A Compositional Framework for Search

Chiu Wo Choi[1], Martin Henz[1], Ka Boon Ng[2]

[1] School of Computing, National University Of Singapore, Singapore
{choichiu,henz}@comp.nus.edu.sg
[2] Honeywell Singapore Laboratory
kevin.ng@honeywell.com

Abstract. Recent developments in constraint programming systems show an increasing emphasis on providing abstractions for configuring search. Existing frameworks for tree search offer customized exploration, branching and state restoration policies. We argue, however, that most of these frameworks do not provide adequate abstractions for more complex search scenarios, where different search techniques are used in different phases of the search, where parts of search trees are systematically discarded, or where tree search is embedded in a context of local optimization. In this paper, we propose to describe complex search scenarios using a compositional framework, realized in the Figaro library, with an abstraction called engine. We demonstrate its expressivity by re-formulating two complex search scenarios from the literature.

1 Introduction

Finite domain constraint programming (CP(FD)) systems provide software architectures for the integration of algorithms for propagation and tree search to solve combinatorial search problems. Recent research in constraint programming (CP) systems is paying increasing attention to the software design in order to match application-specific requirements. One important aspect is to provide abstractions for configuring search. CP systems like ILOG [Per99] and OPL [VPP00] describe tree search in terms of *search goals*, which are inherited from logic programming. The language Oz allows programming of tree search algorithms using the built-in data structure called *spaces* [Sch97,Sch00]. SALSA [LC98] is a language designed for specifying search algorithms using the concept of *choice points*. [1] The Figaro library provides a modular architectures for customizing the state restoration policies, in addition to the branching and exploration of tree search [HMN99,CHN00,Ng01,CHN01].

Among the existing frameworks for tree search, we observe that abstractions are limited to customized exploration, branching and state restoration policies. We argue, however, that as the complexity of application-specific search algorithms increases, it becomes more difficult to describe the corresponding search scenarios. Typical complex search scenarios include dividing the search into multiple phases where each phase employs a different search technique; systematically discarding parts of search trees; and embedding tree search in a context of local

[1] SALSA also supports local search, but this paper only focuses on tree search

optimization. The existing frameworks focus on low-level abstraction details and thus make it tedious for programmers to apply the right search algorithms and techniques. Furthermore, programming low-level details adds additional complexity to the code and is usually an error-prone process.

Modern engineering practice allows engineers to build complex systems by composing small and well-defined units. Such a compositional framework leads to a better understanding of the desired functionality and structure of the overall system. Moreover, it encourages reuse, leading to both a faster development cycle and a more robust system.

Search can be thought of as such a complex system, which can be decomposed into smaller functional units called *engines*. We propose a compositional framework for describing complex search scenarios using engines. Each engine performs a single function (e.g. , search, print), and engines can be composed together to form more complex search scenarios. We are designing and implementing the framework using Figaro [HMN99,CHN00,Ng01,CHN01], a C++ CP library, further pursuing its objective of high reusability and rapid prototyping for search components. Notice that we design our framework within the object-oriented paradigm rather than the logic paradigm.

We present the compositional framework in Section 2. Sections 3 and 4 discuss the details of engines. Sections 5 and 6 discuss the operators that we apply to engines. The first case study (Section 7) discusses engines for embedding tree search in the context of local optimization. The second case study (Section 8) on the three-phase approach of round-robin tournament scheduling demonstrates the practicality of our framework. Section 9 compares the approach with existing frameworks.

2 Framework

The framework identifies different engines for describing search, and how they interact with each other. Each engine should be a coherent functional unit and loosely coupled with one another. Specific design requirements of the framework are as follows:

1. a standard interface for combining engines to form more complex engines,
2. a mechanism to explicitly control the execution of engines for coherent integration,
3. support for engines to maintain stateful information as the consequence of explicit control,
4. orthogonal concepts to provide high reusability, and
5. support for extension to search scenarios other than simple tree search.

