future work.

2 Modelling with ConfSolve

ConfSolve provides the user with an object-oriented declarative language, with a Java-like syntax, which adheres to several key principles:

1. Order never matters. Declaration and usage can occur in any order with no difference in meaning.
2. Everything is an expression, except declarations.
3. All classes are equal: there are no built-in classes with special meanings such as Machine or File.

Variables and Classes: A ConfSolve model consists of a global scope in which strongly-typed variables, classes, and enumerations may be declared. For example, a simple machine may be defined as:

```
enum OperatingSystem { Windows, UNIX, OSX }

class Machine {
  var os as OperatingSystem;
  var cpus as 1..4;
  var memory as int;
}

var m0 as Machine;
```

In which `m0` is a Machine object in the global scope, with members `os`, an enumeration; `cpus`, an integer subrange; and `memory`, an unbounded integer.

Member variables may also declare objects, allowing the nesting of child objects within a parent object. For example, we could add a network interface to the machine definition:

```
class Machine {
  ...
  var en0 as NetworkInterface;
}

class NetworkInterface {
  var subnet as 0..3;
}
```

An instance of NetworkInterface will be created whenever a Machine is instantiated. The lifetime of the NetworkInterface instance is tied to that of its parent object, and is not shared between different instances of Machine.

Inheritance: Objects support classical single inheritance via abstract classes. For example, we declare a class model machine-roles, with specialised subclasses for web servers:

```
abstract class Role {
  var machine as ref Machine;
}

class WebServer extends Role {
  var port as 0..65535;
}
```

References: Associations between objects are modelled using reference types. References are handles to objects elsewhere in the model, which cannot be null. Consider an instance of the web server role:

```
var ws1 as WebServer;
```

In the previous declaration of the Role class, the variable `machine` was declared as a Machine reference. Thus `w1` contains a reference to a machine, in this case it will refer to `m0`, as it is the only machine we have so far declared. The solver will automatically assign the value of a reference to any instance of the appropriate type, so if we always wanted `ws1` to run on `m1` we would also need to write:

```
ws1.machine = m1;
```

Which is an example of an equality constraint.

Constraints: Constraints are expressions which must hold in any solution to the model. For example, introducing a database-server role which can be either a slave or master, and must be peered with another slave or master, as appropriate:

```
enum DatabaseRole { Master, Slave }

class DatabaseServer extends Role {
  var role as DatabaseRole;

  // slave or master
  var peer as ref DatabaseServer;

  // the peer cannot be itself
  peer != this;

  // a master's peer must be a slave,
  // and a slave's peer must be a master
  role != peer.role;
}
```

This allow us to define two database server roles:

```
var masterDB as DatabaseServer;
masterDB.role = DatabaseRole.Master;
masterDB.peer = slaveDB;

var slaveDB as DatabaseServer;
slaveDB.role = DatabaseRole.Slave;
slaveDB.peer = masterDB;
```

Likewise we may define logical boot-disks on a SAN for each physical machine, and assign logical boot-disks to the two roles:

```
var db_disk as LogicalDisk;
db_disk.capacityGB = 2048;

var web_disk as LogicalDisk;
web_disk.capacityGB = 10;
```

Udating the declarations of Machine, WebServer and DatabaseServer with:

```
class Machine {
  ...
  var bootDisk as ref LogicalDisk;
}

class WebServer extends Role {
  ...
  machine.bootDisk = web_disk;
}

class DatabaseServer extends Role {
  ...
  machine.bootDisk = db_disk;
}
```

Sets and Quantifiers: Sets of variables may be declared, for example 10 web servers:

```
var webServers as WebServer[10];
```

A quantified constraint over the members of `webServers` can ensure that each server's port is set to 80, as long as the role is not running on `m0`:

```
forall ws in webServers where ws.machine != m0 {
  ws.port = 80;
};
```

As the port of `m0` is not constrained, the solver is free to choose its value. Should we want to specify it ourselves, we could write:

```
m0.port = 443;
```

So far our model contains only one Machine, `m0`, let's declare a class to describe a rack of 24 machines:

```
class Rack {
  var machines as Machine[24];
}

var r1 as Rack;
var r2 as Rack;
```

Here `machines` is declared as a set of objects, 24 new instances of Machine will be created as children of each Rack instance, in this case `r1`.

Given the following constraints, which place the master and slave databases in different racks:

```
masterDB.machine in r1;
slaveDB.machine in r2;
```

If rack `r2` fails, is there a valid solution? The answer is clearly no, and we can perform a quick impact analysis of such a failure by simply commenting out `r2`. Alternatively we could modify the definition of a Role to be:

```
abstract class Role {
  var machine as ref Machine;
  machine in r2.machines = false;
}
```

In either case ConfSolve will report that there is no valid solution to the model.

Optimisation Minimisation and maximisation constraints may be used for any solver-populated variable. The solver will find not just a valid value, but an optimal value, given some expression to be maximised or minimised. For example, if we prefer database masters and slaves to be in different racks, but this is not a hard constraint, then we can remove the constraints:

```
masterDB.machine in r1;
slaveDB.machine in r2;
```

Replacing them with a constraint maximising the number of machines with peers on different racks:

```
var databases as ref DatabaseServer[2];
var racks as ref Rack[2];

maximize sum r in racks {
  count (db in databases
    where db.machine in r.machines
      != db.peer.machine in r.machines);
};
```

Output The final output of the ConfSolve compiler, once solving is complete, is an object-tree in a format similar to the popular JSON (JavaScript Object Notation) format, which we call CSON (ConfSolve Output Notation). We describe CSON in full in Section 6.3. For example, the CSON corresponding to a model containing only `m0` is as follows:

```
{
  m0: Machine {
    os: OperatingSystem.UNIX,
    cpus: int 4,
    memory: int 1024,
    en0: NetworkInterface {
      subnet: 0,
    },
    bootDisk: ref LogicalDisk web_disk,
  },
  web_disk: LogicalDisk {
    capacityGB: int 10,
  },
}
```

3 Core Syntax of ConfSolve

This section describes the abstract grammar of ConfSolve, which is independent of the concrete grammar which we chose for our implementation, and does not include concrete syntax such as semicolons, comments, or whitespace rules. This provides a concise description of the language, free from unnecessary detail. It also allows others to use their own concrete syntax, but adopt the ConfSolve abstract syntax tree (AST) in order to apply the same translation steps to target MiniZinc.

To avoid redundancy, we first define a minimal core language, and then a series of derived constructs which are defined in terms of the core language.

Syntax of Types:

$S ::= \{i_1, \ldots, i_n\}$	integer subset
$T ::=$	type
bool	boolean
int	integer
S	integer subset
u	enumeration
c	object
$T[]$	set of T
$c[n]$	set of objects, with cardinality n

Identifiers are represented by metavariables: c is a class name, v is a variable name, u is an enum name, a_i is an enum member, l is a field name; i, m and n are integers; and b is a boolean: **true** or **false**.

A ConfSolve type is either a boolean, and unbounded integer, an finite subset of integers, an enumeration, and object, a set of any of those, or a set of objects with a fixed cardinality. Nesting of set types is not supported, as there is no direct way to represent sets of sets in MiniZinc, however an object may contain a set field, which may itself contain objects, giving the user the means to model arbitrary nesting via objects. Variable declarations may not be of type $c[]$, which is reserved for use during type checking (see section 4).

