Figaro: Yet Another Constraint Programming Library

Martin Henz, Tobias Müller, Ng Ka Boon

September 17, 1999

Abstract

Existing libraries and languages for finite domain constraint programming usually have depth-first search (with branch and bound) built-in as the only search algorithm. Exceptions are the languages CLAIRE and Oz, which support the programming of different search algorithms through special purpose programming language constructs. The goal of this work is to make abstractions for programming search algorithms available in a language-independent setting by using the concept of a room.

Figaro is an experimentation platform being designed to study non-standard search algorithms, different memory policies for search (trailing vs copying), consistency algorithms, failure handling and support for modeling. Figaro is conceived as a C++ library providing abstractions based on the concept of a room. This paper focuses on the use and implementation of such abstractions for investigating programmable search algorithms and memory policies in a C++ constraint programming library.

1 Introduction

Languages and libraries for finite-domain constraint programming (CP(FD)) allow to solve finite-domain problems through exhaustive constraint propagation, interleaved with non-deterministic strengthening of a constraint store, leading to the exploration of a search tree.

Languages for CP(FD) allow a semantic embedding of CP(FD)-specific features. Prolog-based languages such as CHIP [DVS+88] semantically embed depth-first search by inheriting Prolog's resolution, and the languages CLAIRE [CL96] and Oz [Smo95] semantically embed more generic constructs that allow to program search algorithms other than depth-first search.

Libraries such as PECOS [Pug92] and Ilog Solver [Pug94] are confined to general-purpose programming languages that do not provide such support. Such libraries currently do not support the programming of search algorithms that significantly deviate from trailing-based depth first search. We show in this work how to support programmable search algorithms in a C++ library. We use the design of rooms [HS99] and transfer it from ML to C++, a language without support for functions as first class values. We call the C++ library Figaro (FInite-domain GAdget with ROoms), since its implementation reuses parts of the Mozart system [Moz99].

We present the design of Figaro by introducing rooms, variables and propagators in Section 2, the notion of search trees in Section 3, and search algorithms in Section 4. Section 5 shows how relative addressing allows to use copying-based search in addition to trailing-based search. Finally, Section 6 describes current directions and perspectives of this work.

2 Variables and Propagators

The constraint store in CP(FD) contains the current domain of each variable of the constraint problem, i.e. the set of possible values it can take. For example, for the usual model of the n-queens problem, we introduce variables $x_i, 0 \leq i < n$ whose initial domains $\{0, \ldots, n-1\}$ represent all possible rows in which the queens of column i can be positioned. In [HS99], a data structure called room is introduced which allows to program search algorithms in a library-based setting. Rooms host variables and propagators and support search. In an object-oriented setting, is natural to introduce a class `room`. Variables are introduced by requesting a new variable with initial domain from `lo` to `hi` from a room.

```
class room {
private: ...
public:
    var newvar(int lo,int hi);
    var getlo(var v);
    var gethi(var v);
...
};
```

For the purpose of this discussion, let us assume that `var` is an abstract data type whose values represent variables. In Section 5, we further discuss the `var` type.

Using the room abstraction, we can introduce variables for the n-queens problem as follows. Here we employ vectors as provided by the Standard Template Library [SL95] for C++.

```
int main(int argc,char *argv[]) {
  int n = atoi(argv[1]);              // number of queens
  room* r = new room();               // create new room
  vector<var> vars(n);                // declare variable vector
  for (int i=0;i<n;i++)               // create n variables;
    vars[i]=r->newvar(0,n-1);         //   one for each row
  ...
}
```

The no-attack constraints can be expressed using three constraints that constrain all variables in a given vector to be pairwise distinct modulo a given offset. Thus given a vector *offset* of 5 integers, the constraint *distinctOffset*(*vars*, *offset*) expresses that for