The BNF grammar in Figure 1 gives an overview of the design. Rule 1 is a composition rule that specifies that two engines are combined to from a new engine using the $\rightarrow$ operator. Engines communicate with each other using constraint stores. This fulfills our design requirement (1). To achieve requirement (2) and (3), we implement an engine object using a demand-driven approach for

Fig. 1 The Syntax of Engine

$\langle engine \rangle ::=$	$\langle engine \rangle \rightarrow \langle engine \rangle$	(Rule 1)
$\mid$	ModelEngine($\langle model \rangle$)	(Rule 2)
$\mid$	PrintEngine($\langle model \rangle$)	(Rule 3)
$\mid$	TreeSearchEngine(...)	(Rule 4)
$\mid$	First($\langle engine \rangle$)	(Rule 5)
$\mid$	Last($\langle engine \rangle$)	(Rule 6)

producing output. The demand-driven approach will be discussed in the next section. Rule 2, 3, and 4 specify how to build primitive engines *ModelEngine*, *PrintEngine*, and *TreeSearchEngine*. A *TreeSearchEngine* (details in Section 4) is composed of orthogonal search components, which design fulfills requirement (4). Rule 5 and 6 specify solution filtering operations over engine: *First* and *Last*. The ability to define with new engines using our object-oriented framework achieves requirement (5). We discuss the details of engines in the subsequent sections.

The simple example below shows how to use our framework for solving the N-Queen puzzle:

ModelEngine(N-Queen) $\rightarrow$ TreeSearchEngine(...) $\rightarrow$ PrintEngine(N-Queen)

The framework divides the problem solving process into three phases. In the first phase (line 1), *ModelEngine* generates a constraint store that contains the N-Queen problem model. In the second phase (line 2), *TreeSearchEngine* takes in the constraint store and looks for all the possible solutions using tree search. In the third phase (line 3), *PrintEngine* transforms the solutions obtained from the tree search engine to the desired output format.

3 Engines

In CP(FD), a constraint store (or simply store) represents a computational state, hosting finite-domain variables and constraints for performing constraint propagation to eliminate inconsistent values. Refer to [Ng01] for a detailed discussion of stores.

Engine is the basic unit of abstraction in our framework. The basic operation of an engine is to manipulate or transform stores. We can view engine as a machinery that takes in a store for processing; and produces a stream of stores as output. Output of stores is demand-driven, meaning that the next store to output is only computed upon request from a subsequent engine.

Fig. 2 The Design of an Engine

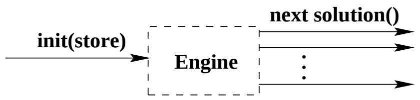
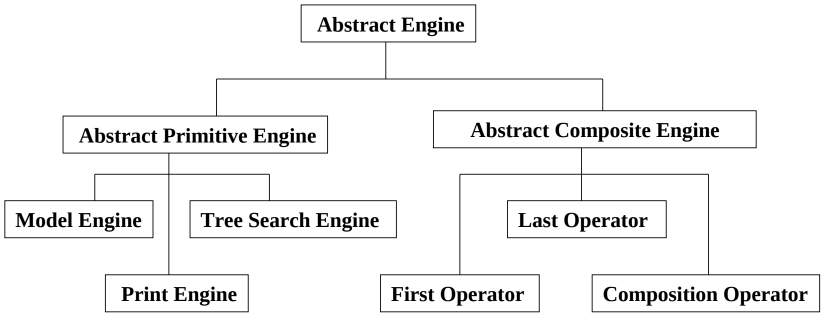


Fig. 3 The Class Hierarchy of Engines



The two main functions of engines are init and next solution. Figure 2 shows the design of an engine. The init function accepts a store as argument and prepares the engine for generating the output stream of stores. The next solution function requests the next output store in the stream, realizing the demand-driven approach. In the engine context, solutions of the engine are represented by stores. For the discussions of engines, we use the terms solution and store interchangeably.

The class hierarchy of engines is shown in Figure 3. Engines are instances of an abstract class (Program 1). There are two types of engines: primitive engines and composite engines, also implemented as an abstract class. Primitive engines are built from basic orthogonal components. Primitive engines include *ModelEngine*, *PrintEngine*, and *TreeSearchEngine*. Composite engines are build from one or more other engines. We can view composite engines as wrappers around engines to form new engines. They include filtering operators (*First* and *Last*), and the composition operator $\rightarrow$. Notice that the operators are themselves engines, so that the resulting type after applying the operator is still engine, and can be further manipulated (refer to the BNF grammar in Figure 1).

ModelEngine is a trivial engine, whose function is to return a store containing the problem description. *ModelEngine* is the component that contains the description of the problem to be solved. It takes model as an argument and performs no operation during init. When there is a request for next solution, it returns a store imposed with the problem model.