Integers are the only unbounded type in ConfSolve, as is the case in MiniZinc. Consequently a set of **int** cannot be declared, a restriction which we formally impose when we describe the type system in section 4. This restriction stems from the fact that quantifiers are unrolled as part of the MiniZinc to FlatZinc compilation process, which is not possible when the domain of the quantifier is infinite. As it is usually undesirable to have unbounded models, it is worth observing that the benefit of the **int** type is that it allows constants whose domain is not known to be declared and assigned separately. Thus one can define **var** *id* **as int** and later write the constraint $id = 4$. It also allows the user to avoid having to specify the domain of functionally defined variable values which ultimately depend on only variables with finite domains, for example the domain of $x = 5 * y + 3$, where y is a constant defined elsewhere.

To reduce the complexity of the MiniZinc encoding, sets of objects $c[n]$ have the same upper and lower bound n on their cardinality. As an alternative, the user may instead use a fixed cardinality set of objects, and a variable cardinality set of references with a constraint that the latter must be resolved to only members of the former. The derived expressions in Section 3.1 address the declaration of such fixed-cardinality sets.

Syntax of Expressions:

$e ::=$	expression
this	current object
v	variable
$e.l$	field access
$u.a$	enum member
e.**size**	set cardinality
e_1 BinOp e_2	binary operator
Fold (v **in** e_1 **where** e_2) (e_3)	fold
bool2int(e)	cast bool to int
-e	negation
!e	logical not
$[e_1, \ldots, e_n]$	set literal
b	boolean literal
i	integer literal
(e)	parenthesis
Fold ::=	fold operator
forall | **exists**	quantification
sum	summation

Variables, constants, binary and unary operators, and parenthetical expressions are defined in the standard manner. Object field access $e.l$ evaluates to the field l of object e. Enum constants are written in a fully-qualified manner as $u.a$, where u is the name of the enumeration and a is a constituent member. The current object can be accessed via **this** within the body of a ClassDecl. For expressions with set-type, e.**size** evaluates to the cardinality of the set given by e. Three folds over sets are defined: universal quantification, **forall**, existential quantification, **exists**, and summation, **sum**. Folds include a **where** expression which filters the set prior to evaluating the fold. Finally, the function **bool2int** provides type-casting between boolean and integer types.

Binary Operators:

BinOp ::=	binary operator
= | > | >= | < | <= | **in** | **subset**	relational
union | **intersection**	set
&& | || | -> | <->	logical
+ | - | / | * | ^ | **mod**	arithmetic

Relational operators use the standard C-like notation, with the addition of **in** which is the set membership operator $\in$, and **subset** which is the subset operator $\subset$. Logical operators are *and*, *or*, *implies*, and *biconditional*. Arithmetic operators are standard, where ˆ is exponentiation, and **mod** is modulo.

Syntax of Models:

Model ::=	model
Declaration*	declarations
Declaration ::=	declaration
ClassDecl	class decl.
EnumDecl	enum decl.
VarDecl	var decl.
Constraint	constraint
ClassDecl ::=	class decl.
abstract? **class** c **extends** c' { (VarDecl | Constraint)* }	
EnumDecl ::=	enum decl.
enum u $\{a_1, \ldots, a_n\}$	
VarDecl ::=	variable decl.
var v **as** T	
var v **as ref** c	object reference
Constraint ::=	constraint
e	hard constraint
maximize e	soft constraint

Identifiers are represented by metavariables: c is a class name, v is a variable name, u is an enum name, a_i is an enum member, l is a field name; i, m and n are integers.

A model consists of a series of declarations, of either classes, enumerations, variables, or constraints. A class declaration may contain any number of nested variable or constraint declarations, and it may extend another class.

A declaration **class** c **extends** c' is well-formed if and only if, there is a well-formed class declaration for c' and the inheritance hierarchy is acyclic, or if c' is the top class, denoted by the distinguished name $\varnothing$. A well-formed class may contain duplicate field names.

Enumerations consist of a name and a non-empty a set of identifiers, which defines its members. Variables are always declared with a type T. Object reference variables may be declared using **var** v **as ref** c. This creates a reference which will resolve at solve-time to an instance of c elsewhere in the model, whereas the declaration **var** v **as** c allocates a new instance of c.

3.1 Derived Syntax

The grammar above describes the core of the ConfSolve language. The full language contains a number of constructs which are derived from the core language, to keep the core and its translation as simple as possible. These syntactic re-write rules are performed by the parser in our implementation. The relation $\triangleq$ means "equal by definition".

Derived Declarations:

class c { (VarDecl | Constraint)* } $\triangleq$
class c **extends** $\varnothing$ { (VarDecl | Constraint)* }

var v **as** $m\,..\,n \triangleq$
var v **as** $\{x \mid x \in \mathbb{N},\ x \geq m \wedge x \leq n\}$

var v **as** $T[m\,..\,n] \triangleq$
var v **as** $T[]$
v.**size** >= m && v.**size** <= n

minimize $e \triangleq$ **maximize** $-e$

Classes without a base type extend the top class, denoted by the distinguished name $\varnothing$. Integer subsets may be declared as ranges. Variable cardinality sets are given a shorthand notation. Minimisation is defined as negated maximisation.

Derived Expressions:

Fold (x **in** e_1) $(e_2) \triangleq$
Fold (x **in** e_1 **where true**) (e_2)

count (x **in** e_1 **where** e_2) $\triangleq$
sum (x **in** e_1 **where** e_2) (1)

count (x **in** e_1) $\triangleq$
count (x **in** e_1 **where true**)

$e_1 != e_2 \triangleq\ !(e_1 = e_2)$

$\{e_1;\ldots;e_n\} \triangleq (e_1 \wedge \cdots \wedge e_n)$

Quantifiers without filters are defined as having an always-true filter. The body of a **forall** expression is defined as the logical conjunction of its sub-expressions. The **count** expression is defined in terms of summation. The "not equals" operator is defined as negated equality. Finally, a semicolon-delimited expression block is defined as the conjunction of its sub-expressions.

4 Type System

From the information presented so far, we are able to recognise a syntactically correct ConfSolve program. However, not all syntactically correct ConfSolve programs are well-formed. For example, a program which compares booleans with integers, declares a set of **int**, or makes use of undeclared variables. To complete our description of ConfSolve, it is necessary to describe its type system.

Static typing serves two purposes, firstly to provide a level of compile-time safety, and secondly to satisfy the CSP solver's requirement that a domain is specified for each variable. The smaller the domain, the better performance one can expect from the solver. This is why the type 1..3 is more desirable than the type **int**.

We formally specify ConfSolve's type system as a *proof system*, which is a declarative specification of the rules governing the assignment of types to expressions. This is separate from the actual type checking algorithm used in the ConfSolve compiler which implements these rules.

Within a proof system types are assigned to expressions via typing *judgements*, which are applied recursively, and take the form:

$$\text{(Name)}\quad \frac{\textit{premises}}{\textit{conclusion}}$$

where premises may be written on multiple lines or separated with a space. Judgements which contain expressions require a *typing environment* to resolve variable names to their declared type, which is similar to the concept of a symbol table, used when implementing such systems.

