Automated Exercises for Constraint Programming

Johannes Waldmann

HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany

Abstract. We describe the design, implementation, and empirical evaluation of some automated exercises that we are using in a lecture on Constraint Programming. Topics are propositional satisfiability, resolution, the DPLL algorithm, with extension to DPLL(T), and FD solving with arc consistency. The automation consists of a program for grading student answers, and in most cases also a program for generating random problem instances. The exercises are part of the `autotool` E-assessment framework. The implementation language is Haskell. You can try them at `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=199`.

1 Introduction

I lecture on Constraint Programming [Wal14a], as an optional subject for master students of computer science. The lecture is based on the books *Principles of Constraint Programming* by Apt[Apt03] and *Decision Procedures* by Kroening and Strichman [KS08]. Topics include propositional satisfiability (SAT) and resolution, the DPLL algorithm for deciding SAT, with extension to DPLL(T) for solving satisfiability modulo theory (SMT), and FD solving with arc consistency.

I do want to assign homework problems, but I do not have a teaching assistant for grading. This is one motivation for writing software that automates the grading of solutions, and also the generation of random (but reasonable) problem instances. Another motivation is that the software is much more available, reliable and patient than a human would be, so I can pose homework problems that would require super-human teaching assistants.

This software is part of the `autotool` E-assessment framework [GLSW11,RRW08]. It provides the following functionality (via a web interface):

- the tutor can choose a problem type, then configure parameters of an instance generator,
- the student can view "his" problem instance (that had been generated on first access) and enter a solution candidate,
- the grading program immediately evaluates this submission and gives feedback, sometimes quite verbose,
- based on that, the student can enter modified solutions, any number of times. The exercise counts as "solved" if the student had at least one correct solution during a given time interval (say, two weeks).

A distinctive feature is that `autotool` exercises are graded "semantically" — as opposed to "schematically", by syntactic comparison with some prescribed master solution. E.g., for propositional satisfiability, the student enters an assignment, and the program checks that it satisfies all clauses of the formula, and prints the clauses that are not satisfied (see more detail in Section 2).

In the language of complexity theory, the student has to find a witness for the membership of the problem instance in a certain problem class, and the software just verifies the witness. In many cases, the software does not contain an actual solver for the class, so even looking at the source code does not provide shortcuts in solving the exercises. But see Section 5 for an example where a solver is built in for the purpose of generating random but reasonable instances.

Section 6 shows an example for an "inverse" problem, where the witness (a structure that is a model) is given, and the question (a formula of a certain shape) has to be found.

If the software just checks a witnessing property, then it might appear that it cannot check the way in which a student obtained the witness. This seems to contradict the main point of lecturing: it is about methods to solve problems, so the teacher wants to check that the methods are applied properly. In some cases, a problem instance appears just too hard for brute force attempts, so the only way of solving it (within the deadline) is by applying methods that have been taught.

Another approach for designing problems is presented in Sections 5,7,8. There, the solution is a sequence of steps of some algorithm, e.g., Decide, Propagate and Backtrack in a tree search, and the witnessing property is that each step is valid, and that the computation arrives in a final state, e.g., a solution in a leaf node, or contradiction in the root. Here, the algorithm is non-deterministic, so the student must make choices.

Another feature of `autotool` is that most output and all input is textual. There is no graphical interface to construct a solution, rather the student has to provide a textual representation of the witness. This is by design, the student should learn that every object can be represented as a term. Actually, we use Haskell syntax for constructing objects of algebraic data types throughout.

For each problem type, the instructor can pose a fixed problem instance (the same for all students). For most problem types, there is also a generator for random, but reasonable instances. Quite often, the generator part of the software is more complicated than the grading part. Then, each student gets an individual problem instance, and this minimizes unwanted copying of solutions.

For fixed problem instances, `autotool` can compute a "highscore" list. Here, correct solutions are ranked by some (problem-specific) measure, e.g., for resolution proofs, the number of proof steps. Some students like to compete for positions in that list and try to out-smart each other, sometimes even writing specialized software for solving the problems, and optimizing solutions. I welcome this because they certainly learn about the problem domain that way.