PrintEngine is a trivial engine, whose function is to transform solutions to the desired output format. *PrintEngine* accepts a model as an argument. During init, it makes a function call to model to print the solution.

Program 1 C++ Abstract Class Declaration of Engine

```
class Engine {
public:
   virtual void init(store*) = 0;
   virtual store* next_solution() = 0;
};
```

Program 2 Actual C++ Implementation of N-Queen

```
Compose(Compose(new ModelEngine(N_Queen),
                new TreeSearchEngine(N_Queen->first_fail(),
                                     new Copying_Node(),
                                     new Depth_First())),
        new PrintEngine(N_Queen));
```

4 Tree Search Engines

Tree search is a search algorithm for finding all solutions of a problem. In CP(FD), the solutions reside on the leaves of the search tree, which together form the output stream of stores in the engine context. The primitive engine *TreeSearchEngine* for performs tree search.

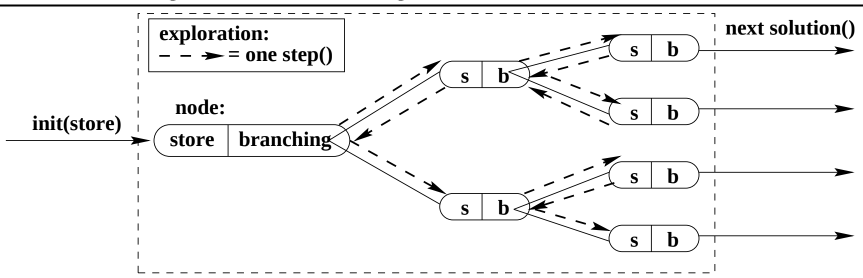
TreeSearchEngine is composed of three orthogonal components: branching, node, and exploration as discussed in [CHN01]. Branching describes the shape of the search tree. Common branching algorithms include naive enumeration and first-fail. A (search tree) node encapsulates the store and branching, and defines the state restoration policies such as copying and trailing. Exploration controls the order of traversal of the search tree.

Program 2 shows the actual C++ implementation of tree search for solving the N-Queen puzzle. This corresponded to the high level syntax we gave in Section 2. `Compose` is the prefix function for implementing the $\rightarrow$ operator due to the limitation of C++ operator overloading. For this example, *Tree Search Engine* takes in first-fail branching, copying state restoration and depth first exploration as components.

In our framework, exploration needs to adopt a demand-driven approach of traversing the search tree to match the design of the engine. Thus, an implementation requires the exploration to explicitly maintain a state during the course of search. The function one step requests the exploration to perform a single step of traversal. Refer to [CHN00] for implementation details.

Figure 4 shows the design of a *TreeSearchEngine*. It takes a branching, a node, and an exploration as arguments. During init, it uses branching to create a root node to initialize exploration. When there is a request for next solution,

Fig. 4 The Design of TreeSearchEngine



it repeatedly calls one step on the exploration until it finds the next leaf node. Then, it returns the store kept within the leaf node as a solution.

The concept of limit in tree search [Per99,VPP00] is a useful facility to control the amount of tree search we want to perform. In our architecture, it is easy to extend the *TreeSearchEngine* to take in another component called limit. Within one step, we then check the current exploring node against the limit to verify if the search must be terminated.

5 Filtering Operator

The default behavior of an engine is to output all solutions. This behavior is not desirable, when only one particular solution is needed. The unary filtering operators act as a wrapper to control the out-flow of solutions, and thus to systematically discard parts of the overall search tree. Here, we discuss two common filtering operators: *First* and *Last*. It is straightforward to apply the idea to other filtering operators.

The unary *First* operator (left side of Figure 5) is applied to an engine and yields a new engine, which keeps a flag to determine if it has already given out a solution. When the new engine receives a request for next solution, and if the flag is true, it indicates that there is no further solution (in our implementation by returning the null pointer). Otherwise, it will return the first solution from the engine. The example:

First(Tree Search Engine(naive,Copying Node,Depth-First))

shows a typical scenario where we are only interested in the first solution of depth-first search.

