A case study of constraint programming for configuration problems*

Kevin McDonald and Patrick Prosser

Department of Computing Science, University of Glasgow, Scotland.
pat@dcs.gla.ac.uk

Abstract. Today, many universities are opting for modular degree programmes. Such modular courses provide greater flexibility for students. However, such a system is naturally complex; modules may feature pre and co-requisites and may run over different periods of times, and have different credit values. General university requirements will need to be met by students to continue their studies. Add to this timetable constraints and the module selection process can be a daunting task. Students may further complicate the process by explicitly wanting to take or avoid modules. They may require a general overview to see what options are available to them, such as the different routes to a particular degree. The University of Glasgow currently has no automated process to help with this. This paper describes our efforts in applying constraint programming to this configuration problem. We show how we went about tackling the problem using the constraint programming language Choco. We present a small example problem, a constraint programming model of this problem, and describe how we deliver explanations. We then present an extension of this model to deal with dynamic problems, where variables and constraints can be activated as a result of decisions made by the user or search process. Throughout this study, our goal has been to keep it simple, attempting to show that an off-the-shelf constraint programming toolkit is up to the task.

1 Introduction

Students design their own degrees. A university degree (for example Computer Science) is typically composed of a set of modules, each corresponding to a specific subject, such as databases, algorithms and data structures, communications, etc. Each of these modules is worth a certain amount of credits. To progress from one year to the next a student must accumulate a minimum amount of credits. In a year of study there will be a set of modules. Typically, a subset of these will be core modules that must be taken. Additional modules must then be selected to achieve the minimum number of credits. This selection may then influence subsequent modules that can, and cannot, be taken in later years. For example, a student cannot take a third year course on algorithms if she has not taken a second year course on discrete mathematics i.e. discrete mathematics is

* This work was supported by EPSRC research grant GR/M90641

a pre-requisite. Similarly, certain modules must be taken together i.e. they are co-requisites. Added to this, there is a limit to the number of modules a student can take in any year, and sometimes in any term.

Students are then faced with a daunting task. They may decide that they want a certain degree, say Software Engineering, yet there are certain modules that they do and do not want to take, and some terms that they want to minimise the number of modules that they take. What modules should they take, and what are the consequences? Much time can be spent by advisors of study assisting students in deciding what to study, and explaining why certain modules cannot be taken i.e. the advisors help the students design their course. Although all this information is available in the university handbook, it may require an expert to interpret it.

What we attempt here, is to demonstrate that the task of advising a course of study is essentially a problem of design, and that a constraint based model is most appropriate. We demonstrate this by presenting the 4th year curriculum from our department, an encoding of this in the Choco constraint programming toolkit, and sample queries that a student may ask of such a model. However, this problem is essentially static; all the variables of the problem are active and must be assigned values. We show how we can readily extend our model to address this, by using don't care values allied to variable and value ordering heuristics. Through out this study, our goal is to demonstrate that design problems can be easily modelled using an off-the-shelf constraint programming toolkit. We make no claims for efficiency, only that we expect that modest sized systems can be fielded with relative ease.

In the next section we introduce the problem of designing a fourth year of study in computer science, and show how we can model this using a constraint programming toolkit. We then show typical queries and how we deliver explanations. In section 5 we extend this model to deal with dynamic problems, where variables and constraints become active as the result of decision making. Section 6 concludes this study.

2 A Fourth Year Problem

The 4th year of study is split across two semesters. All students must do an individual final year project (proj) and complete the module on professional issues (pi). Students have then to take 8 other modules, selected from the following options:

1. Formal Methods (fm) semester 1
2. Information Retrieval (ir) semester 1
3. Security & Cryptography (sc) semester 1
4. Advanced Communications (ac) semester 2
5. Artificial Intelligence (ai) semester 2
6. Algorithmics (al) semester 2
7. Computer Architecture (ca) semester 1
8. Databases & Information Systems (dbis) semester 1

9. Design & Evaluation of Multimedia Systems (dems) semester 2
10. Issues in Collaborative & Distributed Systems (hci) semester 1
11. Modelling Reactive Systems (mrs) semester 2
12. Neural Computing (nc) semester 2
13. Network Communications Technology (nct) semester 1
14. Requirements Engineering & Re-engineering (rer) semester 2
15. Safety Critical Systems (scs) semester 1
16. Synthetic Graphics (sg) semester 2

In the first semester a student must take 4 or 5 modules. The semester 1 module Network Communications Technology (nct) is a pre-requisite for the Advanced Communications (ac) module in semester 2. This allows 8820 different ways for a student to fulfil the degree regulations for final year Computer Science[1].

