Chronological Backtracking versus Formal Methods for Solving CSPs

Malek Mouhoub, Samira Sadaoui and Amrudee Sukpan
Department of Computer Science, University of Regina
3737 Waskana Parkway, Regina SK, Canada, S4S 0A2
{mouhoubm, sadaouis, sukpan1a}@cs.uregina.ca

Abstract
The aim of this paper is to compare two techniques for solving constraint satisfaction problems(CSPs). The first one uses constraint propagation and chronological backtracking algorithm whereas the second one is based on LOTOS specifications. The language LOTOS combines a process calculus with abstract data types. The data part specifies the different constraints of a given CSP. The process part corresponds to the description of the resolution process, such as the constraint propagation and backtracking.

For a given problem, the simulation of the corresponding LOTOS specification can generate zero, one or all possible solutions. No solution means that the specification leads to a deadlock. In the other hand, with the model checking, we can verify if a certain path is a solution, and also complete a partial solution in an incremental way.

Keywords: CSP, constraint propagation, formal methods, LOTOS.

1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem [8, 7] involves a list of variables defined on finite domains of values, and a list of relations restricting the values that the variables can take. If the relations are binary we talk about binary CSPs. Solving a CSP consists of finding an assignment of values to each variable such that all relations (or constraints) are satisfied. When solving a CSP, we may want to find :

- just one solution, with no preference as to which one,
- all solutions,
- an optimal, or at least a good solution, given some objective function defined in terms of some or all the variables.

A CSP is known to be an NP-Hard problem. Indeed, looking for a possible solution to a CSP requires a backtrack search algorithm of exponential complexity in time[1]. To overcome this difficulty in practice, local consistency techniques are used in a pre-processing phase to reduce the size of the search space before the backtrack search procedure. A k-consistency algorithm removes all inconsistencies involving all subsets of k variables belonging to N. The k-consistency problem is polynomial in time, $O(N^k)$, where N is the number of variables. A k-consistency algorithm does not solve the constraint satisfaction problem, but simplifies it. Due to the incompleteness of constraint propagation, in the general case, search is necessary to solve a CSP problem, even to check if a single solution exists. When $k = 2$ we talk about arc consistency. An arc consistency algorithm [8, 2, 3] transforms the network of constraints into an equivalent and simpler one by removing, from the domain of each variable, some values that cannot belong to any global solution.

2 Example: the N-Queen Problem

The problem consists here in placing N queens on a NxN chessboard satisfying the constraint that two queens should not threaten each other. A queen attacks all cells in its same row, column or diagonal as shown in figure 1 (a) in the particular case where N=4. This problem can be converted to an equivalent and a simpler one by restricting each queen to be placed on a different row (or column) (see figure 1 (b)). This way the number of possibilities each queen has will be 4 instead of 16 (N instead of NxN in the case of the N-queen

[1]Note that some CSP problems can be solved in polynomial time. For example, if the constraint graph corresponding to the CSP has no loops, then the CSP can be solved in $O(nd^2)$ where n is the number of variables of the problem and d the domain size of the different variables

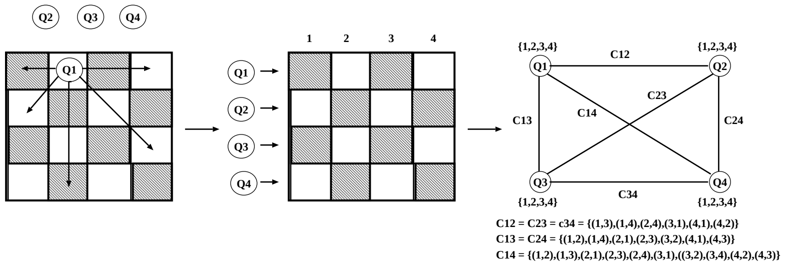
Q2
Q3
Q4
Q1
1
2
3
4
Q1
Q2
Q3
Q4
{1,2,3,4}
C12
{1,2,3,4}
C23
C14
C13
C24
{1,2,3,4}
C34
{1,2,3,4}
C12 = C23 = c34 = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
C13 = C24 = {(1,2),(1,4),(2,1),(2,3),(3,2),(4,1),(4,3)}
C14 = {(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),((3,2),(3,4),(4,2),(4,3)}

Figure 1: The 4-Queen Problem.

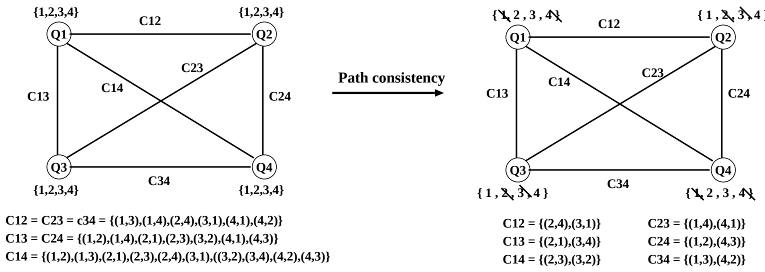

Figure 2: Application of the path consistency to the 4-Queen Problem.

problem). The transformation to a CSP is then straight forward as shown in figure 1.