The unary *Last* operator (right side of Figure 5) yields a new engine, which has a buffer to keep track of the last solution coming out from the argument engine. When the new engine receives a request for next solution, it exhausts all solutions of the argument engine while keeping track of the latest solution in its buffer. When the argument engine has exhausted its search space, the new engine returns the buffered solution. The example:

Last(Tree Search Engine(naive,Copying Node,Branch-and-Bound))

shows a typical scenario of solving an optimization problem where we are only

Fig. 5 The Design of Filtering Operator

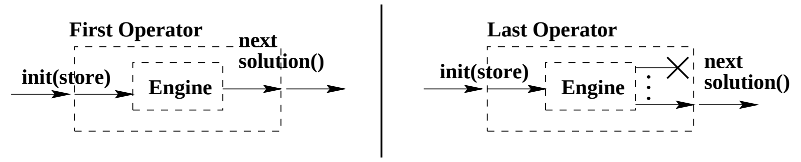
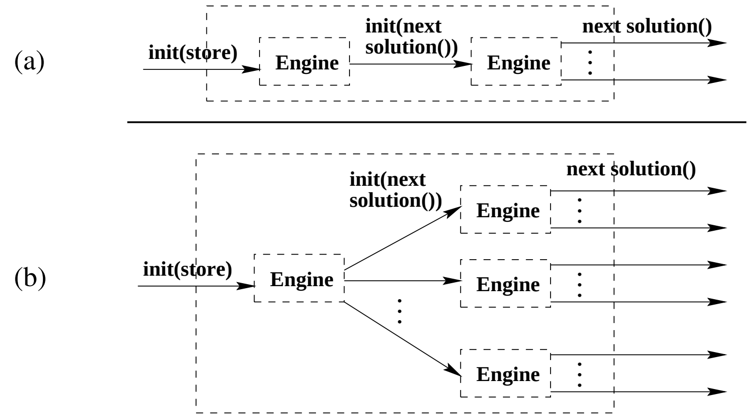


Fig. 6 The Effect of Composition Operator



interested in the last solution (i. e. , the optimal solution) of branch-and-bound search.

6 Composition Operator

The previous two sections show that a unary engine operators can yield interesting functionality. However, in order to build more complex engines, we must provide a way for combining engines together, which is the main feature of our compositional framework. The $\rightarrow$ operator, also called *Composition* operator, provides a standard interface to plug engines together. We can view the $\rightarrow$ operator as an engine to direct the flow of solution streams between two engines. As a consequence, the $\rightarrow$ operator is associative, meaning that $(A \rightarrow B) \rightarrow C \equiv A \rightarrow (B \rightarrow C)$.

The *Composition* operator takes two engines as argument to form a composite engine. During init, the composite engine initializes the first engine. When there is a request for next solution, if the second engine is not initialized, it will take in the next solution from the first engine and initialize the second engine. Then, it will return the next solution from the second engine. If the second engine has no more solutions, it will request the next solution from the first engine, re-initialize the second engine, and the cycle will repeat until the first engine has no more solution.

Figure 6 shows the effect of the *Composition* operator. Part (a) shows the effect of composing two engines, when the first engine can only generate a single solution. The result is a sequential processing of stores from the first engine to the second engine. When the first engine can generate more than one solution as shown in Part (b), the *Composition* operator enumerates all possible solutions from the first engine and feeds them into the second engine, each time re-initializing it with a different store.

The following example demonstrates how to divide a single tree search into two phases, with each phase employing a different tree search technique:

```
1   ModelEngine(model) →
2   First(TreeSearchEngine(Naive,CopyingNode,DFS) →
3         TreeSearchEngine(FirstFail,TrailingNode,LDS)) →
4   PrintEngine(model)
```

The search tree is divided into two parts, upper and lower. The upper part (line 2) performs depth-first search (DFS) using naive enumeration with copying, while the lower part (line 3) performs limited-discrepancy search (LDS) using first-fail with trailing.

7 Embedding Tree Search for Local Optimization

Tree search is useful beyond the usual role as a complete search algorithm. In [NP98], an approximation algorithm is presented based on tree search using ILOG SCHEDULER, whose results exhibit a performance that is competitive with other local search algorithms for solving the job shop scheduling problem (JSSP). The key idea of the approximation algorithm is embedding tree search inside a loop to perform local optimization. During each iteration, the algorithm randomly keeps parts of the best solution before restarting the tree search. This section shows that it makes sense to introduce new engines and components to describe this kind of algorithm in our framework.

We introduce two new engines: the *RelaxEngine* and *Iteration* operator, and give their BNF grammar in Figure 7. The *RelaxEngine* (Rule 7) takes a model as argument and uses it to compute the parts of the best solution to keep for the next iteration of tree search. The *Iteration* operator (Rule 8) takes a controller and an engine as arguments for performing the iteration process. The controller represents a termination condition, while the engine defines the action to be performed in each iteration. We propose the generic *Iteration* operator as a means to compose engines into a loop.