Environments:

$E ::= v_1 : T_1, \ldots, v_n : T_n$	type environments
$dom(v_1 : T_1, \ldots, v_n : T_n) = \{v_1, \ldots v_n\}$	environment domain

We now begin the formal specification. Type system judgements may be made with respect to a typing environment E, of the form $v_1 : T_1, \ldots, v_n : T_n$, which assigns a type to each in-scope variable. We write $\varnothing$ for the initial environment with an empty map.

Typing Judgements:

$E \vdash \diamond$	environment E is well-formed
$\vdash T$	the type T is well-formed
$T <: T'$	type T is a subtype of T'
$E \vdash e : T$	in E, expression e has type T

There are four judgements which we may make. That an environment is well formed, that a type is well-formed, that a type is a subtype of another, and that an expression has a given type.

Rules of Well-Formed Environments and Types:

Where **class** c **extends** $c'\{\ldots\}$ means that there exists such a declaration. Likewise for **enum** $u\ \{a_i{}^{i\in 1..n}\}$.

$$\text{(Env Empty)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env Var)}\ \frac{E \vdash \diamond \quad \vdash T \quad v \notin dom(E)}{E, v : T \vdash \diamond} \qquad \text{(Type Bool)}\ \frac{}{\vdash \textbf{bool}}$$

$$\text{(Type Int)}\ \frac{}{\vdash \textbf{int}} \qquad \text{(Type Int Sub)}\ \frac{}{\vdash S} \qquad \text{(Type Enum)}\ \frac{\textbf{enum}\ u\ \{a_i{}^{i\in 1..n}\}}{\vdash u}$$

$$\text{(Type Obj)}\ \frac{\begin{array}{c} c' \neq \varnothing \rightarrow \vdash c' \\ \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ldots\} \end{array}}{\vdash c} \qquad \text{(Type Set)}\ \frac{\vdash T \quad T \neq \textbf{int}}{\vdash T[\,]} \qquad \text{(Type Obj Set)}\ \frac{\vdash c}{\vdash c[n]}$$

A well-formed environment is either empty, or contains a mapping of variable names to types. A well-formed type is either a bool, and int, an enum, an integer subset, an object, a set of any type other than **int**, or a set of objects with a fixed cardinality. These rules and those which follow make use of definitions introduced in the syntax of types and expressions in section 3.

Rules of Subtyping

$$\text{(Extends)}\ \frac{\textbf{class}\ c\ \textbf{extends}\ c'\ \{\ldots\}}{c <: c'} \qquad \text{(Reflex)}\ \frac{\vdash T}{T <: T} \qquad \text{(Trans)}\ \frac{\begin{array}{c} T <: T' \\ T' <: T'' \end{array}}{T <: T''}$$

$$\text{(Set Subtype)}\ \frac{T <: T'}{T[\,] <: T'[\,]} \qquad \text{(Obj n-Set Subtype)}\ \frac{c <: c'}{c[n] <: c'[n]}$$

$$\text{(Obj Set Subtype)}\ \frac{\vdash c}{c[n] <: c[\,]} \qquad \text{(Int Sub)}\ \frac{}{S <: \textbf{int}} \qquad \text{(Int Sub Union)}\ \frac{}{S <: S \cup S'}$$

Our rules of type assignment make heavy use of subtyping for all types, not just objects, in order to make the type-assignment rules simpler. The first three rules define the familiar rules of class-based inheritance, reflexivity (that a type is a subtype of itself) and transitivity (that a subtype is a subtype of any of its supertypes). The next three rules extend this notion to sets. The final two rules define integer subsets as a subtype of **int**, and that an integer subset is a subtype of the union between that subset and another subset. Thus $\{1,2\}$ <: **int**, the purpose of which will be explained shortly.

With all of the pre-requisites in place, we can finally present the rules of type assignment for expressions:

Rules of Type Assignment: $E \vdash e : T$

(Subsum)
$$\frac{E \vdash e : T \quad T <: T'}{E \vdash e : T'}$$

(Var)
$$\frac{E \vdash \diamond \quad (v : T) \in E}{E \vdash v : T}$$

(Bool Const)
$$\frac{E \vdash \diamond}{E \vdash b : \mathbf{bool}}$$

(Int Const)
$$\frac{E \vdash \diamond}{E \vdash i : \{i\}}$$

(Enum Const)
$$\frac{E \vdash \diamond}{E \vdash u.a : u}$$

(Set)
$$\frac{E \vdash e_i : T \quad \forall i \in 1..n}{E \vdash [e_1, \ldots, e_n] : T[]}$$

(Eq)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 = e_2 : \mathbf{bool}}$$

(Ineq Op)
$$\frac{\oplus \in \{>, >=, <, <=\} \quad E \vdash e_1 : \mathbf{int} \quad E \vdash e_2 : \mathbf{int}}{E \vdash e_1 \oplus e_2 : \mathbf{bool}}$$

(In Op)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T[]}{E \vdash e_1 \ \mathbf{in} \ e_2 : \mathbf{bool}}$$

(Subset Op)
$$\frac{E \vdash e_1 : T[] \quad E \vdash e_2 : T[]}{E \vdash e_1 \ \mathbf{subset} \ e_2 : \mathbf{bool}}$$

(Logical Op)
$$\frac{\oplus \in \text{logical} \quad E \vdash e_1 : \mathbf{bool} \quad E \vdash e_2 : \mathbf{bool}}{E \vdash e_1 \oplus e_2 : \mathbf{bool}}$$

(Set Op)
$$\frac{\oplus \in \{\mathbf{union}, \mathbf{intersection}\} \quad E \vdash e_1 : T[] \quad E \vdash e_2 : T[]}{E \vdash e_1 \oplus e_2 : T[]}$$

(Int Sub Set Op)
$$\frac{\oplus \in \{\mathbf{union}, \mathbf{intersection}\} \quad E \vdash e_1 : S_1[] \quad E \vdash e_2 : S_2[]}{E \vdash e_1 \oplus e_2 : (S_1 \oplus S_2)[]}$$

(Arith Op Int)
$$\frac{\oplus \in \text{arithmetic} \quad E \vdash e_1 : \mathbf{int} \quad E \vdash e_2 : \mathbf{int}}{E \vdash e_1 \oplus e_2 : \mathbf{int}}$$

(Arith Op Int Sub)
$$\frac{\oplus \in \text{arithmetic} \quad E \vdash e_1 : S_1 \quad E \vdash e_2 : S_2}{E \vdash e_1 \oplus e_2 : \{x_1 \oplus x_2 \mid x_1 \in S_1,\ x_2 \in S_2\}}$$

(Dot)
$$\frac{E \vdash e : c \quad c <: c' \quad j \in 1..n \quad \mathbf{class}\ c\ \mathbf{extends}\ c'\ \{\mathbf{var}\ l_i\ \mathbf{as}\ T_i^{i \in 1..n}\}}{E \vdash e.l_j : T_j}$$