In the following sections, I will present exercise problems. For each problem type I'll give

– the motivation (where does the problem fit in the lecture),
– the instance type, with example,
– the solution domain type,
– the correctness property of solutions,
– examples of system answers for incorrect solution attempts,
– the parameters for the instance generator (where applicable).

The reader will note that the following sections show inconsistent concrete syntax, e.g., there are different representations for literals, clauses, and formulas. Also, some system messages are in German, some in English. These inconsistencies are the result of incremental development of the `autotool` framework over > 10 years. The exercises mentioned in this paper can also be tried online (without any registration) at `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=199`

2 Propositional Satisfiability

– instance: a propositional logic formula F in conjunctive normal form,
– solution: a satisfying assignment for F

Motivation: At the very beginning of the course, the student should try this by any method, in order to recapitulate propositional logic, and to appreciate the NP-hardness of the SAT problem, and (later) the cleverness of the DPLL algorithm. We use the problem here to illustrate the basic approach.

Problem instance example:

```
   (  p ||   s ||   t)  && (  q ||   s ||   t) && (  r || ! q || ! s)
&& (  p ||   t || ! r)  && (  q ||   s ||   t) && (  r || ! s || ! t)
&& (  p || ! q || ! s)  && (  q ||   t || ! p) && (  s ||   t || ! q)
&& (  p || ! q || ! t)  && (  q || ! p || ! s) && (  s || ! r || ! t)
&& (  p || ! r || ! t)  && (  r ||   s || ! q) && (! p || ! q || ! r)
&& (  q ||   r || ! p)  && (  r || ! p || ! t) && (! p || ! r || ! s)
```

Problem solution domain: partial assignments, example

```
listToFM [ ( p , False ), (q, True), (s,True) ]
```

Correctness property: the assignment satisfies each clause.

Typical system answers for incorrect submissions: the assignment is partial, but already falsifies a clause

```
gelesen: listToFM
    [ ( p , False ) , ( q , True ) , ( s , True ) ]
Diese vollständig belegten Klauseln sind nicht erfüllt:
    [ (  p || ! q || ! s) ]
```

No clause is falsified, but not all clauses are satisfied:

```
gelesen: listToFM
    [ ( p , False ) , ( s , False ) , ( t , True ) ]
Diese Klauseln noch nicht erfüllt:
    [ (  p || ! q || ! t) , (  p || ! r || ! t) , (  r ||   s || ! q)
    , (  s || ! r || ! t) ]
```

Instance generator: will produce a satisfiable 3-SAT instance according to algorithm `hgen2` [SDH02]. Parameters are the set of variables, and the number of clauses, for example,

```
Param { vars = mkSet [ p , q , r , s , t ] , clauses = 18 }
```

3 SAT-equivalent CNF

- instance: formula F in conjunctive normal form with variables $x_1, \ldots, x_n$,
- solution: formula G in conjunctive normal form with variables in $x_1, \ldots, x_n, y_1, \ldots, y_k$ such that $\forall x_1 \cdots \forall x_n : (F \leftrightarrow \exists y_1 \cdots \exists y_k : G)$
- measure: number of clauses of G.

Motivation: for arbitrary F, this is solved by the Tseitin transform [Tse70]. The student learns the underlying notion of equivalence, and that auxiliary variables may be useful to reduce formula size. The problem is also of practical relevance: for bit-blasting SMT solvers, it is important to encode basic operations by small formulas.

Problem instance example:

```
(x1 + x2 + x3 + x4 + x5) * (x1 + x2 + x3 + -x4 + -x5) *
(x1 + x2 + -x3 + x4 + -x5) * (x1 + x2 + -x3 + -x4 + x5) *
(x1 + -x2 + x3 + x4 + -x5) * (x1 + -x2 + x3 + -x4 + x5) *
(x1 + -x2 + -x3 + x4 + x5) * (x1 + -x2 + -x3 + -x4 + -x5) *
(-x1 + x2 + x3 + x4 + -x5) * (-x1 + x2 + x3 + -x4 + x5) *
(-x1 + x2 + -x3 + x4 + x5) * (-x1 + x2 + -x3 + -x4 + -x5) *
(-x1 + -x2 + x3 + x4 + x5) * (-x1 + -x2 + x3 + -x4 + -x5) *
(-x1 + -x2 + -x3 + x4 + -x5) * (-x1 + -x2 + -x3 + -x4 + x5)
```

Correctness property: existential closure of G is equivalent to F, and $|G| < |F|$.