Program 1 Constraints for the N-Queens Problem

```
int main(int argc,char *argv[]) {
  int n = atoi(argv[]);                  // number of queens
  room* r = new room();                 // create new room
  vector<var> vars(n);                  // declare variable vector
  for (int i=0;i<n;i++)                 // create n variables;
    vars[i]=r->newvar(0,n-1);           //    one for each row
  vector<int> offset(n);                // vector for offset
  for (int i=0;i<n;i++) offset[i]=0;    // horizontal no-attack
  distinctOffset(r,vars,offset);
  for (int i=0;i<n;i++) offset[i]=i;    // diagonal-up no-attack
  distinctOffset(r,vars,offset);
  for (int i=0;i<n;i++) offset[i]=n-i; // diagonal-down no-attack
  distinctOffset(r,vars,offset);
  ...
}
```

every i and j, where $i \neq j$, $vars_i + offset_i \neq vars_j + offset_j$ holds. The implementation of the propagator `distinctOffset` is taken from the C++ constraint programming interface of the Mozart system [MW97]. Reusing Mozart's propagators significantly reduces the implementation effort for Figaro.

In Figaro, constraints are represented by classes which extend an abstract class `propagator`. Propagators are created with a given room, variables and auxiliary values.

```
class distinctOffset : public propagator {
public:
   distinctOffset(room *ro, vector<var>, vector<int>);...}
```

Using the class `distinctOffset`, the 5-queens problem can be expressed as in Program 1.

The creation of propagators will immediately compute the fixpoint with respect to all propagators in the room, according to the propagators' consistency algorithms. In this process, propagators may tell new domains for variables.

The member function `tell` of rooms allows to narrow the domain d_1 of a given variable such that it contains only values from the domain d_2 passed to `tell`. If the intersection of d_1 and d_2 is empty, a failure occurs.

```
room::tell(var v, int lo, int hi);
```

If the intersection of d_1 and d_2 is empty, a failure occurs. Such failures are crucial for constraint programming, since they allow to prune the search tree. As a generic way to indicate failure to search algorithms, failing `tell` operations raise the C++ exception `Failure()`.

3 Search Trees

Usually propagation alone does not suffice to solve constraint problems. Non-deterministic search is necessary, which explores a search tree in a top-down manner. From a node to a child node, constraints are added. At each node, the fixpoint with respect to all propagators is reached before the resulting constraint store is used to devise a suitable constraint for a child node. In that manner, search trees are created dynamically, at each point exploiting the current information in the constraint store. Search trees are represented in Figaro using instances of an abstract class `node`.

```
class node {
public:
  virtual node* child(room *, int)=0;
};
```

The member function `child` of `node` is given an integer i and returns its i^{th} child. Often, search trees are constructed by fixing one variable v of a given set of variables to a value x in the left child ($i = 0$) and excluding x from the domain of v in the right child ($i = 1$). Such a tree is called enumeration tree. The class in Program 2 represents naive enumeration, where the variables of a given vector are enumerated from left to right, starting with the smallest values in their domains.

By recursively applying the `child` function to it results, we are able to explore an enumeration tree. The tree `subtree` is returned when all variables are enumerated. It can be used to place another search tree at the leaves of the enumeration tree, or to collect solutions. For example, the class in Program 3 allows to display a solution to the n-queens problem.

Program 2 Naive Variable Enumeration

```
class naive : public node {
private:
    int idx; vector<var> vars; node * subtree;
public:
  naive(vector<var> vs,int i,node* t) : vars(vs), idx(i), subtree(t) {}
  node* child(room * r, int i) {
    if (i==0) {
      r->tell(vars[idx],r->getlo(vars[idx]),r->getlo(vars[idx]));
      return (idx+1==vars.size() ? subtree : new naive(vars,idx+1,subtree));
    }
    else {
      r->tell(vars[idx],r->getlo(vars[idx])+1,r->gethi(vars[idx]));
      return new naive(vars,idx,subtree);
    }
  }
};
```

Program 3 A Node Class for Printing Solutions

```
class queens_printer : public node {
private:
  vector<var> vars;
public:
  queens_printer(vector<var> vs) : vars(vs) {
      for (int j = 0; i < vars.size(); j++)
      cout<<"col: "<<vars[i]<<"\nrow: "<<r->getlo(vars[i])<<"\n"; }
  node* child(room * r, int i) {return NULL;}
}
```

Note that such `queens_printer` nodes are leaves, because their `child` member function returns the `NULL` pointer. Thus, `queens_printer` nodes can be used as `subtree` of enumeration trees.

