Search design patterns

Henry Müller
Fraunhofer FIRST, Kekuléstr. 7, D-12489 Berlin, Germany
henry.mueller@first.fraunhofer.de

Abstract: In the context of constraint programming search algorithms are normally implemented as monolithic units, which often are complex and error prone, hard to understand and to extend. That holds even for object-oriented state-of-the-art constraint solvers, although the paradigm of object-oriented programming offers various techniques which encourage abstraction, flexibility and code reuse. We apply these techniques on CP search and propose a conception of platform independent, object-oriented search design patterns which state composable search modules. The composition of these modules forms complete search algorithms. Formulating search algorithm with modules has many advantages: They are flexible, reusable and easy to understand. As search modules encapsulate orthogonal abstractions, they categorize search and allow fast and easy search modelling.

1 Introduction

Faced with the huge amount of known CP search algorithms and the invention of new algorithms in rapid rate, it is rather astonishing that there are only little efforts of devising a common and consistent formalism or framework for these algorithms. Even though many algorithms have a lot of similarities and many common features are orthogonal, there is no general scheme describing how different aspects of search can be separated from each other and how these aspects can be composed to express variations of search.

Although many state-of-the-art constraint programming systems are hosted in an object-oriented environment, not much effort is made to exploit the possibilities of object-oriented programming which especially supports abstraction and flexibility, modularity and reuse of software. In [Mül05] we introduced a first impression of the "Generic object-oriented search environment" (Goose), which makes extensive use of object-oriented techniques to modularise CP search algorithms. In this paper present the possibilities of this environment and give some insight in its working:

Goose primarily proposes a catalogue of object-oriented search design patterns to enforce ease of use, modularity and flexibility of search algorithms. In Section 2 we motivate its use and present some central ideas of the conception. Using an example we show how to model search algorithms in Goose and demonstrate the consequences. Then we give an overview over already designed search patterns. Finally we discuss some related work and draw a conclusion (Section 3).

2 Goose conception

Motivation Most CP systems traditionally offer well known standard search algorithms as black box units, i.e. they can be used to solve arbitrary CSPs but the implementation is hidden. Sometimes they can be adjusted with flags to allow variation (e.g. to enable a heuristic). Some modern constraint systems additionally offer primitives which allow the implementation of own search procedures, e.g. choice points. With these primitives custom tailored algorithms can be implemented which solve special CSPs efficiently.

Traditionally CP search algorithms are implemented as monolithic units. But the implementation of any but the most basic monolithic search can easily get very complex, especially if several techniques are woven into one algorithm. Nested loops and conditional clauses can quickly cause high code complexity. Such algorithms are hard to understand, maintain and error-prone. It is also almost impossible to reuse parts of a complex algorithm in a slightly varying algorithm. Hence new algorithms are often copied and modified.

The perception of monolithic search contrasts with the compositional nature of constraints, which are natural modules: The semantic of a CSP is equivalent to the conjunction of the semantics of its constraints. Many search algorithms use the same features, and many of them are orthogonal. What one really wants is to combine such features into a runnable search algorithm, that is, model search out of components representing these features. The difficulty lies in finding these components: It's unclear how to distribute search algorithm behaviour among modules so that they are applicable to many algorithms and how to conceptualise their semantics as compositional as possible. The semantics of single search components should imply the semantic of the search algorithm composition. That is what we do within the Generic object-oriented search environment (Goose) [Mül05].

Search design patterns Goose is a conception of modularised CP-based search algorithms, which consists primarily of a catalogue of object-oriented *search design patterns* which define composable search modules and to the lesser extent of concepts which deal with the handling of these modules. Using well known object-oriented design patterns [GHJV94], which are solutions to often occurring problems of object-oriented design, we formulate search design patterns that define modules out of which search algorithms are composed. Especially the patterns *Factory Method*, *Strategy* and *Decorator* are sufficient to express many variations. A Search design pattern is basically defined as shown in [GHJV94]: It has a name, a problem description which sorts out when the pattern can be applied, it presents a possible solution to the problem and discusses consequences of this solution. A pattern defines its search modules structurally by giving appropriate class and interface definitions. Often a search module (class) describes an algorithm a in abstract steps, and the semantics of these steps (methods) – and thus a – are defined by specifying their pre/post conditions. Search design patterns are designed with emphasis on flexibility and code reuse, are platform independent and can be applied to arbitrary object-oriented constraint programming systems. Search modules represent more or less orthogonal features of search algorithms, i.e. different search strategies, variable and value selection heuristics, alternative implementations with different space and time tradeoffs. A search algorithm is formulated by composition of several modules in a plug and play way.

