GOOSE – A generic object-oriented search environment (extended abstract)*

Student: Henry Müller, Supervisor: Prof. Dr. Ing. Stefan Jähnichen

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

Abstract. The constraint programming community keeps on creating numerous search algorithms. It is an as desirable as difficult task to implement a variety of search algorithms in a single unifying framework. This design proposal states an object-oriented environment which supports development of generic search algorithms. GOOSE is abstract enough to house dissimilar search approaches and separates abstract generic logic from domain details. The presentation addresses multidimensional search structures, behaviour distribution and control flow. Implementing search algorithms according to GOOSE will make them easier to understand and compare, the code will be flexible and reusable.

1 Introduction

A lot of different search algorithms exist in the world of constraint programming. There are some more or less different fundamental search approaches (local search, global search, genetic algorithms, etc.), and each one offers own methods and heuristics. Attempts have been made to categorise search algorithms to make them comparable and understandable [7, 8]. It is desirable to express the different approaches in a unifying frame, what turns out to be quite difficult on a theoretic level. But it is also difficile to manage algorithmic variants on implementation level.

What is the appeal of a unifying framework from the implementation perspective? At the worst each special algorithm is realised as a monolithic unit. Their code is complicated, vast and difficult to grasp. Monoliths often suffer from duplicated code and complex conditional constructs impeding maintenance. Addressing these difficulties, this paper proposes the object-oriented GOOSE framework to support implementation of generic constraint-directed search algorithms:

- At first we consider the basics and origins of this proposal (section 2).
- GOOSE is presented in terms of UML and design patterns (section 3). The goal is to improve reuse and flexibility. GOOSE knows 3 component types:
  (a) Search facade simplifies assembly of GOOSE components and usage of compound search algorithms (section 3.1).

* This work is funded by the EU (EFRE) and the state Berlin, grant no. 10023515.

(b) A general search algorithm is presented and is exemplary extended to a concrete generic backtracking algorithm (section 3.2).
(c) Generic logic operates on compatible search spaces (section 3.3). GOOSE provides a search dimension concept to simplify cooperation between logic and space and to streamline control flow.

- Finally we discuss related work and draw some conclusions (section 4).

As a proof of concept GOOSE was implemented within our object-oriented `firstcs` [5] solver. Up to now the concept handles variants of backtracking search and deals with topics like constraint-based scheduling, static and dynamic variable ordering, justifications and backjumping, optimisation, randomisation and restarting. GOOSE is part of the author's effort to create an intelligent CSP solving system. A high-level *intelligent reasoning instance* shall provide progress judgement, gathering of experiences and dynamic adaptive configuration. It's of particular interest to react rationally on unknown problem types.

2 Basics and origins

The main influence on GOOSE is Prosser's categorisation [8] of backtracking search algorithms (section 3.2). Prosser defines a general search procedure, which uses two functions `label()` and `unlabel()` representing the algorithm's forward and backward move in the search tree. Combination of different algorithms' `label()` and `unlabel()` functions creates new hybrid algorithms.

This design proposal explicitly goes for an efficient object-oriented design, which enforces code reuse and flexibility by the correct use of class inheritance and object composition [3]. It considers Prosser's view [8] in a generic way and allows for a plug & play like exchange of components. The study of design patterns [3], which offer solutions to standard problems of object-oriented software design, was of great help. Used patterns are mentioned for further research, and their effects are discussed for each application (section 3).

GOOSE's design provides a base for search variants. Algorithmic variations are handled with object composition in the first place, as it is generally to be favoured over class inheritance [3]. There are many wrong ways to express algorithmic variations in OOP. The author recommends applying the flexible Strategy or Decorator pattern instead. [3] describes a decorator as a skin over an object that changes its behaviour, a strategy is said to change an object's guts. The flexible strategy and decorator objects are exchangeable at runtime.

3 GOOSE design overview

Generic programming makes algorithms reusable for different data types. In order to create a generic search algorithm, abstract common logic has to be extracted from the entirety of concrete search algorithms. Only common high-level behaviour is to be divided from low-level behaviour, which is relevant to the specific domain and thus stays with the data. Figure 1 shows the design:

Abstract and generic *search logic* classes called *labellers* operate on *search space* classes to do a labelling process. An optional facade encapsulates labellers and spaces, it shields the programmer from subsystem details offering easy access.

Search algorithm partitioning is refined into a two-step abstraction: A general search procedure is offered by the `AbstractLabeller` class. A specialisation like `GenericLabeller`, which defines backtracking search, completes the high-level generic search logic. A generic labeller knows which primitive data manipulation methods are available via polymorphism: The search space implementation has to fit prepared logic compatible interfaces like `Backtrackable`.

3.1 Search facade (optional)

`GenericSearchFactory` simplifies composition of single elements into a meaningful aggregation. It is conducted from the Abstract Factory pattern and relieves the developer from combining search components himself. `AbstractSearch` is an implementation base for derived search algorithms and offers mainly methods to initiate a search process. The only deriving class is `GenericSearch`, which is an envelope for other components: Search queries and maintenance requests are delegated to its labeller, which calls on its part search space methods.