4 Programming Inference Algorithms

During the exploration of a search tree, failure may occur as a result of applying the `child` function of a node. That means one of the decisions leading to the corresponding node was the wrong one. Unfortunately, after that decision was taken the room has changed through creating variables and propagators and telling domains. In order to undo these changes and trying an alternative, we introduce the following operations on rooms.

```
mark room::mark();
void room::backtrack(mark m);
```

The function `room::mark` returns a value that represents the current state of the room, and the function `backtrack` undoes all changes done to the room since the given mark was obtained. A search algorithm using `room::mark` and `room::backtrack` is given in Program 4.

Program 4 First-solution Depth-first Search

```
node* solve_one(room* r,node* t) {
  if (t == NULL) return t;
  int m = r->mark();
  try {return solve_one(r,t->child(r,0));}
  catch (Failure) {
    r->backtrack(m);
    return solve_one(r,t->child(r,1));
  }
}
```

Program 5 Solving N-Queens with Figaro

```
int main(int argc,char *argv[]) {
  int n = atoi(argv[1]);
  room* r = new room();
  vector<var> vars(n);
  ...
  try {solve_one(r,new naive(vars,0,new queens_printer(vars)));
  } catch (Failure) {printf("no solution\n");};
}
```

Using this search algorithm, we are finally able to solve the n-queens problem as shown in Program 5.

Note that in the exposition above, we used several simplifications to clarify the design underlying Figaro. Both enumeration and search can be improved significantly by introducing additional tests and member functions. For instance, the creation of node objects can be avoided, when choosing the left child node of an enumeration node by incrementing the `idx` member of the parent.

5 Copying-Based Search and Relative Addressing

Note that in the previous section the same room is passed between `solve_one` and `child`. Search is done entirely by trailing and backtracking, as in most constraint programming systems. Schulte [Sch99] shows that copying-based search as employed by the Mozart system, combined with recomputation of spaces, can compete with the performance of trailing-based systems. To study the performance of memory policies, it appears to be attractive to provide both copying and trailing in the same system. In order to support copying-based search, we use a suitable C++ copy constructor.

```
class room {
 ...
public:
 room(const &room);
}
```

For combining copying and trailing-based search, we propose that in the copy, all marks are removed and that no information is trailed in a room before the first mark is obtained.

Since we pass the room on which a node operates explicitly to the room, it is straightforward to use copying-based search in our setup. We illustrate this using example of limited discrepancy search (LDS), a search algorithm proposed by Harvey and Ginsberg [HG95]. LDS addresses the question how to avoid getting stuck in a small leftmost subtree in the presence of a strong heuristic for building the search tree. Let us assume that a script uses a heuristic which generates binary nodes whose left child are considered much more likely to lead to a solutions than the right child. Then the number of

Program 6 A Copying-based Search Algorithm For Limited Discrepancy Search

```
node* probe(room* r,node* t,int d) {
  if (t==NULL) return t;
  if (d > 0) {
    room* r1 = r->copy();
    try {return probe(r,t->child(r,1),d-1);}
    catch (Failure) {
      return probe(r1,t->child(r,0),d);}
  } else return probe(r,t->child(r,0),0);
}
void lds_one(room* r,node* t,int d) {
  try {return probe(r,t,d);}
  catch (Failure) {lds_one(r,t,d+1);}
}
```

discrepancies of a solution is the number of right children in the path from the root to the solution. LDS prescribes to search for solutions with a small number of discrepancies first.

The algorithm `lds_one` given in Program 6 searches for a solution according to LDS—assuming that a solution exists—with increasing number of discrepancies, starting with a given `d`, typically 0. The auxiliary function `probe` returns a solution within a given number of discrepancies `d`, if such a solution exists. Note that once the number of allowed discrepancies has reached `0`, there is no need to make copies any longer. Instead, `probe` descends straight down towards a solution.