As mentioned above, the resolution method of solving a CSP is divided into the following two stages :

- In this phase local consistency techniques are performed in order to reduce the size of the search space. Usually these techniques are arc and path consistency (equivalent to 3-consistency) [8, 9] algorithms. In the case of the N-queen problem, only the path consistency is useful. The application of the path consistency algorithm on the 4-queens is shown in figure 2.

- After the local consistency phase is achieved, the backtrack search algorithm is performed to look for a possible numeric solution. Arc consistency is also used during this phase following the principle of the forward check strategy [6] in order to allow branches of the search tree that will lead to failure to be pruned earlier than with simple backtrack. More precisely, the backtrack search algorithm works as follows :

  Choose a node and instantiate the corresponding variable (that we call current variable) to a value (numeric interval) belonging to its domain. Discard from the variable domain the remaining values and run the arc consistency algorithm between the current variable and the non instantiated variables (called future variables). If the network is arc consistent, fix a value on another variable and run again the arc consistency algorithm until each value is fixed on the domain of every variable of the network. We obtain a solution corresponding to the set of numeric intervals fixed on the domain of each variable. If the network does not succeed (is not arc consistent) at some point, choose another value of the domain of the last selected variable. If there is no value to be considered, backtrack and choose another value from the domain of the previous variable.

  During the backtrack search, the following properties are in general used to select the different variables and values :

  – Choose the most constrained variable to assign next. This can be determined by the number and type of the different relations connected to each variable.

  – Choose the least constraining value for each variable.

The application of the backtrack search algorithm with the forward check strategy is illustrated in figure 3. Note that, in this example each queen is placed on a different column and not different row as mentioned previously.

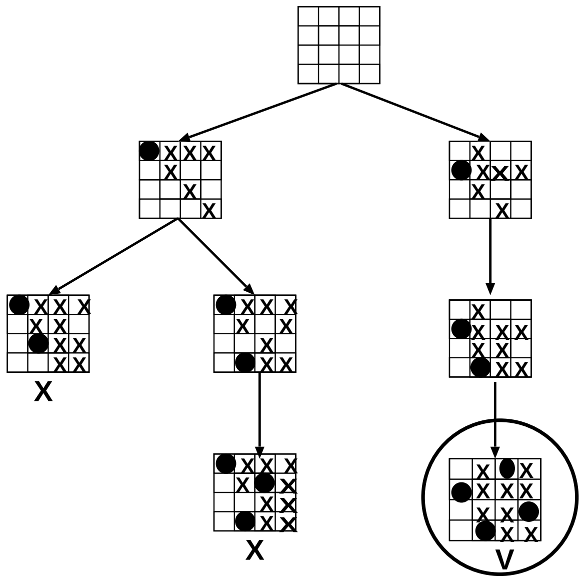

Figure 3: Application of the backtrack search algorithm to the 4-Queen Problem.

3 Formal Methods

The language LOTOS (Language Of Temporal Ordering Specification) is used to represent (specify) and solve (execute) the specified CSP. LOTOS is the ISO standardized formal specification technique for describing concurrent and distributed systems [1]. The specification in LOTOS gives the temporal ordering of actions i.e., the relative ordering of actions in time. LOTOS combines a process calculus with an abstract data type language [4].

The abstract data type (ADT) part specifies the different constraints of a given CSP, their corresponding variables and domains. More precisely, the ADT describes the static part of a system i.e., data structures and value expressions. An ADT definition identifies an algebra formed by sets of data values or domains, and a set of associated operations. The definition also includes equations (axioms) which are equalities between terms. Only positive conditional equalities are allowed.

The process part, describing processes or behavior expressions, defines the external visible behavior of a system. In the case of CSPs, it corresponds to the description of the resolution process. Behavior expressions are built using the process operators, such as the action prefix **;** which denotes the sequence of actions, and **exit** the successful termination of a specification. An action (or interaction) can be followed by the construct **?** in order to input values from the environment e.g., `QUEEN1?p:position` expresses that the first queen is placed on the position p in the chessboard; p has two coordinates given by the user. In LOTOS, the unary constraints are defined using the selection predicates. Such predicates are meant to restrict the values offered within an action. For instance, `g?x:nat [x<2]` means that the domain of the variable x is restricted to the set $\{1, 2, 3\}$.