3.2 Search logic

`AbstractLabeller` represents abstract general search intelligence. It realises with `processLabelling()` an iterative process of making (`label()`) and retracting (`unlabel()`) decisions. `AbstractLabeller` follows the Template Method pattern. It defines an algorithmic framework in abstract steps and leaves `label()` and `unlabel()` open for implementation by derived labellers. Thus the invariant part of search logic is abstractly factored out as a general algorithm, giving subclassing labellers the chance to differ in their decision process.

A generic subclass of `AbstractLabeller` defines `label()` and `unlabel()` corresponding to a specially-tailored search space interface. Collaboration between generic logic and concrete search space is ensured by polymorphism: A special labeller expects a space of a given type, as its decision methods have to know available search space methods. Specialisation of decision making reflects basic search categories like local search, global search or evolutionary search.

3.3 Search spaces

We want a concrete search logic class to work with every compatible search space. E.g. `GenericLabeller` processes spaces of type `Backtrackable`, which contains three other interfaces: `Searchable` defines methods necessary for the abstract search process. `Storable` defines methods concerning state maintenance. Finally methods in `Movable` enable movement over a search space. Different search logic components accessing a search space will mostly need only appointed aspects: The `AbstractMover` component for example needs only movement behaviour and thus requires the `Movable` part. `AbstractLabeller` needs the

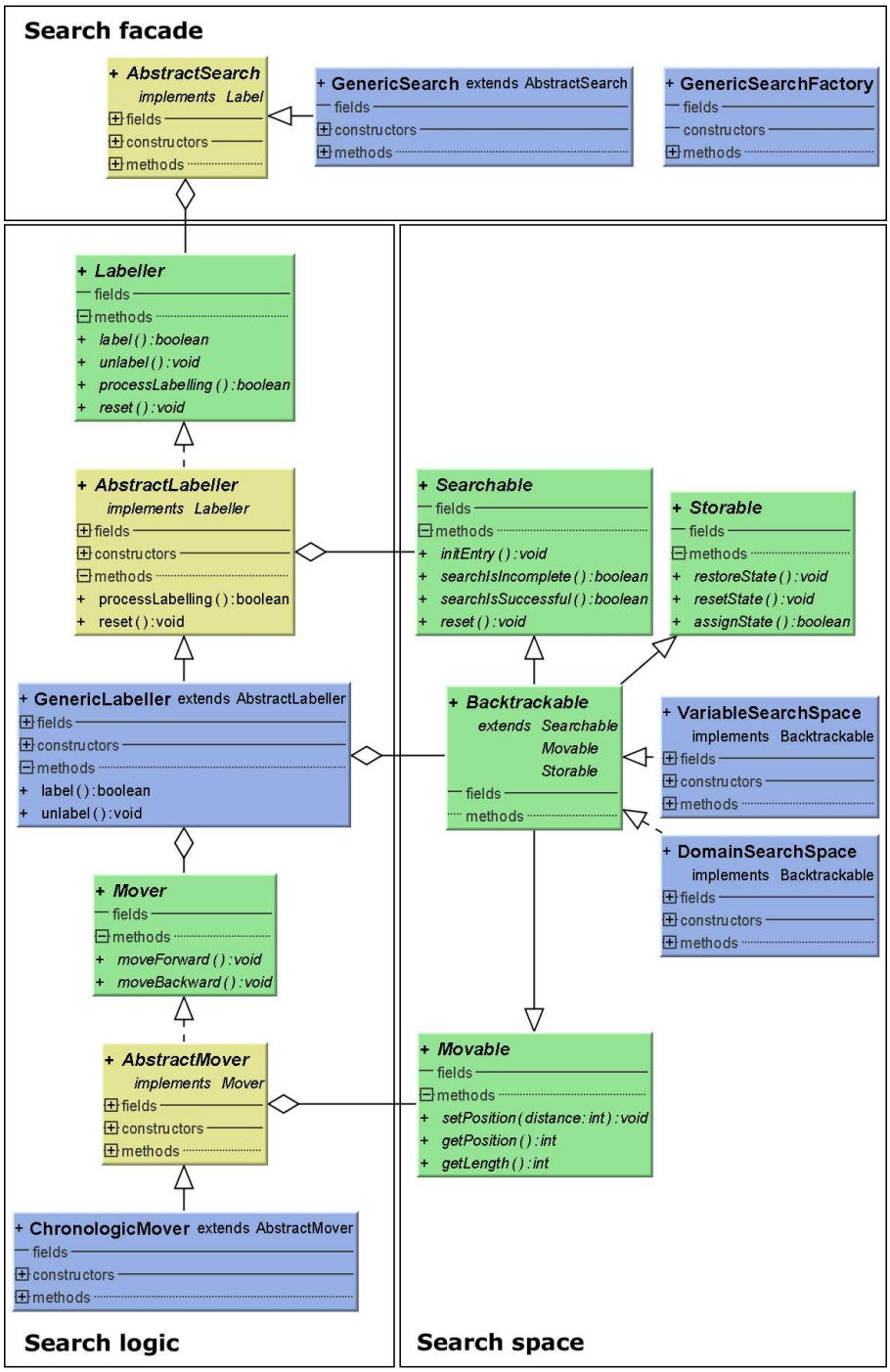

Fig. 1. Core interfaces and classes of GOOSE

logic handling portion, and that is *Searchable*. The point is to decouple logic components from behaviour elements they don't need.