In copying-based search, tree descriptions will use `var` values that stem from calls of `newvar` on a room and apply operations such as `tell` to *copies* of the room. Thus variables must be invariant with respect to copying. We achieve this invariance by using as `var` values the relative address of the variable in the room data structure. In rooms, a dynamic array keeps track of variables, variables are represented by their indices in this array.

```
#define var int
```

The same technique, we use for propagators; trail entries and propagator lists for variables use relative addresses. Note that relative addressing makes copying particularly easy, whereas absolute addressing as employed by the Mozart system necessitates recursive traversal of data structures in rooms similar to copying garbage collection.

6 Directions and Perspectives

The advantage of Figaro over existing libraries for finite domain constraint programming is the programming of search algorithms, allowing to experiment with innovative search algorithms.

Figaro is currently being implemented by reusing parts of the Mozart system. In particular Mozart's sophisticated representation of domains and its propagation algorithms such as serialization and cummulative constraints for scheduling applications are being reused. We hope by this, we can reduce the development time of Figaro so that a first version can be released in late 1999.

Memory Policy

Relative addressing of variables and propagators in rooms allows to use copying-based search in addition to trailing. We illustrated this with a copying-based limited discrepancy search. Both copying-based and trailing-based search can be combined with recomputation [Sch99]. We hope that the flexibility to use both memory policies leads to interesting, possibly adaptive, and more efficient combinations of the two memory policies and recomputation.

Another memory management issue is the creation of nodes, which are in the presented simplified design not explicitly deallocated. We are currently experimenting with search algorithms that explicitly deallocate nodes as well as with automatic memory management systems for C++.

Consistency Algorithms

Different constraint programming languages and libraries use different consistency algorithms, usually variants of AC3 and AC5. In practice, the trade-off between the two seems to be to use more elaborate data structures and reduce pure computation time (AC5) versus simpler data structure and redundant computations (AC3). This trade-off becomes interesting in the light of a widening gap between computation speed and memory access speed in modern processors. We hope to carry out practical experiments on consistency algorithms with Figaro.

Representation of Failure

As in the original room design, we treated failure by exceptions in this paper. Another possibility would be to let rooms assume a failure state, when an exception is encountered. This design issue is not settled yet and depends on practical considerations such as the efficiency of exception handling in C++ compilers. Benchmarks will shed some light on the efficiency of the two mechanisms for failure.

Modeling and Interfacing through Scripting Languages

Figaro is designed as a C++ library for constraint programming. Typically, the library will be linked to applications that make use of constraint programming for problem solving. However, in some applications the need for a more formal and flexible formulation of constraint problems arises. This need is addressed in symbolic programming languages for constraint programming (Prolog-based, Oz, CLAIRE) and in modeling languages for constraint programming such as OPL [Hen99]. To address this need, we provide a generic

interface to scripting languages such as Tcl and Perl. In addition to modeling, the use of scripting languages aids the development of and experimentation with the library and improves interoperability.

Acknowledgements

Gert Smolka collaborated on the development of the room concept and corresponding abstractions in an ML setting, which provided a blueprint for rooms in C++. Christian Schulte helped us see the importance of indirect addressing and pointed out references. The project benefited from a travel grant from the National University of Singapore (project ReAlloc) and the hosting of the third author by the Programming Systems Lab, Saarbrücken.

References

[CL96] Yves Caseau and François Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming*. TU Berlin, 1996.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Proceedings International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, December 1988. Springer-Verlag.

[Hen99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, MA, 1999.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers, San Mateo, CA.

[HS99] Martin Henz and Gert Smolka. Design of a finite domain constraint programming library for ML. draft available at `http://www.comp.nus.edu.sg/ henz/drafts/room.ps`, 1999.

[Moz99] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[MW97] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Maluszyński, editor, *Logic Programming: Proceedings of the 1997 International Symposium*, pages 149–163, Long Island, NY, USA, 1997. The MIT Press.

[Pug92] Jean-François Puget. PECOS: A high level constraint programming language. In *Proceedings of the First Singapore International Conference on Intelligent Systems (SPICIS)*, pages 137–142, Singapore, September/October 1992.

[Pug94] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, November 1994.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of the International Conference on Logic Programming*, 1999. to appear.

[SL95] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard, 1995. STL has since been incorporated into the C++ standard, ISO/IEC 14882-1998.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.