Typical system answers for incorrect submissions:

```
gelesen: (x1 + x2 + x3 + x6) * (-x6 + x1)

nicht äquivalent, z. B. bei Belegung(en)
    listToFM [ ( x1 , True ) , ( x2 , True ) , ( x3 , False )
        , ( x4 , False ) , ( x5 , False ) ]
```

The equivalence check uses a BDD implementation [Wal14b].

Hint for solving the example: invent an extra variable that represents the XOR of some of the x_i.

This problem type is a non-trivial highscore exercise. The given example instance (size 16) has a solution of size 12.

4 Propositional Logic Resolution

- instance: an unsatisfiable formual F in conjunctive normal form,
- solution: a derivation of the empty clause from F by resolution,
- measure: number of derivation steps.

Motivation: the student should see "both sides of the coin": resolution proves unsatisfiability. Also, the student sees that finding resolution proofs is hard, and they are not always short (because if they were, then SAT $\in$ NP $\cap$ coNP, which nobody believes).

Resolution of course has many applications. In the lecture I emphasize certification of UNSAT proof traces in SAT competitions.

Problem instance example: clauses are numbered for later reference

```
0 : ! a || b || c          6 : a || ! b || d
1 : a || b || c            7 : a || ! b || ! d
2 : a || ! c || ! d        8 : ! a || d
3 : ! a || ! c || ! d      9 : ! c || ! d
4 : a || c                 10 : ! b || c || ! d
5 : ! c || d               11 : a || b || ! d
```

Problem solution domain: sequence of resolution steps, example:

```
[ Resolve { left = 4 , right = 8 , literal = a }
, Resolve { left = 12, right = 5, literal = c }
]
```

Correctness property: for each step `s` it holds that `literal s` occurs in clause `left s`, and `! (literal s)` occurs in clause `right s`. If so, then the system computes the resolvent clause and assigns the next number to it. The clause derived in the last step is the empty clause.

Typical system answer for an incomplete submission:

```
nächster Befehl Resolve { left = 4 , right = 8 , literal = a }
    entferne Literal a aus Klausel a || c      ergibt c
    entferne Literal ! a aus Klausel ! a || d  ergibt d
        neue Klausel        12 : c || d
nächster Befehl Resolve { left = 12 , right = 5 , literal = c }
    entferne Literal c aus Klausel c || d      ergibt d
    entferne Literal ! c aus Klausel ! c || d  ergibt d
        neue Klausel        13 : d
letzte abgeleitete Klausel ist Zielklausel? Nein.
```

Instance generator: is controlled by, for example,

```
Config { num_variables = 5 , literals_per_clause_bounds = ( 2 , 3 ) }
```

The implementation uses a BDD implementation [Wal14b] to make sure that the generated formula is unsatisfiable. The generator does not actually produce a proof trace, because it must exist, by refutation completeness.

5 Backtracking and Backjumping: DPLL (with CDCL)

- instance: a CNF F,
- solution: a complete DPLL proof trace determining the satisfiability of F.

Motivation: The DPLL algorithm [DP60,DLL62] is the workhorse of modern SAT solvers, and the basis for extensions in SMT.

This exercise type should help the student to understand the specification of the algorithm, and also the design space that an implementation still has, e.g., picking the next decision variable, or the clause to learn from a conflict, and the backjump target level.

This is an instance of the "non-determinism" design principle for exercises: force the student to make choices, instead of just following a given sequence of steps. It fits nicely with abstract descriptions of algorithms that postpone implementation choices, and instead give a basic invariant first, and prove its correctness.