In Figure 1 we represent the above curriculum as a constraint satisfaction problem [8] using the Choco toolkit [3]. The Choco function, level4(), delivers an object representing a constraint satisfaction problem composed of integer variables and constraints. Each module is a 0/1 variable, with a value of 1 if taken, 0 otherwise. The two compulsory modules, Professional Issues (pi) and Individual Project (proj), are represented for completeness so that a student is aware that they must be taken (i.e. assigned a value of 1). The constraint on line $\langle B \rangle$ represents the pre-requisite: Network Communications Technology (nct) is a pre-requisite for Advanced Communications (ac). The constraint on line $\langle C \rangle$ guarantees that either 4 or 5 modules are taken in the first semester, and the constraint on line $\langle D \rangle$ guarantees that either 3 or 4 modules are taken from the second semester. The final constraint $\langle E \rangle$ ensures that 8 modules are taken in total. Some of these constraints might initially appear superfluous, but as we will soon show, they are there to allow more interesting queries by the user.

3 Making Choices: an example

The above problem is not *solved* in the conventional sense; instead a user interacts with it. The interaction involves enforcing a decision and then seeing the consequences of this, asking for an explanation as to why certain choices are forced upon the user, or why certain choices are not available. We now present a typical sequence of decisions and queries.

Assume we have created the problem (i.e. p:Problem := level4()), and that we have (male) student X. X wants to get started on his project and reckons that he might do well to lighten his load in the first semester. In Choco, we create a new world (i.e. world+()), set sum1 to 4 (i.e. setVal(sum1,4)), and propagate this through the problem (i.e. propagate(p)). This will set sum2 to 4, forcing the student to take 4 modules in the second semester. Note that in creating a new world, we can manually retract the most recent decision by backtracking via the

[1] In fact, this is an under estimate. Typically in a module's exam, 2 questions have to be answered from 3. In addition, each student has to choose a final year project.

```
[level4() : Problem
 -> let pb   := makeProblem("Level 4",20),
        proj := makeIntVar(pb,"proj",0,1),
        pi   := makeIntVar(pb,"pi",0,1),
        fm   := makeIntVar(pb,"fm",0,1),
        dbis := makeIntVar(pb,"dbis",0,1),
        hci  := makeIntVar(pb,"hci",0,1),
        scs  := makeIntVar(pb,"scs",0,1),
        ca   := makeIntVar(pb,"ca",0,1),
        ir   := makeIntVar(pb,"ir",0,1),
        nct  := makeIntVar(pb,"nct",0,1),
        sc   := makeIntVar(pb,"sc",0,1),
        al   := makeIntVar(pb,"al",0,1),
        rer  := makeIntVar(pb,"rer",0,1),
        mrs  := makeIntVar(pb,"mrs",0,1),
        ai   := makeIntVar(pb,"ai",0,1),
        nc   := makeIntVar(pb,"nc",0,1),
        sg   := makeIntVar(pb,"sg",0,1),
        dems := makeIntVar(pb,"dems",0,1),
        ac   := makeIntVar(pb,"ac",0,1),
        must := list(proj,pi),
        sum1 := makeIntVar(pb,"sum1",list(4,5)),
        sum2 := makeIntVar(pb,"sum2",list(3,4)),
        sem1 := list(fm,dbis,hci,scs,ca,ir,nct,sc),
        sem2 := list(al,rer,mrs,ai,nc,sg,dems,ac)
    in (post(pb, sumVars(must) == 2),            // <A>
        post(pb, implies((ac == 1),(nct == 1))), // <B>
        post(pb,sumVars(sem1) == sum1),          // <C>
        post(pb,sumVars(sem2) == sum2),          // <D>
        post(pb,sum1 + sum2 == 8),               // <E>
        pb)]
```

Fig. 1. The 4th year of study, as a constraint program

function call world-(), and in the extreme we can return to our initial problem state via the call world=(0).

