THE UNIVERSITY *of* EDINBURGH
informatics

cisa | Centre for Intelligent Systems and their Applications

Microsoft Research PhD Scholarship

Configuration Inference using CSPs

John. A Hewson, *University of Edinburgh*
john.hewson@ed.ac.uk

Supervisor: Paul Anderson, *University of Edinburgh*
dcspaul@ed.ac.uk

The complexity of typical computing installations has increased to the point where automated configuration is desirable. We developed a high-level, declarative specification language (ConfSolve) for system configuration, from which valid configurations are inferred via compilation of the specification into a Constraint Satisfaction Problem (CSP).

1 Background

This is a complete example of a simplified datacentre configuration problem, where a set of four heterogeneous services need to be allocated onto three heterogeneous machines. All ConfSolve source code is shown.

2 Primitives

Each machine has a set of up to six capabilities, which may or may not be present. Likewise ach service will require certain capabilities in order to run. Below, we define an enumeration of capabilities, and declare a new primitive set type, with cardinality from zero to six.

```
enum Capability {
   IsIISEnabled, IsSQLEnabled, HasDualProc,
   HasQuadProc, HasRAID5, HasGigEther
}

primitive Capabilities extends Capability[0..6] {
}
```

3 Classes

ConfSolve is object-oriented, and provides single inheritance. We need to declare classes to describe both machines and services, shown below. Note the *runs_on* variable which is declared as an object *ref*erence.

```
class Machine {
   var cpu as int;
   var memory as int;
}

class Service {
   var required_cpu as int;
   var required_memory as int;
   var required_capabilities as Capabilities;
   var runs_on as ref Machine;
}
```