Composition of search modules into complete algorithms has advantages over the traditional way of implementing search: Modules encapsulate concepts into reasonable, understandable units. This significantly accelerates development of new algorithms and simplifies correctness and termination proofs. Provided with adequate design, modules are generic and reusable in different contexts. They categorise search and enable modelling of search algorithms. A search model is essentially a composition description of compatible search modules, and thus done fast and easily. Aggregated algorithms are less error prone and comparable. Models are aggregated to form complete, runnable programs. Goose defines an XML language *GooseML* to express search (and problem) models and an aggregation algorithm capable of transforming a model into a runnable, platform-specific program (Figure 1). Problem and search model are interconnected with shared variables. For every search component used in the model the host system must provide an implementation which fullfills the specification given by the underlying search design patterns. A special feature of GooseML are *variations*, which makes formulation of algorithmic variation easy, i.e. stating a base algorithm and several slight variations.

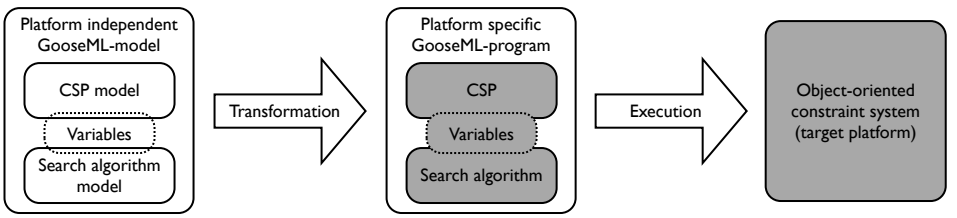

Figure 1: Transformation of a GooseML-Model into a platform-specific, runnable form

Goose defines four *abstraction layers*, and each search module can be associated with one layer. I is the most abstract, IV the most special abstraction layer. Layer I defines an *abstract decision maker* which is the frame for every search algorithm. Layer II specialises this algorithm to a *search strategy* (e.g. local search or systematic search). Layer III contains *variation modules*, which alter or enhance the behaviour of other modules. Layer IV holds the most special *search space* modules. An *aggregated search algorithm* always consists of at least 3 parts: The abstract decision algorithm, a search strategy and a search space. Variation components can express variations of this algorithmic skeleton.

Goose is inspired by Prosser's categorisation of labelling search algorithms [Pro93], which allows to express variations of CP search algorithms in a uniform way. It distinguishes algorithms by the way in which they move vertically along the search tree, e.g. moving back chronologically in an inconsistent state or jumping back over not involved variables. With Goose we generalise and extend this approach to allow deeper categorisation by further separation of orthogonal features. It allows not just labelling but the formulation of arbitrary algorithms. Identified features are worked up with object-oriented methods and encapsulated into modules to enforce flexibility, extendibility and code reuse. The number of possible algorithms arises by building all possible variations over these modules. Goose abstraction layers I and II embody a general form of Prosser's scheme: The generic decision algorithm successively *makes decisions* as long as no solution is found and *undoes decisions* when an inconsistent state arises. The concrete form of these actions is

realised in the search strategy, e.g. a systematic search may try to extend partial solutions and backtrack. On top of this we generify search strategies by factoring out data structures into a special search space. Strategies can thus be reused for different spaces. Furthermore we vary behaviour of search strategies and search spaces with variation modules.