The specifications of the N-Queen problem are built with the monolithic style where only observable interactions are presented and ordered as a collection of alternative sequences of actions in branching time. The specifications are executable because of the operational semantics of LOTOS. There are many supporting tools for LOTOS such as the EUCALYPTUS environment [5]. The simulation of the specifications generates zero, one or all possible solutions. No solution means that the specification leads to a deadlock. In the other hand, the model checker can verify if a certain path is a solution, and also complete a given partial solution in an incremental way. We present in figures 4 and 5 two specifications for the 4-Queen problem: the first one checks whether the problem is consistent, and the second one supports the interactive consistency check.

3.1 Global Consistency Check

In this case, we are interested in generating one or all possible solutions. The specification that checks the consistency of the 4-Queen problem is given in figure 4. For the 4-Queen problem, we have only two solutions that are produced by the standard simulation:

```
specification Queens[QUEEN]:exit
library Boolean endlib
type Natural is Boolean
sorts nat
opns
     0 (*! constructor *): -> nat
     1 (*! constructor *): -> nat
     2 (*! constructor *): -> nat
     3 (*! constructor *): -> nat
     4 (*! constructor *): -> nat
     _eq_:nat,nat-> bool
     _ - _ : nat, nat  -> nat
eqns
     forall x:nat
     ofsort bool
     x eq x = true;
     0 eq 1 = false; 0 eq 2 = false; 0 eq 3 = false; 0 eq 4 = false;
     1 eq 0 = false; 1 eq 2 = false; 1 eq 3 = false; 1 eq 4 = false;
     2 eq 0 = false; 2 eq 1 = false; 2 eq 3 = false; 2 eq 4 = false;
     3 eq 0 = false; 3 eq 1 = false; 3 eq 2 = false; 3 eq 4 = false;
     4 eq 0 = false; 4 eq 1 = false; 4 eq 2 = false; 4 eq 3 = false;
     ofsort nat
     0-x=x; x-0=x;
     x-x=0;
     1-2=1; 1-3=2; 1-4=3;
     2-1=1; 2-3=1; 2-4=2;
     3-1=2; 3-2=1; 3-4=1;
     4-1=3; 4-2=2; 4-3=1;
endtype

type position is Natural, Boolean
sorts position
opns
    put (*! constructor *): nat,nat -> position
    nattack: position, position -> bool
    _neqp_: position, position-> bool
    nthp: position, nat ->bool
eqns
    forall c1,c2:nat, r1,r2:nat
    ofsort bool
    put(c1,r1) neqp put(0,0) = (not(c1 eq 0)) and (not(r1 eq 0));

    nthp(put(c1,r1), c1) = true;
    not((c1-c2) eq 0) => nthp(put(c1,r1),c2)=false;

    (((c1-c2) eq 0) or ((r1-r2) eq 0)) or ((r1-r2) eq (c1-c2)) =>
    nattack(put(c1,r1),put(c2,r2))=false;
    not((((c1-c2) eq 0) and ((r1-r2) eq 0)) and ((r1-r2) eq (c1-c2)))
    =>nattack(put(c1,r1),put(c2,r2))=true;
endtype

behaviour
   QUEEN ?p1,p2,p3,p4:position
   [(p1 neqp put(0,0)) and (p2 neqp put(0,0)) and (p3 neqp put(0,0))
     and (p4 neqp put(0,0)) and nthp(p1,1) and nthp(p2,2) and nthp(p3,3)
     and nthp(p4,4) and nattack(p1,p2) and nattack(p1,p3) and nattack(p1,p4)
     and nattack(p2,p3) and nattack(p2,p4) and nattack(p3,p4)];
     exit
endspec
```