(This)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{this}_c : c}$$

(Set Card)
$$\frac{E \vdash e : T[]}{E \vdash e.\mathbf{size} : \mathbf{int}}$$

(Channel)
$$\frac{E \vdash e : \mathbf{int}}{E \vdash \mathbf{int2bool}(e) : \mathbf{bool}}$$

(Quant)
$$\frac{Q \in \{\mathbf{forall}, \mathbf{exists}\} \quad E \vdash e_1 : T[] \quad E, v : T \vdash e_2 : \mathbf{bool} \quad E, v : T \vdash e_3 : \mathbf{bool}}{E \vdash Q\ v\ \mathbf{in}\ e_1\ \mathbf{where}\ e_2\ (e_3) : \mathbf{bool}}$$

(Sum)
$$\frac{E \vdash e_1 : T[] \quad E, v : T \vdash e_2 : \mathbf{bool} \quad E, v : T \vdash e_3 : \mathbf{int}}{E \vdash \mathbf{sum}\ v\ \mathbf{in}\ e_1\ \mathbf{where}\ e_2\ (e_3) : \mathbf{int}}$$

The first rule is that of subsumption, that an expression of a type may also take any of its supertypes. This is followed by variable resolution in the environment, and boolean integer and enumeration constants. The type of an integer constant is a singleton set containing the constant's value. This is significant, because ConfSolve forbids sets of integers, thus the set literal $[1,2,3]$ is legal in ConfSolve, having type $\{1\} \cup \{2\} \cup \{3\} = \{1,2,3\}$. Indeed, it is the reason why the integer literal 1 has type $\{1\}$, and why the type system makes an effort not to promote to integer subsets to **int** too readily.

The rules for comparisons (Eq)–(Logical Op) are relatively straightforward, and make heavy use of the subtyping rules. For example, recall that S is a subtype of **int** and is this subject to the rule (Ineq Op). Likewise, when we state that both expressions in the equality (Eq) rule must be of type T, that is not to say that they are of the same subtype, only that for both types there exists a common supertype for which they may be substituted.

The set operation (Set Op) follows a similar form, but (Int Sub Set Op) provides a specialised rule for handling integer subsets, which applies the intersection or union operator to the subset itself, as appropriate. The arithmetic operations (Arith Op Int) and (Arith Op Int Sub) follow this same pattern, with the latter being somewhat unusual. Namely, that for arithmetic operations between integer subsets, the resulting type is the integer subset containing the result of the application of the operation to all pairs in the two source subsets. The motivation for this is the same as for the (Int Const) rule, that the expression $1+2$ has type 3, and thus the set literal $[1+2]$ is legal.

The next four rules, from (Dot) to (Channel) specify types for member variable access, the **this** pseudo-variable, set cardinality, and channeling integers to booleans.

The final two rules specify types for quantification and summation expressions, which are the only rules which introduce variables into the environment, *i.e.*, the scope of v is e_2 and e_3, the **where** and body clauses, respectively.

In order to provide a succinct notation, the system has the following derived properties: if $T <: T'$ then $\vdash T$ and $\vdash T'$; and if $E \vdash e : T$ then $E \vdash \diamond$ and $\vdash T$ and $\mathit{freevars}(e) \subseteq \mathit{dom}(E)$. That is, that the subtype judgement always involves well-formed types, and that any free variables in an expression are well-formed members of its environment.

5 ConfSolve and MiniZinc

In this section we provide an overview of the process of compiling and solving a ConfSolve model, and of the MiniZinc constraint modelling language.

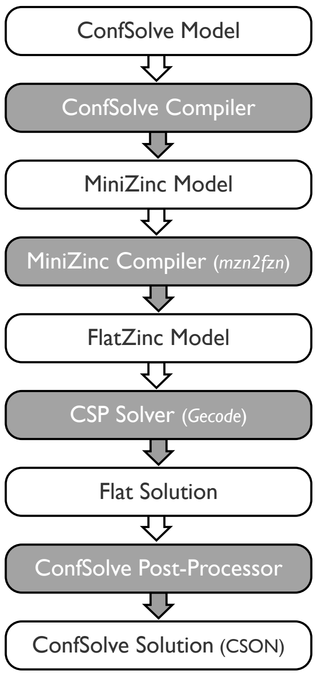

Figure 1: Compiling and solving a ConfSolve model. White boxes are files; shaded boxes are processes.

Compiling and solving a ConfSolve model requires several steps, which are illustrated in Figure 1; the steps are as follows:

1. The ConfSolve compiler is invoked; the model is translated into a CSP expressed in MiniZinc. This is described in Section 6.
2. The MiniZinc model is compiled into a FlatZinc model using *mzn2fzn* [7].
3. The FlatZinc model is solved using a constraint solver. In our implementation we use Gecode [8].
4. The solution found by the solver is parsed by the ConfSolve post-processor and combined with the original model to produce an object-tree (CSON) representing the solution. This is described in Section 6.3.

Implementation Our prototype implementation of the ConfSolve compiler and post-processor consists of approximately 1900 lines of F#/OCaml.

5.1 MiniZinc

MiniZinc [7] is a modelling language for constraint programming (CP) problems, in which models are specified as declarative constraints over sets of variables and domains. MiniZinc models are compiled into FlatZinc, which is a low-level input format supported by many CP solvers.

MiniZinc is a form of many-sorted first-order logic. Variables can be defined with integer, boolean, or set domains; arrays of variables may be declared; constraint expressions include quantification over sets, and logical implication. However, MiniZinc is restricted in its expressibility: sets are limited to bounded integer and boolean members (no sets of sets), arrays cannot contain other arrays and must be of fixed size, and quantifications are restricted to expressions with a constant value at compile-time. These restrictions arise from the fact that MiniZinc is designed to be a thin wrapper around the primitives supported by CP solvers.

Example Let us examine a MiniZinc model generated by the ConfSolve compiler. This model corresponds to the four instances of the DatabaseServer class from the example in Section 2, the relevant ConfSolve model for which is:

```
enum DatabaseRole { Master, Slave }

class DatabaseServer extends Role {
  var role as DatabaseRole;

  // slave or master
  var peer as ref DatabaseServer;

  // the peer cannot be itself
  peer != this;
  // a master's peer must be a slave,
  // and a slave's peer must be a master
  role != peer.role;
}

// instances
var masterDB as DatabaseServer;
masterDB.role = DatabaseRole.Master;
masterDB.peer = slaveDB;

var slaveDB as DatabaseServer;
slaveDB.role = DatabaseRole.Slave;
slaveDB.peer = masterDB;
```

The model begins with variable declarations. As MiniZinc does not support records or objects, each field in the DatabaseServer class is declared as a separate array, where the number of elements in the array is equal to the number of DatabaseServer instances in the model, in this case 4. The domain of the array elements correspond to the type of the field. In the following example the DatabaseServer_role has as its domain the contiguous set of integers 1..2, which are indexes into the DatabaseRole enumeration:

```
array[1..4] of var 1..2: DatabaseServer_role;
array[1..4] of var 1..4: DatabaseServer_peer;
```