We will demonstrate the consequences of modelling search algorithms with an example of depth first search (DFS) with backtracking over variables. Figure 2 shows the class diagram specifiying available modules and the search model as an object diagram (attributes and methods hidden). This ought to give an impression of how to form a complete, runnable search algorithm out of modules. The classes are search modules defined by different search design patterns, and the algorithm is assembled by aggregating instances of these classes. The outermost object chronologicBacktracking is an interface between the host system and the algorithm. Thus, the aggregate forms an easy to use black box and can be handled like any other native search procedure. A search request sent to chronologicBacktracking is *delegated* to varLabeller, which realises a generic backtracking search over the compatible search space variables, which is basically a variable sequence. Compatibility of modules is structurally defined by interfaces, i.e. chronologicBacktracking works on a search space of type *Backtrackable* ◯. The chronologicMover helps varLabeller, which performs the abstract *high level search logic*, to move chronologically over the search space variables, which contains *data structures and low level helper functions* regarding these structures.

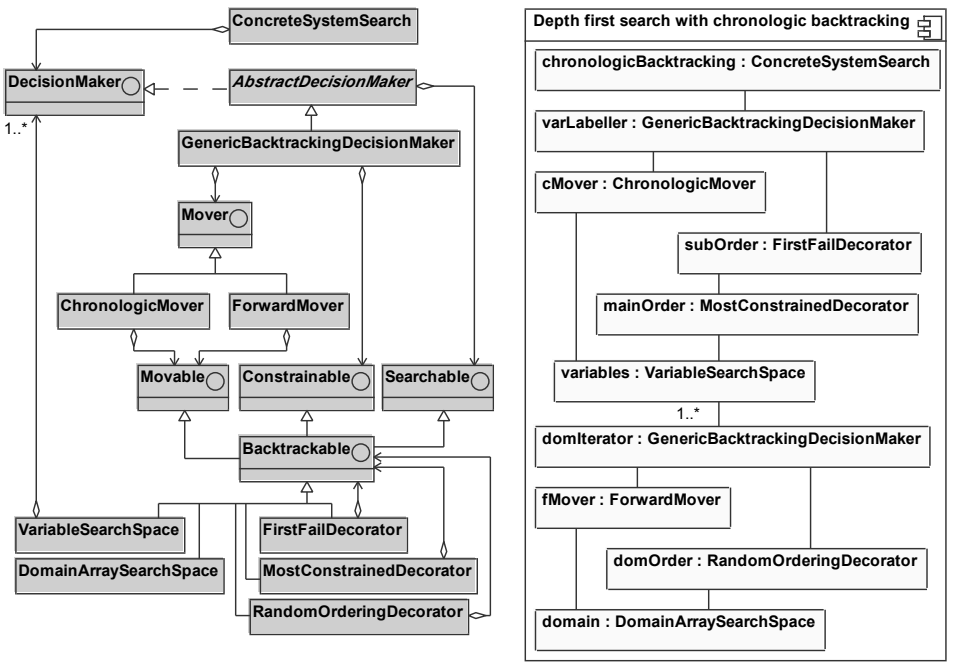

Figure 2: Class diagram of search modules and object graph of an aggregated search algorithm

DFS constructs a decision tree while systematically traversing the search space. The class

GenericBacktrackingDecisionMaker is *reused* in two contexts: The model uses varLabeller to construct the *vertical dimension* of the decision tree, thereby moving forth and back chronologically. And the variable sequence object variables contains one domIterator object for each variable's domain, each doing the job of finding a consistent value for its variable. A domIterator constructs the *horizontal dimension* of the tree by moving forth from one value to the next.

The two mover objects are examples for necessary variation modules. The decorator classes are optional variation components, their instances can be added to the model by wrapping them around a compatible object. In the figure two variable ordering heuristics and one value ordering heuristic are applied in that way. Decorators can be recursively stacked to add their behaviour to the algorithm. Finally, it should be mentioned, that an aggregated search algorithms can be manipulated both statically in advance and dynamically during runtime. The heuristics are a good example for objects that can be inserted into the algorithm and can be removed again during runtime

Figure 3 shows available Goose design patterns and the level of abstraction each pattern is most concerned with. Each pattern defines one or more classes which are used to aggregate an algorithm, e.g. the algorithm in Figure 2 uses classes from four different patterns: *Abstract decider* defines the Goose layers in the first place and defines the common base algorithm. The base algorithm can drive arbitrary search algorithms, but up to now with *Generic Tree search* we focus on systematic search. *Search dimension* is a special search space design that allows generic handling of both dimensions of a decision tree, i.e. stating heuristics according to the *Sorting* pattern, which can equally used to sort variables and values of a single variable. Correctness and termination are considered by means of a general search space traversal graph for arbitrary algorithms and by means of a decision tree for systematic tree search.