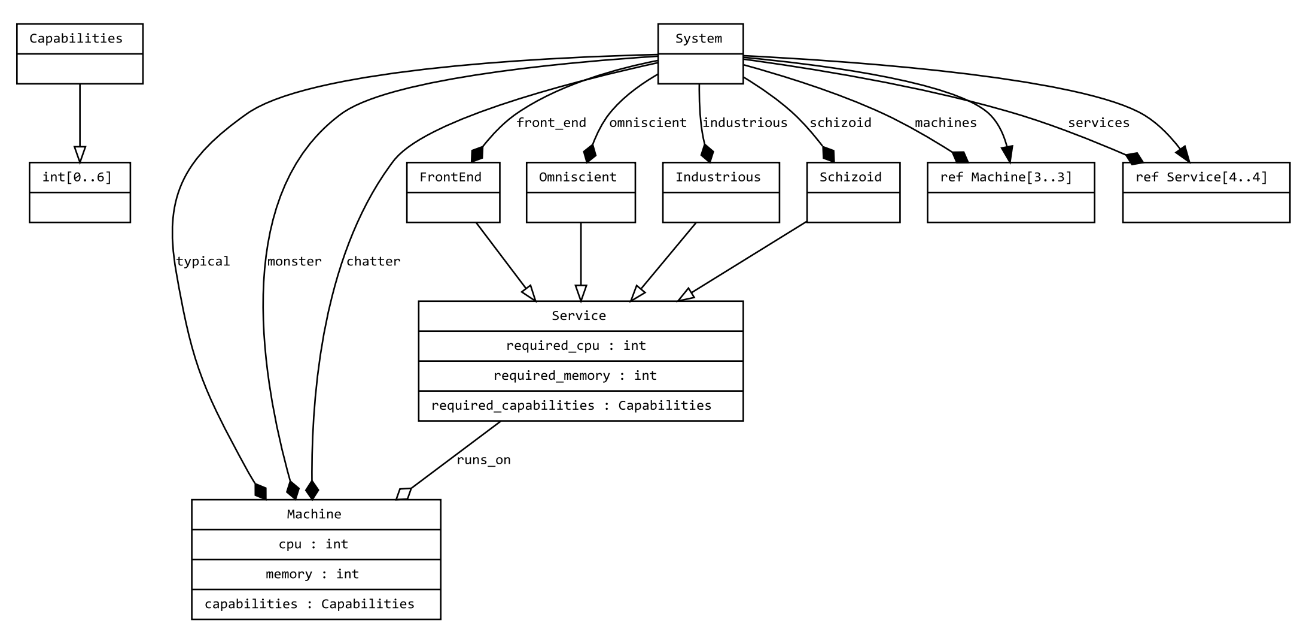

Figure 1 – UML Class diagram of the completed specification

Next, the three machines are declared as subtypes of the *Machine* class, Figure 1 shows a UML class diagram.

```
class Typical extends Machine {
   where cpu == 3;
   where memory == 2048;
   where capabilities == { Capability.HasDualProc,
                           Capability.HasRAID5 };
}

class Monster extends Machine {
   where cpu == 12;
   where memory == 16384;
   where capabilities == { Capability.IsIISEnabled,
                           Capability.HasDualProc,
                           Capability.HasQuadProc,
                           Capability.HasRAID5,
                           Capability.HasGigEther };
}

class Chatter extends Machine {
   where cpu == 12;
   where memory == 16384;
   where capabilities == { Capability.IsIISEnabled,
                           Capability.HasGigEther };
}
```

Likewise, the four services are declared as subtypes of the *Service* class.

```
class FrontEnd extends Service {
   where required_cpu == 1;
   where required_memory == 512;
   where required_capabilities == { Capability.IsIISEnabled,
                                    Capability.HasGigEther };
}

class Omniscient extends Service {
   where required_cpu == 6;
   where required_memory == 4096;
   where required_capabilities == { Capability.IsSQLEnabled,
                                    Capability.HasRAID5 };
}

class Industrious extends Service {
   where required_cpu == 1;
   where required_memory == 512;
   where required_capabilities == { Capability.HasDualProc };
}

class Schizoid extends Service {
   where required_cpu == 2;
   where required_memory == 1024;
   where required_capabilities == { Capability.HasDualProc };
}
```

4 Root Class

Now that the class structure has been declared, we need to define some instances of the classes. This is done in a special *root* class, called *System*. The root class is a singleton and will be automatically instantiated.

```
root class System {
    var typical as Typical;
    var monster as Monster;
    var chatter as Chatter;

    var front_end as FrontEnd;
    var omniscient as Omniscient;
    var industrious as Industrious;
    var schizoid as Schizoid;

    ...
}
```

5 Constraints

The description of the system objects is now complete. But which instantiations are valid? We need to specify some constraints over the variables declared in *System*.

We need to ensure that when a service *runs_on* a given machine, that the machine provides the required capabilities, and that the amount of CPU and RAM consumed by other services running on the same machine does not exceed that provided.

To express this succinctly, two set variables (of references) are declared, which will be resolved automatically at runtime:

```
var machines as (ref Machine)[3];
var services as (ref Service)[4];

forall (m in machines, s in services) {
    if (s.runs_on == m) {
        sum (s.required_cpu) <= m.cpu;
        sum (s.required_memory) <= m.memory;
        s.required_capabilities subset m.capabilities;
    }
}
```