Constraints are first-order expressions which restrict the values a variable may take from its domain. Each is a boolean expression which must evaluate to true. In this case, the constraints come from the DatabaseServer class, and so are wrapped with a `forall` expression which applies the constraint to all instances of DatabaseServer, and defines the value of the variable `this` to be an index into the field arrays for the DatabaseServer class. The identifier `this` has no special meaning in MiniZinc, and was chosen simply to facilitate translation:

```
constraint
 forall (this in 1..4) (
  DatabaseServer_peer[this] != this
 );

constraint
 forall (this in 1..4) (
  DatabaseServer_role[this] !=
   DatabaseServer_role[DatabaseServer_peer[this]]
 );
```

Each MiniZinc model must have a `solve` goal, which can be to either minimise or maximise an expression, or simply satisfy the constraints in the model. In this case the goal is the latter:

```
solve satisfy;
```

Finally, the model must specify which variables values will be output by the solver. This is useful when the model includes constants or intermediate variables, the values of which are not of interest. In this case, it is simply:

```
output [
 show(DatabaseServer_role),
 show(DatabaseServer_peer)
];
```

5.2 Constraint Satisfaction

The semantics of ConfSolve are defined in terms of MiniZinc. The specification of a MiniZinc problem is independent from how it will be solved, therefore ConfSolve does not make any formal guarantees about constraint satisfaction, such as the completeness of the search. Instead it is the solver which defines these properties.

There are many different approaches to CSP solving with different performance characteristics and different formal guarantees. In this paper we make use of the Gecode [8] solver which performs a global search via the classic CSP algorithm, constraint propagation. A global search is complete when variables have finite domains, but the **int** type in both ConfSolve and MiniZinc has an infinite domain. This means that any ConfSolve model which contains decision variables of type **int** will result in an incomplete search. In practice, the CSP solver uses a bound such as 32 bits for an integer, which is of course a finite domain, and the search within this range is complete. The Gecode solver which we use has a bound of 32-bits for integers.

The practical implications of search completeness are straightforward; if we limit ourselves to bounded variables then solvers exist which guarantee that if there is a solution, then it will be found. However, there is no guarantee that we will have enough time or memory to conclude the search. We evaluate some examples of this in section 7.

6 Translating ConfSolve to MiniZinc

This section defines the translation from a ConfSolve model to MiniZinc in terms of their abstract grammars. The translation occurs in two phases: a counting phase in which indexes are generated for each object, and an upper-bound on the number of object instances in the model is calculated; and a translation phase in which a MiniZinc abstract syntax tree is constructed.

6.1 Static Allocation

The static allocation phase determines the upper bound on the number of instances of each class, assigns each object an index, and records which indices are assigned to the subclasses of a given class. Its purpose is to generate the following two data structures, for use in the later translation phase:

count is a map from a class names c to an integer representing the count of the number of instances of c in the model.

indices is a map from a class name c to a set of integers representing the indices of each instance of c or one of its subclasses.

The values of *count* and *indices* are updated incrementally as counting progresses via the method described below. In order to handle inheritance we introduce the concept of a root class, that is, the topmost class in any given hierarchy. Formally, $root(c)$ is c when the superclass of c is $\varnothing$, otherwise it is the topmost superclass of c. The process of counting is as follows:

Given the definition **class** c **extends** c', for each global declaration **var** v **as** T:

when $T = c$, $count(root(c))$ is incremented, and its value is added to the sets $indices(c)$, and to $indices(c^*)$ for each ancestor c^* which c extends. This process is then repeated for each field **var** v **as** T declared in class c or any ancestor of c.

when $T = c[n]$, the case for $T = c$ is repeated n times.

6.2 Translation

The translation describes the process of generating a MiniZinc abstract syntax tree from a ConfSolve abstract syntax tree.

We make use of the notation $[\![x]\!]$ to mean "the translation of x", where x is some syntactic construct, such as a type or expression.

Translation of Types $[\![T]\!]$:

$$\mathbf{int} \triangleq \mathbf{int}$$
$$\mathbf{bool} \triangleq \mathbf{bool}$$
$$\{i_1,\ldots,i_n\} \triangleq \{i_1,\ldots,i_n\}$$
$$u \triangleq 1\,..\,num(u)$$
$$c \triangleq \begin{cases} \{\,indices(c)\,\} & \text{if } c \text{ is abstract} \\ 1\,..\,count(c) & \text{otherwise} \end{cases}$$
$$c[n] \triangleq \mathbf{set\ of}\ [\![c]\!]$$
$$B[\,] \triangleq \mathbf{set\ of}\ [\![B]\!]$$

Here we define $[\![T]\!]$ to be the MiniZinc translation of a ConfSolve type T, where $num(u)$ is the number of elements in enum u. Each ConfSolve type maps directly onto a MiniZinc type, with the exception of enumerations and objects which are translated to integer indices. For set types, the translation is recursive, but only to one level, because MiniZinc does not permit sets of sets. The translation process has not yet begun: the translation of a type is used as an intermediate step in the translations which follow.

Translation of Global Variable Declarations:

For each global declaration **var** v **as** T, introduce a declaration:

when $T = c[n]$

$[\![T]\!]$: v = {for $i \in 1..n, index(c)$}

when $T = c$

$[\![T]\!]$: v = $index(c)$

otherwise

var $[\![T]\!]$: v

For each global declaration **var** v **as ref** c, introduce a declaration:

var $[\![c]\!]$: v

The translation phase proceeds in a similar manner to the counting phase, and makes use of object indices generated in exactly the same manner as those in the *count* map. It is important that these indices are the same, so that an index corresponds to the correct value in the *indices* map. We define a function $index(c)$ which generates a new index of class $root(c)$ by counting, and returns its value.

Translation begins with global variable declarations, as shown above. When the type of the declared variable is a set of objects of class c with cardinality n, indices are generated for each of the n objects. When the type is a object of class c, a single index is generated. For all other types no values are assigned, and a **var** declaration is introduced.

Reference variable declarations are translated in the same manner as the "otherwise" case, that is a **var** declaration is introduced whose domain is the translation of the c type given in Translation of Types above.

Translation of Class-Level Variable Declarations:

For each ClassDecl defining a class c where $count(c) > 0$, containing fields **var** f_i **as** $T_i{}^{i \in 1..n}$, introduce an array for each field f_i:

when $T_i = c'[n]$

array $[1\,..\,count(c)]$ **of** $[\![T_i]\!]$: $c\_f_i =$
 [for $i \in 1..count(c)$,
  { for $j \in 1..n, index(c')$ **}]**

when $T_i = c'$

array $[1\,..\,count(c)]$ **of** $[\![T_i]\!]$: $c\_f_i =$
 [for $i \in 1..count(c), index(c')$ **]**

otherwise

$\textbf{array } [1 \mathbin{..} count(c)] \textbf{ of var } [\![T_i]\!] : c\_f_i$

Where class c contains fields **var** f_i **as ref** $c_i{}^{i \in 1..n}$, introduce an array for each field f_i:

$\textbf{array } [1 \mathbin{..} count(c)] \textbf{ of var } [\![c_i]\!] : c\_f_i$

Variable declarations nested within classes are translated by introducing an **array** which will contain the values of that field for all instances of some class c. When the type of the declared variable is a set of objects of class c' with cardinality n, an array of sets (the only nested construct permitted by MiniZinc) is declared, and for each of the $count(c)$ instances, n indices are generated. When the type is an object of class c' the declaration is simpler: an array of indices is introduced, one for each of the $count(c)$ instances. For all other types no values are assigned, and array of **var** is introduced.

