A Student Advisory System: a configuration problem for constraint programming [1]

Kevin McDonald and **Patrick Prosser** [2]

Abstract. Today, many universities are opting for modular degree programmes. Such modular courses provide greater flexibility for students, allowing them to learn about subjects previously inaccessible to them. Such a system is naturally complex. Modules may feature pre and co-requisites and may run over differently periods of times (and have different credit values). General university requirements will need to be met by students to continue their studies.

Students themselves may further complex 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. To find answers students may need to visit several different faculties to investigate possible module selections. Students may have to spend large periods of time with their adviser manually working through the university course catalogue finding what modules are and are not available to them. This paper describes our initial efforts in applying constraint programming to this problem.

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 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 (maybe so that they can complete a year's project). 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 have attempted 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.

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.

We represent the above curriculum as a constraint satisfaction problem [4] using the Choco toolkit [1]. This is shown in Figure 1. 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 B represents the pre-requisite: Network

[1] This work was supported by EPSRC research grant GR/M90641.
[2] Computing Science, Glasgow University,17 Lilybank Gardens, Glasgow G12 8RZ, Scotland, email: pat@dcs.gla.ac.uk

```
[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)]
```

Figure 1. level4(), a constraint programming encoding of the curriculum design problem for 4^{th} year Computer Science at Glasgow University

Communications Technology (nct) is a pre-requisite for Advanced Communications (ac). The constraint on line C guarantees that either 4 or 5 modules are taken in the first semester, and the constraint on line D guarantees that either 3 or 4 modules are taken from the second semester. The final constraint E ensures that 8 modules are taken in total. Some of these constraints might initially appear superfluous, but as we will soon see, 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 interact 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 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. This might not be unreasonable considering that this model would only be used by 4th year Computer Scientists. However, if and when we move to a more complex environment with more naive users, we expect that a good user interface will be essential.

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 [2] 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 $S \backslash d_n$, 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 $S \backslash \{d_n, d_i\}$, 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.

5 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

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

of study. These failed systems tend to be dominated by attempts 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[4] 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 [5]). However, we should expect that some day it will become a necessity, especially so as universities become more involved in distance learning. One current example is lifelong learning in the Open University. It is our intention to tackle a larger problem and develop a user interface. We should then be in a position to field the system and evaluate it. In addition, we might also consider higer levels of consistency. At present, only arc-consistency is established when making a decision. It might be interesting to investigate problems where higher levels of consistency are required, i.e. problems where illegal decisions are less obvious.

So far, we have not needed to use techniques from dynamic constraint satisfaction [3]. However, as the model becomes larger and richer we expect this will be a necessity. We are pleasantly surprised at the simplicity and effectiveness of Junker's explanation technique; this fits well with the above problem.

ACKNOWLEDGEMENTS

We would like to thank the Computing Science department of Glasgow University for supporting this final year student 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.

REFERENCES

[1] CHOCO. http://www.choco-constraints.net/ home of the choco constraint programming system.
[2] Ulrich Junker, 'Quickxplain: Conflict detection for arbitrary constraint propagation algorithms', in *IJCA'01 workshop on Modelling and Solving Problems with Constraints*, pp. 81–88, (2001).
[3] S. Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proceedings of AAAI-90*, pp. 25–32, (1990).
[4] Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.
[5] Janet van der Linden. An approach to dealing with non-standard constraint satisfaction problems. PhD Thesis, Oxford Brookes, 2000.

[4] ... a final year project carried out by the first author and supervised by the second.