Student X does not want to take the first semester module Network Communication Technology (nct). Again, we create a new world, now moving to world 2. We set variable nct to 0 (i.e. setVal(nct,0)) and propagate this decision. Since Network Communication Technology is a pre-requisite for Advanced Communication (ac) the variable ac is set to 0 via propagation. Consequently, student X now has a reduced set of options in the second semester.

The above steps can be performed quite easily within the Choco interpreter with just a handful of functions, for selecting and setting variables. That is, as

a proof of concept we need not develop a user interface, but merely interact via the interpreter[2].

We could have encoded the above problem differently. In particular, we could have had a constraint stating that 4 or 5 modules must be taken in the first semester i.e. sumVars(sem1) == 4 OR sumVars(sem1) == 5, and similarly that 3 or 4 modules must be taken in the second semester i.e. sumVars(sem2) == 3 OR sumVars(sem2) == 4. However, this would not have allowed student X to make the decision that he will take 4 modules in semester 1. By doing away with the variable sum1, we no longer allow the student to make the strategic decision to spread his study load evenly over the two semesters. Clearly, our choice of model influences the kinds of decisions that a user can make.

4 Giving Explanations

In [4] Junker presents a simple and elegant method for delivering explanations for conflicts. Assume we have a sequence of decisions $S = d_1, d_2, ..., d_n$, where d_n is our last decision before we detect a conflict. Therefore we know that d_n must be one of the culprits. But what other decisions might be involved? Junker proposes that we retract all our decisions and then enforce d_n. If this results in a contradiction we have an explanation, i.e. d_n on its own. If this is not the case we then attempt to make the sequence of decisions $S \setminus \{d_n\}$, up to conflict. Assume that on making decisions $d_1 ... d_i$ we again have a conflict. We can then be sure that decisions d_n and d_i together are a subset of the culprit decisions. We then repeat this process, making decisions d_n and d_i, and then the sequence of decisions $S \setminus \{d_n, d_i\}$, again up to conflict, always adding the last decision that fails to the set of culprits. This set of culprits is then a sound and minimal explanation[3].

Junker's technique can be easily extended to cover the situation where we want to determine why propagation sets a specific variable to a specific value i.e. why student X must take a given module or cannot take a given module. We modify the above procedure such that rather than stopping when a contradiction is detected, we stop when a specified variable is set to a specified value. In fact, we can generalise even further, producing an explanation for the removal of a value, a set of values, the setting of a variable, etc.

We use this procedure to deliver explanations. We record all decisions made by the user in a history list H. When a user asks for an explanation, we return to our initial problem via the Choco function call world=(0). This returns us to the first world, where no decisions have been made. We then use the above procedure on the list H building up the list of culprits.

[2] And this might not be unreasonable considering that this would only be used by 4th year Computer Scientists.

[3] However, there may be many other explanations.

5 Dynamic Constraint Satisfaction

The dynamic constraint satisfaction problem (dcsp) is sometimes misunderstood i.e. there are a number of ways we might think of a csp as being dynamic. The first, and most obvious, is a csp where variables and constraints may be added and retracted from the problem after a solution has been found [1, 2, 6]. Such a task might be thought of as maintaining a problem. Another notion of dynamicity is proposed in [5] and more recently in [7]. Here we have a problem that involves initially two sets of variables. The set V_a is the set of active variables, and these variables are to be assigned values. The set V_i is the set of inactive variables, and they do not initially take part in problem solving. There are constraints between variables, and some of these constraints can *activate* variables in V_i such that they now participate in problem solving.

Mittal and Falkenhainer [5] propose a technique where there are functions to activate or deactivate constraints, and these are expressed within the constraint themselves. They also suggest an alternative approach, using don't care values in the domains of variables; when a variable is inactive it can take such a value.

Consider the following simple example of Figure 2, taken from Mittal and Falkenhainer.

```
variables
        v1 in {a,b}
        v2 in {c,d}
        v3 in {e,f}
        v4 in {g,h}

active variables
        Va = {v1,v2}

constraints
        v1 = a -> v2 = d
        v1 = b -> v2 = c
        v2 = c & v3 = e -> v4 = h
        v1 = b -> active(v3)
        v3 = e -> active(v4)

solutions
        {a,d,-,-}
        {b,c,f,-}
        {b,c,e,h}
```

Fig. 2. A dynamic csp

We see that if variable v1 takes the value b then v3 becomes active and must be assigned a value e or f. The 3 solutions are listed with a - meaning the variable is inactive, and therefore should not be instantiated.