Once again, reference variable declarations are translated in the same manner as the "otherwise" case, where a **var** declaration is introduced whose domain is the translation of the type c_i.

Translation of Global Constraints:

For each global constraint e, introduce a statement:

$\textbf{constraint } [\![e]\!]$

For each global constraint **maximize** e, update the objective expression o:

when o is undefined

$o = [\![e]\!]$

otherwise

$o = o + [\![e]\!]$

The translation of hard constraints consists of translating their expressions, which we discuss later. The objective expression o is the sum of every maximisation goal's expression, and is maintained throughout the translation phase. Each **maximize** constraint corresponds to a subexpression in the objective function.

Translation of Class-Level Constraints:

For each ClassDecl defining a class c where $count(c) > 0$, for each constraint e, introduce a statement:

$\textbf{constraint forall } (\text{this } \textbf{in } 1 \mathbin{..} count(c))\ ([\![e]\!])$

For each global constraint **maximize** e, update the objective expression o:

when o is undefined

$o = [\![e]\!]$

otherwise

$o = o + \textbf{sum } (\text{this } \textbf{in } 1 \mathbin{..} count(c))\ ([\![e]\!])$

Constraints may also be declared at the class-level, in which case they apply to every instance of that class. The translation occurs in the same manner as for global constraints, except that the constraints are placed over all instances in aggregate via **forall** for hard constraints and **sum** for maximisation constraints.

Translation of Expressions $[\![e]\!]$:

$$\begin{aligned}
v &\triangleq [\![v]\!] \\
\textbf{this} &\triangleq \text{this} \\
e.l &\triangleq classof(e)\_l[[\![e]\!]] \\
u.a &\triangleq eindex(u,a) \\
e_1 \text{ Op } e_2 &\triangleq [\![e_1]\!]\ [\![\text{Op}]\!]\ [\![e_2]\!] \\
e.\textbf{size} &\triangleq \textbf{card}([\![e]\!]) \\
\text{Fold } (v \textbf{ in } e_1 \textbf{ where } e_2)\ (e_3) &\triangleq \text{(see below)} \\
\textbf{bool2int}(e) &\triangleq \textbf{bool2int}([\![e]\!]) \\
-e &\triangleq -[\![e]\!] \\
!e &\triangleq \textbf{not } [\![e]\!] \\
[e_1, \ldots, e_n] &\triangleq \text{(see below)} \\
e_1 \ \hat{}\ e_2 &\triangleq [\![e_1]\!] \textbf{ pow } [\![e_2]\!] \\
\textbf{true} &\triangleq \textbf{true} \\
\textbf{false} &\triangleq \textbf{false} \\
i &\triangleq i
\end{aligned}$$

The MiniZinc translation $[\![e]\!]$ of a ConfSolve expression e is given above, where $eindex(u,a)$ is the index of element a in the declaration **enum** u $\{a_1 \ldots a_n\}$, and $classof(e)$ is the identifier c when the type of e is an object of class c. Expressions are translated recursively, with the exception of literals.

The keyword **this** is translated into the identifier "this", which has no special meaning in MiniZinc. Instead, it is simply a quantified variable defined in the **forall** expression in the prior Translation of Class-Level Constraints section.

Translation of Variables $[\![v]\!]$:

Within the scope of class c, the translation $[\![v]\!]$ of a variable v is:

when v is declared in class $c' \in c^*$

$c'\_v[\text{this}]$

otherwise

v

Variables in expressions are translated in one of two ways depending on their type. If the variable occurs within the scope of class c and was declared in c' which is either c or one of its ancestors, then the result is a lookup in the array corresponding to field v of the current instance (*i.e.*, **this**) of class c'. For example, where c' is DatabaseServer, and v is "role", the translation is:

```
DatabaseServer_role[this]
```

Otherwise, v is either a global variable or a quantified variable (from a fold), and translates directly to its identifier.

Translation of Folds:

The translation of a fold expression Fold (v **in** e_1 **where** e_2) (e_3) is given by:

Fold ($[\![v]\!]$ **in** r) (b)

where $r \triangleq$

when e_1 is of type $c[n]$

$1\,..\,count(c)$

when e_1 is of type $u[\,]$

$1\,..\,num(u)$

when e_1 is of type $\{i_1,\dots,i_n\}[\,]$

$\{i_1,\dots,i_n\}$

when e_1 is of type **bool**[ ]

$\{\textbf{true},\textbf{false}\}$

and $b \triangleq$

when Fold $=$ **sum**

bool2int(v **in** $[\![e_1]\!]$ `/\` $[\![e_2]\!]$) `*` $[\![e_3]\!]$

otherwise

v **in** $[\![e_1]\!]$ `/\` $[\![e_2]\!]$ `->` ($[\![e_3]\!]$)

Folds such as **foreach** translate to a similar construct in MiniZinc, but one in which the fold must be over a set of constant value, because the MiniZinc compiler unrolls the fold at compile-time. Therefore, the translation consists of a fold of an expression body b over a constant range r, which need not be contiguous.

The range r depends on the type of the expression e_1, which the fold is over. When it is a set of objects of type c, the range is the indices of all c instances. When the type is an enum, it is the valid indices of the enumeration. When the type is an integer subset the range is that subset.

Ranges are larger than one might expect. Because MiniZinc requires that ranges are constant, the range must contain all values of the relevant type, and we must correspondingly wrap the body expression e_3 with the implication, $v \in e_1 \Rightarrow e_3$, so that the constraint is placed over only members of the set in the current solution. In fact, because ConfSolve also allows a filter expression e_2, this becomes $v \in e_1 \wedge e_2 \Rightarrow e_3$.

The body expression b in fact takes two forms. For a **forall** or **exists**, the logical form just mentioned is used. For a **sum**, an arithmetic form is used: $bool2int(v \in e_1 \wedge e_2) \times e_3$, where *bool2int* returns a 0/1 value given a false/true boolean.

Translation of Binary Operators $[\![\text{BinOp}]\!]$**:**

`&&` $\triangleq$ `/\`

`||` $\triangleq$ `\/`

`/` $\triangleq$ **div**

BinOp$'$ $\triangleq$ BinOp$'$

Binary operators are directly translated to MiniZinc operators. BinOp$'$ denotes all operators not explicitly listed.

Reduction of Set Literal Expressions:

The reduction of a set literal expression $[e_1,\dots,e_n]$ is given by:

In the current scope, insert the declaration:

var set__s **as** T

Where T is a well-formed type which satisfies the (Set) typing judgement in section 4 and s is a unique integer.

In the current scope assume the constraint:

constraint e_1 **in** set__s $\wedge \cdots \wedge$ e_n **in** set__s

Finally, the derived expression is:

$[e_1,\dots,e_n] \triangleq$ set__s

The translation of set literal expression is defined in terms of a reduction to a variable and associated constraints at the ConfSolve level, which should be performed before any of the other transformation steps previously listed. This reduction is necessary as it allows variables to appear inside set literals, which would otherwise not be legal in MiniZinc.