Figure 4: Global Consistency Check.

```
specification Queens[QUEEN1,QUEEN2,QUEEN3,QUEEN4]:exit
library Boolean endlib
type Natural is Boolean
sorts nat
opns
     0 (*! constructor *): -> nat
     1 (*! constructor *): -> nat
     2 (*! constructor *): -> nat
     3 (*! constructor *): -> nat
     4 (*! constructor *): -> nat
     _eq_:nat,nat-> bool
     _ - _ : nat, nat  -> nat
eqns
     forall x:nat
     ofsort bool
     x eq x = true;
     0 eq 1 = false; 0 eq 2 = false; 0 eq 3 = false; 0 eq 4 = false;
     1 eq 0 = false; 1 eq 2 = false; 1 eq 3 = false; 1 eq 4 = false;
     2 eq 0 = false; 2 eq 1 = false; 2 eq 3 = false; 2 eq 4 = false;
     3 eq 0 = false; 3 eq 1 = false; 3 eq 2 = false; 3 eq 4 = false;
     4 eq 0 = false; 4 eq 1 = false; 4 eq 2 = false; 4 eq 3 = false;
     ofsort nat
     0-x=x; x-0=x; x-x=0;
     1-2=1; 1-3=2; 1-4=3; 2-1=1; 2-3=1; 2-4=2;
     3-1=2; 3-2=1; 3-4=1; 4-1=3; 4-2=2; 4-3=1;
endtype

type position is Natural, Boolean
sorts position
opns
      put (*! constructor *): nat,nat -> position
      nattack: position, position -> bool
      _neqp_: position, position-> bool
eqns
      forall c1,c2:nat, r1,r2:nat
      ofsort bool
      (((c1-c2)eq 0)o((r1-r2)eq0))or((r1-r2)eq(c1-c2))=>
      nattack(put(c1,r1),put(c2,r2))=false;
      not((((c1-c2) eq 0)and((r1-r2)eq0))and((r1-r2)eq(c1-c2)))=>
      nattack(put(c1,r1),put(c2,r2))=true;
      put(c1,r1) neqp put(0,0) = (not(c1 eq 0)) and (not(r1 eq 0));
endtype

behaviour
  QUEEN1 ? p1:position [p1 neqp put(0,0)];
  QUEEN2 ? p2:position [p2 neqp put(0,0) and nattack(p1,p2)];
  QUEEN3 ? p3:position [p3 neqp put(0,0) and nattack(p1,p3) and nattack(p2,p3)];
  QUEEN4 ? p4:position [p4 neqp put(0,0) and nattack(p1,p4) and nattack(p2,p4)
                      and nattack(p3,p4)];
  exit
endspec
```

Figure 5: Interactive Consistency Check.

- `put(1, 2), put(2, 4), put(3, 1), put(4, 3)`
- `put(1, 3), put(2, 1), put(3, 4), put(4, 2)`

If there is no solution, the simulation directly leads to a deadlock (no progress is no more possible). In the other hand, with the random simulation, the EUCALYPTUS tool generates randomly only one solution, if it exists.

In LOTOS, with the model checking, the user can verify if a given path (assignment of values to variables) is a solution. For instance, the user can check if the following behavior is correct: `put(2, 1), put(4, 4), put(1, 2), put(3, 3)`. In this case, the answer is no.

The model checker can also complete a given partial solution in an incremental way. This case consists of giving values for some variables, and then the tool generates all the possible values for the rest of the variables. For instance, after creating the following behavior: `put(1, 1), put(4, 2)`, the tool produces all the possible solutions (depth-first search), or only one solution (breath-first search), for instance `put(3, 4), deadlock`.

3.2 Interactive Consistency Check

In figure 5, we give the specification which fellows the forward check principle mentioned in section 3, and illustrated in figure 3. The specifier progressively enters the positions of each of the four queens. We note that a queen can be placed in any position in the chessboard. A queen is placed if it is not attacked by the other already positioned queens.

The simulation of the specification can lead to a solution or not. No solution means that the simulation leads to a deadlock because the third or fourth queen can not be placed. For example, if the user assigns the first queen to (1, 1), the second one to (4, 2), and the third one to (3, 4), then the specification leads to a deadlock since the fourth queen cannot be placed.

With 4 queens, we have 6 constraints, and with n queens, we have n*(n-1)/2 constraints. In the LOTOS specifications, the constraints are specified with the predicate *nattack* defined in the data part. For instance, *nattack(p1, p2)* expresses that the queen placed at position p1 should not attack the queen at position p2. This constraint is associated with the constraint $C12$ given in figure 2.

4 Conclusion

In this paper, we have shown how LOTOS specifications support the CSP techniques such as global consistency and constraint propagation.

Our future work consists of:

- Comparing the efficiency, in terms of time and memory space costs, of the C code automatically generated from the LOTOS specifications with the algorithms defined in CSP.
- Studying the language E-LOTOS (Extended-LOTOS) to describe and solve constraint satisfaction problems in general and those based on temporal constraints in particular.

References

[1] *ISO LOTOS- A Formal DEscription Technique Based on The Temporal Ordering of Observational Behaviour*. International Organization for Standardization- Information Processing Systems Open Systems Interconnection, Geneve, July 1987.

[2] C. Bessière, E. Freuder, and J. Regin. Using inference to reduce arc consistency computation. In *IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[3] C. Bessière and J. C. Régin. Refining the basic constraint propagation algorithm. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 309–315, Seattle, WA, 2001.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *in P.H.J. van Eijkand, C.A. Vissers and M. Diaz, eds., The Formal Description Technique LOTOS (North-Holland, Amsterdam) 303-326*, 1989.

[5] H. Garavel. *An Overview of the EUCALYPTUS Toolbox*. http://www.inrialpes.fr/vasy/cadp/, June 1996.

[6] R. Haralick and G. Elliott. Increasing tree search efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[7] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[8] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[9] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.