Santa's Reindeer Challenge - A solution using DMN and BPMN

Rob Parker, 11 Dec 2017

Problem

The DMCommunity.org has created a Christmas challenge. The challenge is replicated below;

Santa always leaves plans for his elves to determine the order in which the reindeer will pull his sleigh. This year, for the European leg of his journey, his elves are working to the following schedule, which will form a single line of nine reindeer. Here are the rules:

1. *Comet behind Rudolph, Prancer and Cupid*
2. *Blitzen behind Cupid*
3. *Blitzen in front of Donder, Vixen and Dancer*
4. *Cupid in front of Comet, Blitzen and Vixen*
5. *Donder behind Vixen, Dasher and Prancer*
6. *Rudolph behind Prancer*
7. *Rudolph in front of Donder, Dancer and Dasher*
8. *Vixen in front of Dancer and Comet*
9. *Dancer behind Donder, Rudolph and Blitzen*
10. *Prancer in front of Cupid, Donder and Blitzen*
11. *Dasher behind Prancer*
12. *Dasher in front of Vixen, Dancer and Blitzen*
13. *Donder behind Comet and Cupid*
14. *Cupid in front of Rudolph and Dancer*
15. *Vixen behind Rudolph, Prancer and Dasher.*

Approach

With this type of problem I would typically use a search based on variable domain narrowing and constraint propagation. However, in this case I wanted to use a DMN model. Hence I took a different approach. This approach uses a brute force search to enumerate all arrangements in turn, testing each arrangement against the set of rules until a valid arrangement is found. Hence, whilst not the most elegant or efficient solution, it is a working solution based on DMN.

Model

I've modelled the order of reindeer as a sequential set of integers from 0 to 8. Hence if 0 represents the lead reindeer and 8 represents the last reindeer, the constraints can be represented as;

A in front of B, implies A<B

C behind D and E, implies C>D and C>E

Thus each row in the DMN table (shown further below) encapsulates the solution constraints along these lines.

Using the brute force approach meant that my decision table must take a solution configuration and return either true or false based on the validity of the solution. Hence I used a Collect hit policy such that I could coalesce the results of the set of constraints into a single outcome. This required that for each constraint, I had to introduce an additional (complementary) rule indicating if the original business rule was violated. In addition, given the hit policy, this approach prevents partial solutions from passing a subset of rules and thus producing a false positive.

Each rule is given a single integer result. The original business rules if met will return a 1. The complementary introduced rules will return a 0 if met. Thus the decision table requires 30 rules. The Collect hit policy is set to return the minimum value for the result set. Hence if any of the complementary rules have a hit, then a zero will be returned by the rule and thus the collect hit policy. Hence only if a value of 1 is returned via the hit policy is the solution valid. The full DMN table is shown below.

Figure 1 - The full DMN table – 30 rules comprising the 15 given constraints and the 15 complementary constraints.

A zoomed in snippet of the DMN table is shown below. As shown in rows 1 and 2, row 1 represents the rule that Comet must be behind Rudolph, Prancer and Cupid. Hence the ordinal number of Rudolph must be less than that of Comet, so must Prancer and so on for Cupid. The output variable 'Solved' is set to 1 if this condition is met. The complementary rule (row 2) will be matched if the ordinal number of Comet is less than or equal to any one of the ordinal numbers of Rudolph, Prancer or Cupid. If any of these conditions are met, then the rule will be matched and an output value of zero will be returned for that particular rule (which indirectly indicates that rule 1 was not met and thus this is an invalid configuration).

Reindeer — Enter Advanced Mode

C<	Input									Output	
	Comet	Rudolph	Prancer	Cupid	Blitzen	Donder	Vixen	Dancer	Dasher	Solved	Annotation
	integer	integer	integer	integer	integer	integer	integer	integer	integer	integer	
1		<Comet	<Comet	<Comet						1	Comet behind Rudolph, Prancer and Cupid
2	<= Rudolph, <= Prancer, <=Cupid									0	Failed above
3					>Cupid					1	Blitzen behind Cupid
4					<= Cupid					0	Failed above
5						> Blitzen	> Blitzen	> Blitzen		1	Blitzen in front of Donder, Vixen and Dancer

Figure 2 Decision Table snippet showing a subset of rules in greater resolution.

As shown in the top left corner of the decision table, the hit policy is set to 'C' for collect. In particular, the hit policy is configured to return the minimum value from the set of 'Solved' variables generated by rules which match the input values. Thus if any complementary rule is matched, indicating a violation of the given constraints, a zero value will be amongst the set of 'Solved' variables. Hence if the hit policy returns a zero, then the solution is not valid. If the solution is valid, then only the 15 given constraints should match, and thus the set of 'Solved' variables should contain all ones. In this case the hit policy will return one indicating the given inputs are a valid configuration.