Problem instance example:

```
[ [ 3 , -4 ] , [ 4 , 5 ] , [ 3 , -4 , 5 ] , [ 1 , -2 ]
, [ 3 , 4 , 5 ] , [ 1 , 2 , 4 , 5 ] , [ -1 , 4 , -5 ]
, [ -1 , -2 ] , [ 2 , 3 , -4 ] , [ -3 , -4 , -5 ] , [ 2 , 3 , -5 ]
, [ -2 , -3 , 4 ] , [ -1 , -4 ] , [ -3 , 4 ] , [ 1 , -3 , -4 , 5 ] ]
```

Problem solution domain: sequence of steps, where

```
data Step = Decide Literal
          | Propagate { use :: Clause, obtain :: Literal }
          | SAT
          | Conflict Clause
          | Backtrack
          | Backjump { to_level :: Int, learn :: Clause }
          | UNSAT
```

Correctness property: the sequence of steps determines a sequence of states (of the tree search), where

```
data State = State { decision_level :: Int
           , assignment :: M.Map Variable Info
           , conflict_clause :: Maybe Clause
           , formula :: CNF
           }
data Info = Info { value :: Bool, level :: Int, reason :: Reason }
data Reason = Decision | Alternate_Decision
            | Propagation { antecedent :: Clause }
```

A state represents a node in the search tree. For each state (computed by the system), the next step (chosen by the student) must be applicable, and the last step must be `SAT` or `UNSAT`.

The `reason` for a variable shows whether its current value was chosen by a `Decision` (then we need to take the `Alternate_Decision` on backtracking) or by `Propagation` (then we need to remember the antecedent clause, for checking that a learned clause is allowed).

A `SAT` step asserts that the current assignment satisfies the formula. A `Conflict c` step asserts that Clause `c` is falsified by the current assignment. The next step must be `Backtrack` (if the decision level is below the root) or `UNSAT` (if the decision level is at the root, showing that the tree was visited completely).

A Step `Propagate {use=c, obtain=l}` is allowed if clause `c` is a unit clause under the current assignment, with `l` as the only un-assigned literal, which is then asserted. A `Decide` step just asserts the literal.

Then following problem appears: if the student can guess a satisfying assignment σ of the input formula, then she can just `Decide` the variables, in sequence, according to σ, and finally claim `SAT`. This defeats the purpose of the exercise.

The following obstacle prevents this: each `Decide` must be negative (assert that the literal is `False`). This forces a certain traversal order. It would then still be possible to "blindly" walk the tree, using only `Decide`, `Conflict` (in the leaves), and `Backtrack`. This would still miss a main point of DPLL: taking shortcuts by propagation. In the exercise, this is enforced by rejecting all solutions that are longer than a given bound.

Instance generator: will produce a random CNF F. By completeness of DPLL, a solution for F (that is, a DPLL-derivation) does exist, so the generator would be done here. This would create problem instances of vastly different complexities (with solutions of vastly different lengths), and this would be unjust to students. Therefore, the generator enumerates a subset S of all solutions of F, and then checks that the minimal length of solutions in S is near to a given target.

The solver is written in a "PROLOG in Haskell" style, using `Control.Monad.Logic` [KcSFS05] with the *fair disjunction* operator to model choices, and allow a breadth-first enumeration (where we get shorter solutions earlier).

There is some danger that clever students extract this DPLL implementation (`autotool` is open-sourced) to solve their problem instance. I think this approach

requires an amount of work that is comparable to solving the instance manually, so I tolerate it.

DPLL with CDCL (conflict driven clause learning): in this version of the exercise, there is no `Backtrack`, only `Backjump { to_level = l, learn = c }`. This step is valid right after a conflict was detected, and if clause `c` is a consequence of the current antecedents that can be checked by *reverse unit propagation*: from `not c` and the antecedents, it must be possible to derive the empty clause by unit propagation alone. This is a "non-deterministic version" of Algorithm 2.2.2 of [KS08].

I introduced another point of non-determinism in clause learning: the student can choose any decision level to backjump to. Textbooks prove that one should go to the second most recent decision level in the conflict clause but that is a matter of efficiency, not correctness, so we leave that choice to the student.