6 CSP Generation

The specification is now complete. The ConfSolve compiler is invoked, and the specification is compiled into a Constraint Satisfaction Problem (CSP) expressed in the MiniZinc language (a standard cross-solver format). MiniZinc is not object-oriented, and the generated code is therefore more complex. A snippet of this code is given in Figure 2, below.

```
include "globals.mzn";
set of int : Int = -65535..65535;

/* variables */
var Int : root_typical_cpu = 3; /* root.typical.cpu */
var Int : root_typical_memory = 2048; /* root.typical.memory */
var set of Int : root_typical_capabilities; /* root.typical.capabilities */

...

var Int : root_front__end_required__cpu = 1; /* root.front_end.required_cpu */
var Int : root_front__end_required__memory = 512; /* root.front_end.required_memory */
var set of Int : root_front__end_required__capabilities; /* root.front_end.required_capabilities */

...

var set of {1, 2, 3} : root_machines; /* root.machines */
var set of {4, 5, 6, 7} : root_services; /* root.services */

var {1, 2, 3} : root_front__end_runs__on; /* root.front_end.runs_on */
var {1, 2, 3} : root_omniscient_runs__on; /* root.omniscient.runs_on */
var {1, 2, 3} : root_industrious_runs__on; /* root.industrious.runs_on */
var {1, 2, 3} : root_schizoid_runs__on; /* root.schizoid.runs_on */

var Int : sum_s_required__cpu_402_m1_s1; /* sum.s.required_cpu.402.m1.s1 */
var Int : sum_s_required__memory_402_m1_s1; /* sum.s.required_memory.402.m1.s1 */
var Int : sum_s_required__cpu_402_m1_s2; /* sum.s.required_cpu.402.m1.s2 */
var Int : sum_s_required__memory_402_m1_s2; /* sum.s.required_memory.402.m1.s2 */
var Int : sum_s_required__cpu_402_m1_s3; /* sum.s.required_cpu.402.m1.s3 */
var Int : sum_s_required__memory_402_m1_s3; /* sum.s.required_memory.402.m1.s3 */
var Int : sum_s_required__cpu_402_m1_s4; /* sum.s.required_cpu.402.m1.s4 */
var Int : sum_s_required__memory_402_m1_s4; /* sum.s.required_memory.402.m1.s4 */
var Int : sum_s_required__cpu_402_m2_s1; /* sum.s.required_cpu.402.m2.s1 */

...

/* constraints */
constraint (card(root_typical_capabilities) <= 6;;
constraint (card(root_monster_capabilities) <= 6);
constraint (card(root_chatter_capabilities) <= 6);
constraint (card(root_front__end_required__capabilities) <= 6);
constraint (card(root_omniscient_required__capabilities) <= 6);
constraint (card(root_industrious_required__capabilities) <= 6);
constraint (card(root_schizoid_required__capabilities) <= 6);
constraint (card(root_machines) >= 3);
constraint (card(root_services) >= 4);

/* System */
constraint (card(root_typical_capabilities) = 2);
/* System */
constraint (3 in root_typical_capabilities);
/* System */
constraint (5 in root_typical_capabilities);
/* System */
constraint (card(root_monster_capabilities) = 4);

...

/* forall: sum */
constraint (sum_s_required__cpu_402_m1_s1 =
        (((((root_front__end_required__cpu * bool2int((4 in root_services))) * bool2int((root_front__end_runs__on = 1))) +
        ((root_omniscient_required__cpu * bool2int((5 in root_services))) * bool2int((root_omniscient_runs__on = 1)))) +
        ((root_industrious_required__cpu * bool2int((6 in root_services))) * bool2int((root_industrious_runs__on = 1)))) +
        ((root_schizoid_required__cpu * bool2int((7 in root_services))) * bool2int((root_schizoid_runs__on = 1)))));

/* forall: sum */
constraint (sum_s_required__memory_402_m1_s1 =
        (((((root_front__end_required__memory * bool2int((4 in root_services))) * bool2int((root_front__end_runs__on = 1))) +
        ((root_omniscient_required__memory * bool2int((5 in root_services))) * bool2int((root_omniscient_runs__on = 1)))) +
        ((root_industrious_required__memory * bool2int((6 in root_services))) * bool2int((root_industrious_runs__on = 1)))) +
        ((root_schizoid_required__memory * bool2int((7 in root_services))) * bool2int((root_schizoid_runs__on = 1)))));

/* forall: sum */
constraint (sum_s_required__cpu_402_m1_s2 =
        (((((root_front__end_required__cpu * bool2int((4 in root_services))) * bool2int((root_front__end_runs__on = 1))) +
        ((root_omniscient_required__cpu * bool2int((5 in root_services))) * bool2int((root_omniscient_runs__on = 1)))) +
        ((root_industrious_required__cpu * bool2int((6 in root_services))) * bool2int((root_industrious_runs__on = 1)))) +
        ((root_schizoid_required__cpu * bool2int((7 in root_services))) * bool2int((root_schizoid_runs__on = 1)))));

...
```

Figure 2 – A sample of the generated *MiniZinc* code. The actual file has 378 lines in total.