Solve Statement:

when o is undefined

solve satisfy

otherwise

solve maximize o

The translation to MiniZinc concludes with the introduction of a solve statement, the purpose of which is to provide a criteria for the solver's search, which may be either a satisfaction of the constraints, or maximisation of an objective expression.

6.3 Solutions

After solving, the output of the ConfSolve post-processor is an object-tree, the syntax of which we refer to as CSON (ConfSolve Output Notation). A concrete example of CSON is given in Section 2

To obtain a solution, the translated MiniZinc is compiled into FlatZinc and solved using Gecode, which outputs assignments for each variable in a simple text-based format defined by FlatZinc. Generating a CSON tree from this text is straightforward: the steps of the translation process are repeated, but whenever a MiniZinc variable would be introduced, we instead read its value from the output file, and emit the corresponding CSON representation:

Syntax of CSON:

V ::=	value
i	integer
true | **false**	boolean
$u.a$	enum member
c {Member*}	object
ref Target	object reference
$T[n]\{V_1,\ldots,V_n\}$	set literal
Member ::=	member
$v : V$	variable name : value
Target ::=	target
v	variable
Target.l	field access
Target[i]	set access

Each CSON value corresponds to a type in ConfSolve. The solution output consists of a single anonymous object representing the global scope. Nested within this are values for each variable. In the special case of references, the value is the fully-qualified name of the target variable, in which members of sets may be accessed via index, for example $v[i].f$ resolves to the value of field f of the i^{th} element of the set v.

7 Evaluation

Our evaluation of ConfSolve aims to show that the system can be used to model a number of diverse configuration problems, and to successfully analyse them. Furthermore, we want to ensure that ConfSolve is able to perform adequately on large models.

The evaluation was performed on a machine with a 2GHz Intel Core i7 processor and 8GB of RAM, running Mac OS X version 10.8.1. We used the 64-bit MiniZinc to FlatZinc converter version 1.5.1 with the `--no-optimize` flag, and the 64-bit Gecode FlatZinc interpreter version 3.7.1.

7.1 Virtual Machine Placement

In this evaluation we use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration. Each physical machine is identical, having 8 CPUs and 16GB or memory. Each virtual machine has variables representing its requirements on the physical machine resources. These declarations are as follows:

```
class Machine {
  var cpu as int;     // 1 unit = 1/2 core
  var memory as int;  // MB
  var disk as int;    // GB

  cpu = 16;           // 2x Quad Core
  memory = 16384;     // 16 GB
  disk = 2048;        // 2 TB
}

abstract class VM {
  var host as ref Machine;
  var disk as int;
  var cpu as int;
  var memory as int;
}
```

Virtual machines may be one of two sizes, large and small. Large machines have 4 CPU units, 3.5GB of memory, and 500GB of disk. Small machines have 1 CPU unit, 768MB of memory and 20GB of disk:

```
class SmallVM extends VM {
  cpu = 1;
  memory = 768;
  disk = 20;
}

class LargeVM extends VM {
  cpu = 4;
  memory = 3584;
  disk = 500;
}
```

The infrastructure consists of two racks of 48 physical machines, onto which we wish to allocate 350 small and 100 large virtual machines:

```
// physical machine instances
var rack1 as Machine[48];
var rack2 as Machine[48];

// virtual machine instances
var smallVMs as SmallVM[350];
var largeVMs as LargeVM[100];
```

We define a constraint on virtual machine placement, as otherwise there is nothing to prevent every virtual machine from having the same host:

```
var machines as ref Machine[96];
var vms as ref VM[450];

forall m in machines {
  sum vm in vms where vm.host = m {
    vm.cpu;
  } <= m.cpu;

  sum vm in vms where vm.host = m {
    r.memory;
  } <= m.memory;

  sum vm in vms where vm.host = m {
    r.disk;
  } <= m.disk;
};
```

This constraint states that for each physical machine, the sum of the required quantity of each resource over all virtual machines hosted on it, must be less than the quantity of that resource provided by the physical machine. In other words, that the virtual machines assigned do not, in aggregate, consume more resources than are available. This is repeated for the three resources, `cpu`, `memory`, and `disk`.

From this model, ConfSolve is able to automatically generate assignments of virtual machines to physical machines (PMs), by automatically finding values for the `host` variable of each `VM` instance.

The performance achieved when scaling the problem up to 750 virtual machines is shown in Table 1. The problem size was increased until the Gecode solver consumed all available free memory on our test machine, which was 4.5GB.

Problem	ConfSolve	Cauldron [5]
VM Allocation 4:2	151	1717
VM Allocation 8:2	165	2115
VM Allocation 16:4	177	3995
VM Allocation 17:5	194	-
VM Allocation 100:48	1485	-
VM Allocation 250:48	8288	-
VM Allocation 450:96	44,700	-
VM Allocation 550:96	58,758	-
VM Allocation 650:96	77,174	-
VM Allocation 750:96	94,536	-

Table 1: VM : PM Allocation run-time (milliseconds), averaged over three runs.

7.2 Cauldron Test Suite

ConfSolve uses a similar object-oriented language to Cauldron [5] (see Section 8), a policy-based design tool which is able to describe CIM [9] models. Cauldron is able to generate solutions to these constraint-based system designs; an example of configuring an enterprise server with physical partitions is provided in [5].

HP Labs kindly provided us with a copy of the Cauldron binary (version `rel.10c`) and a number of sample problems from their test suite. The example described at length in [5] is very much representative in terms of scope and size, of the examples provided to us by HP. We translated a representative subset of these problems into ConfSolve models in order to confirm that ConfSolve can represent existing constraint-based configuration models. We then benchmarked the solution time of these equivalent ConfSolve and Cauldron models, the results of which can be found in Table 2. ConfSolve consistently out-performs Cauldron by a factor of around four on the test hardware described at the beginning of this section. The build of Cauldron which we were provided with makes use of a private build of the VeriFun [10] theorem prover in conjunction with a custom SAT solver which uses the LazySAT [11] algorithm. Given that Cauldron dates from 2007 and is no longer under development, more recent SAT solvers may provide a performance improvement, but Cauldron is a "black box" and we have no way to determine if this is the case.

As the Cauldron sample problems are relatively small in terms of search-space, we translated our large-scale VM example into an equivalent Cauldron model in order to compare performance at scale. The results in Table 1 show that, for this example at least, Cauldron does not scale to practical problem sizes, failing when only 17 virtual machines are to be allocated to 5 physical machines. ConfSolve was able to scale to 750 virtual machines.

Problem	ConfSolve	Cauldron [5]
Geometry	93	327
Firewall	145	504
QM4	394	2077
ServerComplex7	652	3149
ServerComplex10	875	4254

Table 2: Cauldron test suite run-time (milliseconds), averaged over three runs

8 Related Work

ConfSolve is, to the best of our knowledge, the first declarative configuration language to target a CSP solver (Cauldron targets a SAT solver). CSP is well suited to solving finite combinatorial problems, and branch-and-bound optimisation, which makes it a natural match for configuration problems.