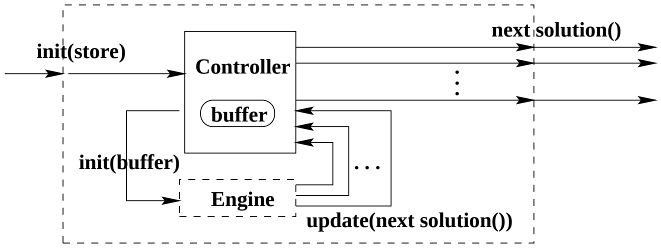
The design of the *Iteration* engine is depicted in Figure 8. The init operation initializes the controller. The controller holds a store in its buffer. When the iteration engine receives a request for next solution, and if its engine is not initialized, the engine is initialized with the store in the buffer. The iteration engine then retrieves the next solution from its engine, updates the controller and returns the solution. If the engine has no more solution, it checks the controller's stopping condition. If the condition is not satisfied, the iteration engine repeats the process by re-initializing the engine with the buffered store.

The *RelaxEngine* is a primitive engine used to keep the parts of the best solution. The design of the *RelaxEngine* is simple. Upon init, it keeps the best

Fig. 7 The Syntax of Iterative Relaxation

$\langle engine \rangle$ ::=	RelaxEngine($\langle model \rangle$)	(Rule 7)
|	Iteration($\langle controller \rangle$,$\langle engine \rangle$)	(Rule 8)

Fig. 8 The Design of Iteration Operator



solution found. When there is a request for next solution, it returns a store by keeping parts of the best solution using certain heuristics.

The following example shows how to implement the approximation algorithm mentioned earlier, using our framework with the newly introduced engines:

```
1    ModelEngine(JSSP) →
2    First(TreeSearchEngine(...) →
3    Last(Iteration(Controller,(RelaxEngine(JSSP) →
4                                First(TreeSearchEngine(...))))) →
5    PrintEngine(JSSP)
```

The process starts by requesting next solution from this composite engine. The sequence of requests are cascaded to the *ModelEngine* (line 1), which passes the store containing the job shop scheduling model to the *TreeSearchEngine* (line 2) to look for the first feasible solution. Then, the feasible solution is passed to the composite engine (lines 3-4). The composite engine uses the *Iteration* operator to perform local optimization until the controller stopping condition is met. The local optimization is made up of a *RelaxEngine* that keeps random parts of the best solution and a *TreeSearchEngine* that looks for a better solution. The best solution is then passed to *PrintEngine* (line 5) for appropriate display.

8 Round Robin Tournament Scheduling

The literature on planning of intermural round robin sports tournaments [Cai77,dW88,Sch92] generally agrees on a decomposition of the scheduling process into three phases, namely pattern generation, pattern set generation and timetable generation. Recently, constraint programming has been applied to all three phases [Hen01] and achieved a significant improvement over integer programming on a difficult benchmark problem [NT98].

The first phase consists of generating all possible patterns, each of which indicate a possible sequence in which a team can play home or away in the round robin. The second phase consists of generating feasible sets of patterns, and the third phase consists of assigning opponent teams to time slots in the timetable, which finally leads to the desired round robin schedule.

This process can be cast in our compositional framework for search as follows:

```
1    ModelEngine(PatternSet(ModelEngine(Round-Robin) →
2                                TreeSearchEngine(...))) →
3    TreeSearchEngine(Naive,CopyingNode,DFS) →
4    TreeSearchEngine(FirstFail,CopyingNode,LDS) →
5    PrintEngine(...)
```

The overall model is generated by using an auxiliary engine (lines 1-2) which generates all patterns. The auxiliary engine represents the first phase of the solution process. The resulting patterns are used by the model engine to generate constraints for pattern sets. The second and third phases are represented by the engines in lines 3 and 4. Here, naive enumeration is used for pattern set generation and first-fail for the timetable generation.

9 Comparison with Related Works

This work relates closely to SALSA, a language for search algorithms [LC98], and the search part of OPL [Per99,VPP00].