Search dimensions A flexible concept of backtrackable spaces evolved while working with them. A backtrackable space is seen by `GenericLabeller`'s logic as an abstract search dimension, which is an ordered scope of variation where the variations allow movement over this scope. The dimension has a pointer, which indicates the current position. The dimension pointer is moved explicitly. If logic initiates a state change on the space, it will happen at the indicated position.

Even basic search algorithms will use telescoped iterative control structures. For example a set of variables constitutes a search dimension, which in turn includes an entangled domain dimension. The outer loop iterates over variables trying to find a problem solution, the inner loop iterates over values trying to find consistent variable assignments. Thinking of a search tree, the variable dimension corresponds to vertical movement and the domain dimension to horizontal movement in the tree. GOOSE is chiefly about code reuse, so it suggests itself to represent an inner loop with an appropriate labeller and search space.

Control flow and distributed behaviour There are two ways to organise control flow of entangled dimensions: The first approach models a parallel control flow, where the outer labeller knows of the inner one and controls it. The second approach hides the inner labeller in the outer search space building a linear control chain. GOOSE uses linear control flow, because it makes the outer logic more general. How is behaviour correctly distributed among search dimensions? A good sign that behaviour is in the right dimension is the absence of long reference chains, which point out misplaced code. Generally the developer may either choose to manage an entangled situation with a single labeller-space-pair or to resolve inner loops by boxing logic and space components.

4 Related work and Conclusion

Gent and Underwood formulate in [4] scheme code for a general search algorithm, which can be modified to realise special algorithms by implementing certain service functions. The algorithm is formulated with instruments of constructive type theory and gives GOOSE a theoretical base regarding correctness.

In [7] Jussien and Lhomme propose the generic PLM algorithm for systematic and non-systematic search, it uses the components Propagation, Learning and Moving. It remains theoretical when arguing with sets and predicates. GOOSE in contrast accents implementation but includes the three components implicitly.

In [2] Givry and Jeannin present the ToOLS library, which supports CP programmers in building complex search algorithms. [2] makes no statements about generic search or concrete design measures. In contrast to GOOSE ToOLS commits itself to partial search and local/global search hybrids. The library is search tree centred while GOOSE stays general. ToOLS is part of the constraint solver Eclair, GOOSE can be implemented in any object-oriented system.

Van Hentenryck presents in [9] search capabilities of OPL, a modelling language for combinatorial optimisation combining algebraic and set notations, a constraint language and instruments to specify search procedures. GOOSE is an object-oriented design proposal, OPL abstracts and hides object-oriented details.

Chatzikokolakis et al. combine in [1] methods of global constructive search and local repair search and present the generic Construction-Repair algorithm. `AbstractLabeller` resembles the CR algorithm, what implies that GOOSE can handle both repair algorithms and construction-repair hybrids.

Conclusion GOOSE is an object-oriented design proposal for generic constraint-directed search. The few works explicitly stating a generic search algorithm for CSP either stay abstract [6, 7] or formulate rather a general algorithm, which resembles `AbstractLabeller`, instead of a generic one [1, 6, 7]. GOOSE focuses problems on implementation level and considers both the general and the generic element. Future work will cover the introduction of non-systematic generic search algorithms and dynamic adaptive search configuration, e.g. switching dynamically from chronological tree movement to backjumping or from global to local search etc. The author hopes that GOOSE may serve interested developers as a large design pattern to create sophisticated search algorithms easily.

References

1. Konstantinos Chatzikokolakis, George Boukeas, and Panagiotis Stamatopoulos. Construction and repair: A hybrid approach to search in csps. In George A. Vouros and Themis Panayiotopoulos, editors, *SETN*, volume 3025 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 2004.
2. S. de Givry and L. Jeannin. Tools : A library for partial and hybrid search methods, 2003. Paratre dans CP-AI-OR 2003.
3. E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
4. Ian P. Gent and Judith L. Underwood. The logic of search algorithms: Theory and applications. In *Principles and Practice of Constraint Programming*, pages 77–91, 1997.
5. 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) at the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003.
6. M.S. Fox J.C. Beck. A generic framework for constraint-directed search and scheduling. *AI Magazine*, 19(4):101–130, 1998.
7. N. Jussien and O. Lhomme. Unifying search algorithms for csp, 2002. Research Report 02-3-INFO, Ecole des Mines de Nantes, Nantes, France.
8. Patrick Prosser. Hybrid Algorithms For The Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268, 1993.
9. Pascal van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, 2000.