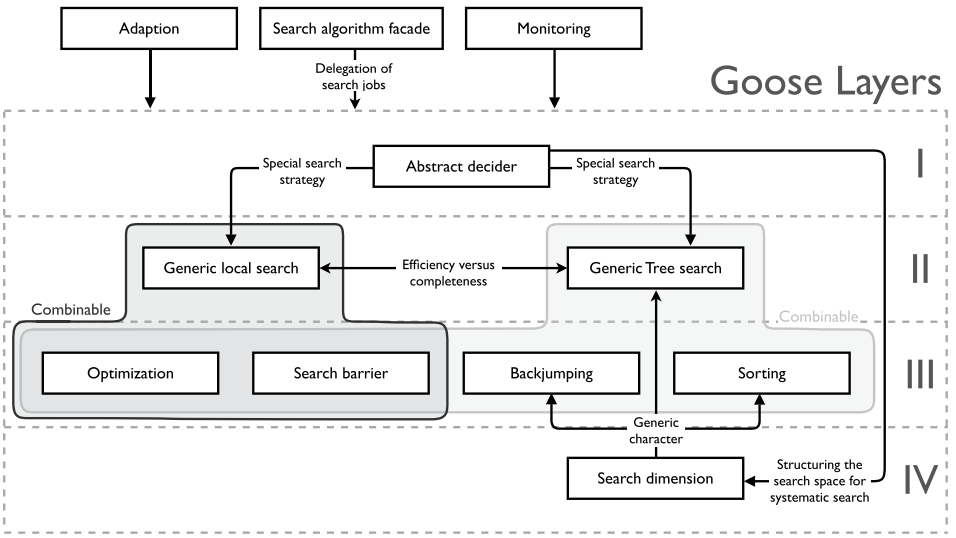

Figure 3: The Goose design patterns

3 Related Work and conclusion

Most constraint solvers offer limited or no search modelling capabilities (Koalog, CHIP, ILOG Solver (OPL is an addon), `firstcs` [HMSW03], SICStus prolog, ECL^iPS^e, JSolver). Normally, some standard search procedures are offered as monolithic black box algorithms with flags for configuration, and sometimes heuristics can be used. Object-oriented constraint systems can implement Goose search design patterns, thus gaining the presented search modelling features.

There are basic differences between Goose and other modelling approaches. Firstly Goose is a set of search design pattern, which can be implemented in different constraint solvers. The pattern catalogue is an open framework offering several abstractions, arbitrary concepts can be added or changed. Other modelling approaches like OPL [Hen99] – probably the most expressive tree search modelling language – are proprietary and shut systems. OPL achieves a concise and elegant formulation of search algorithms, but it is quite complex and neglects accessibility to other systems. Goose is open and makes use of standards like UML and XML for easy access of modern IT systems. By the use of standard OO techniques the Goose concepts can be understood by many people, and modelling of search is a simple plug and play process.

Goose is a new approach which improves implementation and modelling of CP search in many ways. Its concepts will be published in detail in the author's doctoral thesis [Mül08].

References

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hen99] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[HMSW03] Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs — A Pure Java Constraint Programming Engine. Juli 2003. submitted to the 2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'03).

[Mül05] Henry Müller. GOOSE – A generic object-oriented search environment (extended abstract), 2005. Submitted to Eleventh International Conference on Principles and Practice of Constraint Programming, CP 2005. The submission is available on the CP website: http://lia.deis.unibo.it/ zk/DP2005/DP2005.htm.

[Mül08] Henry Müller. *Konstruktion und Adaption generischer, modularisierter Suchalgorithmen im Kontext objektorientierter Constraint-Programmierung*. PhD thesis, Technische Universität Berlin, (forthcoming) 2008.

[Pro93] Patrick Prosser. Hybrid Algorithms For The Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268, 1993.