The key behind SALSA is that search is essentially a transition from one state to another based on the choice made. By following this principle, SALSA main abstraction is the choice point. Choice points can be combined together to form a search algorithm. To provide for branch-and-bound optimization, there are ways to combine the choice points with a function evaluation. By using a choice point abstration, it makes it easy for SALSA to describe both local and global (tree) search. Unfortunately, as pointed out in [Per99], some search exploration like LDS requires an unnatural implementation. The key abstraction presented in this paper is an engine, which encapsulates not only the branching algorithm, but also the state restoration policy and the exploration algorithm. This leads to a more flexible way of composing different search phases. Engines encapsulate the internal aspects of branching, restoration and exploration, and thus allow the software designer to concentrate on the macro view, the composition of engines to form the overall solution. We hope that this information hiding leads to a more efficient way to design new innovative constraint-based search techniques.

The search part in OPL is similar to Perron's search procedures. The key idea in this case is the search goals. Beside goals, there are evaluators, selectors and limits. Evaluators provide the search exploration and selectors serve as filters. Goals are similar to engines, but do not encapsulate the state restoration policy. An important characteristic is that in our demand-driven model, an engine does not need to collect all the possible solutions before passing them onto the next engine. In other words, after a solution is found, we can pass the control to the next engine. Although we do not know the implementation details of OPL, our experience suggests that a demand-driven approach would yield better flexibility and reusability.

10 Conclusion and Future Work

We developed a compositional framework based on engine for describing complex search scenarios. Tree search was encapsulated within an engine. With the operators on engines, the framework allowed to build complex search engines by composing the different engines together. Examples and two case studies demonstrated the expressivity of our framework.

The compositional framework will probably incur some overhead to the overall system. The investigation to minimize such overhead is an interesting direction in the near future. Moreover, the future direction of this work is to realize the extension to other dimensions of search. Visualization is a specific extension that is of particular interest as there is a need to provide tools for performance debugging. The facilities and abstractions to provide parallel search based on the concept of engine are also worth investigation. Lastly, it is interesting to see if it is possible to provide engines that facilitate the integration of local search algorithms to tree search.

References

[Cai77] William O. Cain, Jr. The computer-assisted heuristic approach used to schedule the major league baseball clubs. In Shaul P. Ladany and Robert E. Machol, editors, *Optimal Strategies in Sports*, number 5 in Studies in Management Science and Systems, pages 32–41. North-Holland Publishing Co., Amsterdam, New York, Oxford, 1977.

[CHN00] Tee Yong Chew, Martin Henz, and Ka Boon Ng. A toolkit for constraint-based inference engines. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, Lecture Notes in Computer Science 1753, pages 185–199, Boston, MA, 2000. Springer-Verlag, Berlin.

[CHN01] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In Toby Walsh, editor, *Principles and Practice of Constraint Programming—CP 2001, Proceedings of the Seventh International Conference*, Lecture Notes in Computer Science, Cyprus, 2001. Springer-Verlag, Berlin. to appear.

[dW88] D. de Werra. Some models of graphs for scheduling sports competitions. *Discrete Applied Mathematics*, 21:47–65, 1988.

[Hen01] Martin Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 49(1):163–168, January 2001.

[HMN99] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, Las Cruces, New Mexico, USA, 1999. held in conjunction with ICLP'99.

[LC98] François Laburthe and Yves Caseau. SALSA: A language for search algorithms. In Michael Maher and Jean-François Puget, editors, *Principles and Practice of Constraint Programming*, pages 310–324, Pisa, Italy, 1998. Springer-Verlag, Berlin.

[Ng01] Ka Boon Kevin Ng. *A Generic Software Framework For Finite Domain Constraint Programming*. Master's thesis, School of Computing, National University of Singapore, 2001.

[NP98] Wim Nuijten and Claude Le Pape. Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3:271–286, 1998.

[NT98] George L. Nemhauser and Michael A. Trick. Scheduling a major college basketball conference. *Operations Research*, 46(1):1–8, 1998.

[Per99] Laurent Perron. Search procedures and paralleism in constraint programming. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, Alexandria, VA, USA, 1999. Springer-Verlag, Berlin.

[Sch92] Jan A. M. Schreuder. Combinatorial aspects of construction of competition dutch professional football leagues. *Discrete Applied Mathematics*, 35:301–312, 1992.

[Sch97] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[Sch00] Christian Schulte. *Programming Constraint Services.* Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000. To appear in Lecture Notes in Artificial Intelligence, Springer-Verlag.

[VPP00] Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.