The Figure 2 problem is presented as Choco code in Figure 3. Rather than use letters as domain values we use numbers, such that 1 substitutes for a, and 8 substitutes for h. Any variable that is not initially in the set V_a has an additional value DC in its domain (i.e. the don't care value). In our example, DC might have a value of -99. Rather than have a function to *activate* a variable we just remove the DC value from the domain. For example the constraint

```
v1 = b -> active(v3)

          becomes

v1 = b -> v3 <> DC
```

i.e. to activate a variable we post a conditional unary constraint.

However, this encoding alone does not prevent us enumerating unwanted solutions. To do this we must exploit variable and value ordering heuristics. When we select a variable for instantiation we prefer to select an active variable. That is, we select a variable that does not contain DC in its domain. If all the remaining (future) variables are inactive, i.e. each variable has DC in its domain, then the variable ordering heuristic performs an additional, out of character, function; it forces the instantiation of the variable to take the DC value.

```
[dcsp() : Problem
 -> let pb  := makeProblem("dcsp",4),
        v1 := makeIntVar(pb,"v1",{1,2}),
        v2 := makeIntVar(pb,"v2",{3,4}),
        v3 := makeIntVar(pb,"v3",{5,6,DC}),
        v4 := makeIntVar(pb,"v4",{7,8,DC})
    in (post(pb,implies(v1 == 1,v2 == 4)),
        post(pb,implies(v1 == 2,v2 == 3)),
        post(pb,implies(and(v2 == 3,v3 == 5),v4 == 8)),
        post(pb,implies(v1 == 2,v3 <> DC)),        // <A>
        post(pb,implies(v3 == 5,v4 <> DC)),        // <B>
        pb)]
```

Fig. 3. A dynamic csp using don't care and ordering heuristics

Variables v3 and v4 are initially inactive, with domains $\{5, 6, DC\}$ and $\{7, 8, DC\}$ respectively. The constraints in lines $\langle A \rangle$ and $\langle B \rangle$ correspond to the activation of variables v3 and v4. When a variable is inactive it will have in its domain the DC value, and when a variable becomes active the DC value is removed from its domain. Only active variables are selected for instantiation, or offered to the

user at a decision point. When no active variables remain, inactive variables are selected and assigned the DC value.

5.1 Dynamic Variables and Constraints

Our initial problem, the fourth year problem, is not dynamic. Variables correspond to modules, and these are either taken or not taken. There is only one set of variables, and these are all active. Consider the richer (but admittedly artificial) problem of Figure 4, where we have two options, A or B. If we select A then the sum of the variables a1, a2, and a3 must be equal to 4, and the variables b1, b2, and b3 should be inactive, i.e. the b* variables take the don't care value DC. Alternatively, if we select B the sum of the variables b1, b2, and b3 must be 4, and variables a1, a2, and a3 must remain inactive. Initially, only variables

```
[Pb() : Problem
 -> let pb   := makeProblem("Pb",8),
        A    := makeIntVar(pb,"A",list(0,1)),
        B    := makeIntVar(pb,"B",list(0,1)),
        a1   := makeIntVar(pb,"a1",list(0,1,DC)),
        a2   := makeIntVar(pb,"a2",list(1,2,DC)),
        a3   := makeIntVar(pb,"a3",list(1,2,DC)),
        b1   := makeIntVar(pb,"b1",list(0,1,DC)),
        b2   := makeIntVar(pb,"b2",list(1,2,DC)),
        b3   := makeIntVar(pb,"b3",list(1,2,DC)),
        la   := list(a1,a2,a3),
        lb   := list(b1,b2,b3)
    in (post(pb,1 - A == B),                        // <A>
        post(pb,implies(A == 1,sumVars(la) == 4)),// <B>
        post(pb,implies(A == 1,a1 !== DC)),        // <C>
        post(pb,implies(A == 1,a2 !== DC)),        // <D>
        post(pb,implies(A == 1,a3 !== DC)),        // <E>
        post(pb,implies(B == 1,sumVars(lb) == 4)),// <F>
        post(pb,implies(B == 1,b1 !== DC)),        // <G>
        post(pb,implies(B == 1,b2 !== DC)),        // <H>
        post(pb,implies(B == 1,b3 !== DC)),        // <I>
        pb)]
```