7 CSP Solving

The CSP, expressed in MiniZinc, is then solved using the *Gecode* solver (although others may be used), which finds all four solutions to this problem in 4ms. Alternatively, we can choose to find just a single solutions, or as many as possible in a fixed period of time, which is more feasible for very large problems in the future.

8 Solutions

The output of Gecode is a simple text-based description of the variable assignments. These are parsed by ConfSolve and used to populate the existing object-oriented model of the system, including assignments of primitive values, and object references which the solver has calculated. Figure 3 shows a UML instance diagram for solution #1.

9 Output Generators

Finally, the fully populated object model is used to generate configuration files. In order to configure any system, we must be able to produce any output format. The solution is to provide an interface to the in-memory object model, available via C# and JavaScript. We developed an output generator which produces XML, using just over thirty lines of JavaScript.

10 Future Work

We intend on scaling up the problem size to find the limits of current CSP solvers. We are considering investigating distributed constraints, and possibly using SMT as a solver backend.

Acknowledgements

This work was supported by Microsoft Research through its European PhD Scholarship Programme.

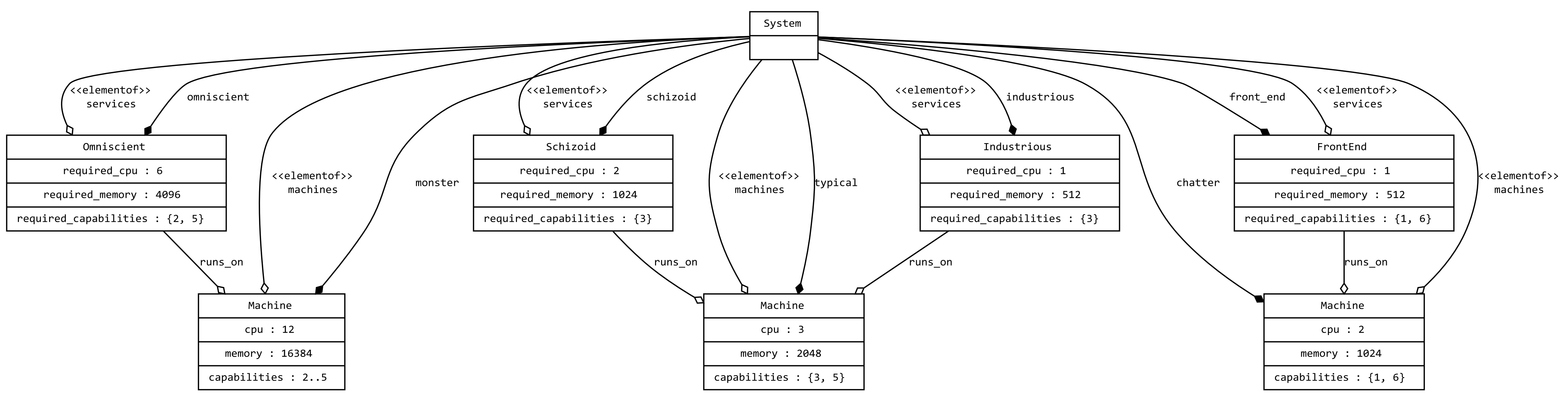

Figure 3 – UML instance diagram of solution #1. The *runs_on* references have been resolved by the CSP solver.