Non-declarative policy languages with eventcondition-action (ECA) semantics, such as Ponder, have been used to describe network configuration problems [12], however the success of declarative system configuration tools such as Cfengine [1] has led to an interest in producing declarative policy tools for system administrators. Couch & Glifix were early experimenters with using Prolog for this purpose [13], followed by Narain [14], and more recently Yin [15]. Couch & Glifix conclude that a declarative language is essential for producing a convergent policy, due to the fact that an ECA language would require a policy to handle every possible failure scenario, but that Prolog is not a suitable language for system administration.

SAT solvers were used by Narain to solve network configuration problems [16, 6], via means of the the Alloy [17] modelling system and Kodkod [18] SAT-based relational logic solver. However, in both cases performance issues were encountered with the conversion of their model to SAT, and the difficulty of modelling was high: writing new predicates required expert knowledge of their modelling system's internals.

The Alloy Analyzer [17] is a mature modelling system which shares some commonality with ConfSolve: both provide an object-oriented specification language with logical constraints, and both require the user to specify an upper-bound on the number of objects in the search space. The Alloy modelling language is significantly more general than that of ConfSolve: modelling the dynamic properties of a system, *i.e.* its states, is a key feature, and its modelling language is necessarily far more general than ConfSolve, which is concerned only with the goal state of a system. As a system for model checking and verification, Alloy does not support optimisation of models, unlike MiniZinc, and in turn ConfSolve. This makes Alloy unsuitable as a backend for ConfSolve.

A notable SAT-based work is Cauldron [5], an object-oriented configuration language based on the CIM [9] model of classes, object references, and arrays, which is similar to the model adopted by ConfSolve. Solutions are generated using the VerifFun theorem prover, which itself relies on a SAT solver. Unfortunately, the Cauldron language is not rigorously defined, whether or not its search is complete is unclear, and its translation to SAT is not disclosed. Furthermore, its implementation does not scale well to practically-sized problems, such as those in Section 7.1.

PoDIM [19] is a framework for the Eiffel language which allows constraints on objects to be described with a SQL-like syntax. The language lacks a clear definition beyond its Eiffel implementation, which does not scale to problems of practical size.

An extension of the SmartFrog configuration language with constraints is hypothesised in [3], though no detailed work has been published.

s-COMMA [20] is a general-purpose object-oriented CP language which directly targets Gecode and other CP solvers, however it does not provide object references, variable cardinality sets or arrays, or quantification over decision variables.

Work towards developing ConfSolve is discussed in the workshop paper [21], which outlines an earlier, less efficient MiniZinc encoding.

9 Conclusion and Future Work

We have developed an object-oriented system configuration language in which constraints are used to specify valid configurations. We define its translation to the standard constraint modelling language MiniZinc, and find solutions to these models using a state-of-the-art constraint solver. Writing a complex object-oriented model in ConfSolve is considerably simpler than implementing the corresponding problem directly in MiniZinc. While the scalability of any constraint-based model is problem-specific, we have shown that ConfSolve models are able to scale to problems of a practical size using a common use-case.

While ConfSolve allows the user to model and analyse systems, it does not replace existing declarative configuration systems such as Cfengine or Puppet, but instead provides a platform which in the future could be used to augment existing languages, construct new tools which suggest solutions to specific problems, or perform impact analyses of proposed configuration changes.

It is possible to solve constraint problems at very large scale by performing a local search, which only explores a subset of the search-space, but can solve constraint problems many orders of magnitude larger. Local-search solvers such as [22] may soon support modelling with MiniZinc-like expressivity, which would allow them to be readily targeted by ConfSolve.

Implementation The ConfSolve compiler (v0.6) is written in F# / OCaml and is available for download at `http://homepages.inf.ed.ac.uk/s0968244/confsolve`

Acknowledgements.

Thanks to Sharad Singhal at HP Labs for providing the Cauldron test suite and binaries. This work was funded by Microsoft Research through their European PhD Scholarship Programme.

References

[1] Burgess, M.: Cfengine: a site configuration engine. USENIX Computing systems **8**(3) (1995) 309–402

[2] Puppet Labs: Puppet (2008) Available from `http://www.puppetlabs.com/puppet/`.

[3] Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., Toft, P.: The SmartFrog configuration management framework. SIGOPS Operating Systems Review **43** (January 2009) 16–25

[4] Oppenheimer, D., Ganapathi, A., Patterson, D.: Why do Internet services fail, and what can be done about it? In: 4th USENIX Symposium on Internet Technologies and Systems, USENIX Association (2003)

[5] 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), IEEE Computer Society (2006) 113–122

[6] Narain, S., Levin, G., Malik, S., Kaul, V.: Declarative infrastructure configuration synthesis and debugging. Journal of Network and Systems Management **16**(3) (2008) 235–258

[7] Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: 13th International Conference on Principles and Practice of Constraint Programming (CP 2007), Springer (2007) 529–543

[8] Gecode Team: Gecode: Genetic constraint development environment (2006) Available from `http://www.gecode.org`.

[9] Distributed Management Task Force Inc.: Common information model (CIM) standards (2010) Available from `http://www.dmtf.org/standards/cim/`.

[10] Walther, C., Schweitzer, S.: About VeriFun. In: Proceedings of the 19th International Conference on Automated Deduction (CADE-19). Volume 2741 of Lecture Notes in Computer Science., Springer (2003) 322–327

[11] Singla, P., Domingos, P.: Memory-efficient inference in relational domains. In: Proceedings of the 21st national conference on Artificial intelligence. Volume 1 of AAAI'06., AAAI Press (2006) 488–493

[12] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY '01). (2001) 18–38

[13] Couch, A., Gilfix, M.: It's elementary, dear Watson: applying logic programming to convergent system management processes. In: Proceedings of the 13th conference on Large Installation System Administration (LISA '99), USENIX Association (1999)

[14] Narain, S., Cheng, T., Coan, B., Kaul, V., Parmeswaran, K., Stephens, W.: Building autonomic systems via configuration. In: Proceedings of IEEE Autonomic Computing Workshop. (2003)

[15] Yin, Q., Cappos, J., Baumann, A., Roscoe, T.: Dependable self-hosting distributed systems using constraints. In: Proceedings of the Fourth Conference on Hot Topics in System Dependability, USENIX Association (2008) 11–11

[16] Narain, S.: Network configuration management via model finding. In: Proceedings of the 19th conference on Large Installation System Administration (LISA '05), USENIX Association (2005) 15

[17] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(2) (2002) 256–290

[18] Torlak, E., Jackson, D.: Kodkod: A relational model finder. Tools and Algorithms for the Construction and Analysis of Systems (2007) 632–647

[19] Delaet, T., Joosen, W.: PoDIM: A language for high-level configuration management. In: Proceedings of the 21st conference on Large Installation System Administration (LISA '07), USENIX Association (2007)

[20] Soto, R., Granvilliers, L.: On the pursuit of a standard language for object-oriented constraint modeling. (2008) 123–133

[21] Hewson, J., Anderson, P.: Modelling system administration problems with CSPs. In: Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef'11). (2011) 73–82

[22] Benoist, T., Estellon, B., Gardi, F., Megel, R., Nouioua, K.: Localsolver 1.x: a black-box local-search solver for 0-1 programming. 4OR: A Quarterly Journal of Operations Research (2011) 1–18