If the `Backjump` does not go high enough, then learning the clause was not useful (it is just a `Backtrack`). If the `Backjump` does go too high (in the extreme, to the root), then this will lead to duplication of work (re-visiting parts of the tree). Note that the target node of the backjump *is* re-visited: we return to a state with a partial assignment that was seen before. But this state contains the learned clause, so the student should use it in the very next step for unit propagation, and only that avoids to re-visit subtrees.

A challenge problem: the following pigeonhole formula is unsatisfiable for $n > m$, but this is hard to see for the DPLL algorithm: "there are n pigeons and m holes, each pigeon sits in a hole, and each hole has at most one pigeon" [Cla11]. I posed this problem for $n = 5, m = 4$. The resulting CNF on 20 variables ($v_{p,h}$: pigeon p sits in hole h) has 5 clauses with 4 literals, and 40 clauses with 2 literals. My students obtained a DPLL solution with 327 steps, and DPLL-with-CDCL solution with 266 steps. (Using software, I presume.)

6 Evaluation in Finite Algebras

- instance: a signature Σ, two Σ-algebras A, B, both with finite universe U,
- solution: a term t over Σ with $t_A \neq t_B$.

Motivation: the introduction, or recapitulation, of predicate logic basics. The exercise emphasizes the difference and interplay between syntax (the signature, the term) and semantics (the algebras).

This exercise type shows the design principle of inversion: since we usually define syntax first (terms, formulas), and semantics later (algebras, relational structures), it looks natural to ask "find an algebra with given property". Indeed I have such an exercise type ("find a model for a formula"), but here I want the other direction.

Problem instance example:

```
Finden Sie einen Term zur Signatur
    Signatur
        { funktionen = listToFM [ ( p , 2 ) , ( z , 0 ) ]
        , relationen = listToFM [ ]
        , freie_variablen = mkSet [ ]
        }
, der in der Struktur
    A = Struktur
            { universum = mkSet [ 1 , 2 , 3 ]
            , predicates = listToFM [ ]
            , functions = listToFM
                    [ ( p
                      , {(1 , 1 , 3) , (1 , 2 , 3) , (1 , 3 , 3) , (2 , 1 , 2) ,
                         (2 , 2 , 1) , (2 , 3 , 1) , (3 , 1 , 3) , (3 , 2 , 1) ,
                         (3 , 3 , 2)}
                      )
                    , ( z , {(3)} )
                    ]
            }
eine anderen Wert hat
als in der Struktur
    B = Struktur
            { universum = mkSet [ 1 , 2 , 3 ]
            , predicates = listToFM [ ]
            , functions = listToFM
                    [ ( p
                      , {(1 , 1 , 1) , (1 , 2 , 3) , (1 , 3 , 3) , (2 , 1 , 2) ,
                         (2 , 2 , 1) , (2 , 3 , 1) , (3 , 1 , 3) , (3 , 2 , 1) ,
                         (3 , 3 , 2)}
                      )
                    , ( z , {(3)} )
                    ]
            }
```

here, k-ary functions are given as sets of $(k+1)$-tuples, e.g., $(2,2,1) \in p$ means that $p(2,2) = 1$.

Problem solution domain: terms t over the signature, e.g.,

```
p (p (p (p (z () , z ()) , z ()) , z ()) , z ())
```

Correctness property: value of term t in A is different from value of t in B.

Example solution: the student first notes that the only difference is at $p_A(1,1) = 3 \neq 1 = p_B(1,1)$, so the solution can be $p(s,s)$ where $s_A = 1 = s_B$. Since $z_A() = 3, p_A(3,3) = 2, p_A(3,2) = 1$, a solution is

```
p (p (p (z(),z()),z()),p (p (z(),z()),z()))
```

Instance generator: is configured by the signature, and the size of the universe. It will build a random structure A, and apply a random mutation, to obtain B. It also checks that the point of mutation is reachable by ground terms, and none of them are too small.

7 Satisfiability modulo Theories: DPLL(T)

- instance: a conjunction F of clauses, where a clause is a disjunction of literals, and a literal is a Boolean literal or a theory literal
- solution: a DPLL(T) proof trace determining the satisfiability of F.