Fig. 4. A problem with dynamic variables and constraints

A and B are active. Therefore, initially a user can only make decisions on A or B. Assume we select variable A and set it to 1. Constraint $\langle A \rangle$ will force B to 0. Constraint $\langle B \rangle$ is now active, forcing the sum of variables a1, a2, and a3 to be equal to 4. Constraints $\langle C \rangle$, $\langle D \rangle$ and $\langle E \rangle$ now activate the a* variables. The b* variables remain inactive. Our variable and value ordering will now allow us to make decisions on the a* variables, and will ultimately instantiate all the b* variables to the don't care value.

When we have a constraint that involves variables that can be inactive we must condition them with the activating event. For example, in constraint $\langle B \rangle$ above the constraint sumVars(la) == 4 is only applied when A is set to 1, and constraints $\langle C \rangle$, $\langle D \rangle$ and $\langle E \rangle$ then activate the a* variables. That is, the constraint implies(A == 1,sumVars(la) == 4) is satisfied when A has a value of 0, and the a* variables can then take any values (although they will be forced to take the DC value due to our ordering heuristics). When A has a value of 1 the constraint can only be satisfied when sumVars(la) == 4, and constraints $\langle C \rangle$, $\langle D \rangle$ and $\langle E \rangle$ will by then have activated the a* variables.

6 Future Work and Conclusion

Within this department, there have been an number of failed attempts at producing a system that can be used to advise students on a course of study. These failed attempts tend to be dominated by efforts to capture the student handbook in a data base and allow access to it via a user interface. Essentially, such systems fail because they do not capture the dynamic effects of decision making. In this project our goal has been to produce a convincing demonstration of constraint programming as a solution to this problem. Our goal was not to produce a fully fledged system, but rather to produce a proof of concept. We believe that we have done that.

Clearly, the above test case (4th year Computer Science) is very small. Attempting to extend this to cover a faculty, let alone an entire university, is a huge task (for example, see van der Linden's PhD thesis [9]). However, we should expect that some day it will become a necessity, especially so as universities become more involved in distance learning.[4]

We are pleasantly surprised at the simplicity and effectiveness of Junker's explanation technique; this fits well with the above problem. As for dynamic constraint satisfaction, we believe that the proposed scheme of using don't care values, conditioned constraints, and variable and value ordering, allows a simple and effective solution to this problem. However, we make no claims as to the efficiency of such a scheme. As problems become large it may be unacceptable to maintain large sets of inactive variables and constraints. However, for small to medium sized problems, the emphasis is most probably ease of development and maintenance. We believe our scheme meets that bill.

Acknowledgements

We would like to thank the Computing Science department of Glasgow University for supporting this project, and in particular Alison Mitchell for patiently explaining the problem to us. We would also like to thank the OCRE project team for their help with Choco.

[4] One current example is lifelong learning in the Open University.

References

1. Amit Bellicha. Maintenance of a solution in a dynamic constraint satisfaction problem. *Applications of Artificial Intelligence in Engineering*, pages 261–274, 1993.
2. Christian Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proceeding of the Ninth National Conference on Artificial Intelligence*, pages 221–226, 1991.
3. CHOCO. http://www.choco-constraints.net/ home of the choco constraint programming system.
4. Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCA'01 workshop on Modelling and Solving Problems with Constraints*, pages 81–88, 2001.
5. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, 1990.
6. P. Prosser. A reactive scheduling agent. In *Proceedings of IJCAI-89*, 1989.
7. Timo Soininen and Esther Gelle. Dynamic constraint satisfaction in configuration. In *Proceedings of the AAAI99 workshop on Configuration, http://wwwold.ifi.uni-klu.ac.at/ alf/aaai99/*, 1999.
8. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
9. Janet van der Linden. An approach to dealing with non-standard constraint satisfaction problems. PhD Thesis, Oxford Brookes, 2000.