Approach

To solve the problem and drive the decision table, I used a simple BPMN process with a script task to call the decision model. I implemented this using the Camunda BPMN and DMN engine implementation. The script task uses a set of nested loops to generate all possible permutations. Each permutation is evaluated by the decision engine and the script continues until a valid permutation is found. Whilst not elegant or efficient, the valid solution is found within a few seconds or so on a basic laptop.

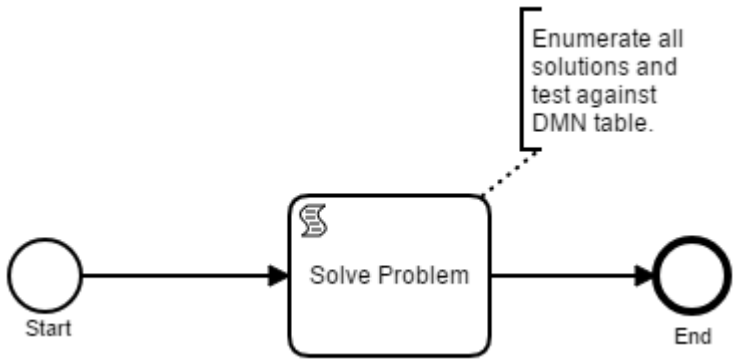

Figure 3 Simple BPMN driver process

Sample groovy script from the script task is shown below. The approach is to generate all combinations of ordering arrangements. For each arrangement, pass it to the decision service and test the result. If the result is valid, then print the current configuration.

```
import org.camunda.bpm.engine.ProcessEngineServices;
import org.camunda.bpm.engine.DecisionService;

//
// lookup the engine services from the BPMN execution service to access the Decision Service
//
def services = execution.getProcessEngineServices();
def decisionService = services.getDecisionService();

//
// define a map for storing the solution configuration
//
def map = [:];

//
// Enumerate all combinations. Given each reindeer must be given a unique number, continue
past duplicate numbers
//
for (Comet = 0; Comet < 9; Comet++) {
    map.put("Comet", Comet);
    for (Rudolph = 0; Rudolph < 9; Rudolph++) {
        if (Rudolph == Comet)
            continue;
        map.put("Rudolph", Rudolph);
        for (Prancer = 0; Prancer < 9; Prancer++) {
            if (Prancer == Rudolph || Prancer == Comet)
                continue;
            map.put("Prancer", Prancer);
            for (Cupid = 0; Cupid < 9; Cupid++) {
                if (Cupid == Prancer || Cupid == Rudolph || Cupid == Comet)
                    continue;
                map.put("Cupid", Cupid);
                for (Blitzen = 0; Blitzen < 9; Blitzen++) {
                    if (Blitzen == Cupid || Blitzen == Prancer || Blitzen == Rudolph ||
Blitzen == Comet)
                        continue;
                    map.put("Blitzen", Blitzen);
                    for (Donder = 0; Donder < 9; Donder++) {
                        if (Donder == Blitzen || Donder == Cupid || Donder == Prancer ||
Donder == Rudolph || Donder == Comet)
                            continue;
                        map.put("Donder", Donder);
                        for (Vixen = 0; Vixen < 9; Vixen++) {
                            if (Vixen == Donder || Vixen == Blitzen || Vixen == Cupid || Vixen
== Prancer || Vixen == Rudolph || Vixen == Comet)
                                continue;
                            map.put("Vixen", Vixen);
                            for (Dancer = 0; Dancer < 9; Dancer++) {
                                if (Dancer == Vixen || Dancer == Donder || Dancer == Blitzen
|| Dancer == Cupid || Dancer == Prancer || Dancer == Rudolph || Dancer == Comet)
                                    continue;
                                map.put("Dancer", Dancer);
                                for (Dasher = 0; Dasher < 9; Dasher++) {
                                    if (Dasher == Dancer || Dasher == Vixen || Dasher ==
Donder || Dasher == Blitzen || Dasher == Cupid || Dasher == Prancer || Dasher == Rudolph ||
Dasher == Comet)
                                        continue;
                                    map.put("Dasher", Dasher);

                                    //
                                    // Call the decision engine with the current configuration
and test the result for solved or not?
                                    //
                                    def result =
decisionService.evaluateDecisionTableByKey("Reindeer", map);

                                    //
                                    // If solved, print the current configuration...
```

```
                                        //
                                        if (result.getSingleEntry() == 1) {
                                            println "Solved!!!";
                                            println map;

                                        }
                                    }

                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 4 Groovy script to generate all possible configurations and test against the decision table.

Results

The implementation finds a single valid solution within a few seconds. The sample groovy script will continue until all combinations are tested and exhausted. In this case only one valid solution is found as follows;

[Comet:6, Rudolph:2, Prancer:0, Cupid:1, Blitzen:4, Donder:7, Vixen:5, Dancer:8, Dasher:3]

An interesting future direction could be to implement an approach more akin to constraint satisfaction approaches using DMN…