Motivation: Satisfiability modulo Theories (SMT) considers arbitrary Boolean combinations of atoms taken from a theory T, e.g., the theory of linear inequalities over the reals. DPLL(T) is a decision procedure for SMT that combines the DPLL algorithm with a "theory solver" that handles satisfiability of conjunctions of theory literals [NOT06].

E.g., the Fourier-Motzkin algorithm (FM) for variable elimination is a theory solver for linear inequalities. It is not efficient, but I like it for teaching: it has a nice relation to propositional resolution, and it is practically relevant as a pre-processing step in SAT solvers [EB05]. Also, some students took the linear optimization course, some did not, so I do not attempt to teach the simplex method.

We treated DPLL in Section 5, and give only the differences here.

Problem instance example:

```
[ [  p ,  q ] , [ ! p , ! 0 <= +  x ]
, [ ! q ,  0 <= + 2 -1 * x ] , [  0 <= -3 +  x ] ]
```

this represents a set of clauses, where `! p` is a (negative) Boolean literal, and `0 <= -3 + x` is a (positive) theory literal.

Problem solution domain: sequence of steps, where

```
data Step = Decide Literal
          | Propagate { use :: Conflict , obtain :: Literal }
          | SAT
          | Conflict Conflict
          | Backtrack
          | Backjump { to_level :: Int, learn :: Clause }
          | UNSAT
data Conflict = Boolean Clause | Theory
```

Note that this `Step` type results from that of Section 5 by replacing `Clause` with `Conflict` in two places (arguments to `Propagate` and `Conflict`).

Correctness property: The sequence of steps determines a sequence of states (of the tree search). As long as we use only the `Boolean Clause :: Conflict` constructor, we have a DPLL computation — that may use theory atoms, but only combines them in a Boolean way. The underlying theory solver is only used in the following extra cases:

A `Conflict Theory` step is valid if the conjunction of the theory literals in the current assignment is unsatisfiable in the theory. E.g., `! 0 <= x` and `0 <= -3 + x` is not a Boolean conflict, but a theory conflict.

A `Propagate { use = Theory, obtain = l }` step is valid if `l` is a theory literal that is implied by the theory literals in the current assignment, in other words, if `! l` together with these literals is unsatisfiable in the theory.

Example solution:

```
[ Propagate {use = Boolean [  0 <= -3 +  x ], obtain =  0 <= -3 +  x  }
, Propagate {use = Theory, obtain =  0 <= +  x }
, Propagate {use = Boolean [ ! p , ! 0 <= +  x ], obtain = ! p }
, Propagate {use = Boolean [ p , q ], obtain = q}
, Propagate {use = Boolean [ ! q ,  0 <= + 2 -1 * x ], obtain =  0 <= + 2 -1 * x }
, Conflict Theory
, UNSAT ]
```

E.g., to validate the second step (theory propagation), the T-solver checks that the conjunction of `(0 <= -3+x)` (from the current assignment) and `!(0 <= x)` (negated consequence) is unsatisfiable. We arrive at a T-conflict at the root decision level, so the input formula is unsatisfiable.

8 Solving Finite Domain Constraints

- instance: a relational Σ-structure R over finite universe U, and a conjunction F of Σ-atoms
- solution: a complete FD tree search trace determining the satisfiability of F.

Motivation: Finite Domain (FD) constraints can be seen as a mild generalization of propositional SAT. Methods for solution are similar (tree search), but have differences. In particular, I use FD constraints to discuss (arc) consistency notions, as in [Apt03], and this automated exercise type also makes that point.

The design principle is again non-determinism: the student has to make a choice among several possible steps. In particular, propagation and conflict detection are done via *arc consistency deduction*.

Problem instance example:

```
Give a complete computation of an FD solver
that determines satisfiability of:
    [ P ( x , y , z ) , P ( x , x , y ) , G ( y , x ) ]
in the structure:
```

```
Algebra
    { universe = [ 0 , 1 , 2 , 3 ]
    , relations = listToFM
          [ ( G , mkSet
                 [ [ 1 , 0 ] , [ 2 , 0 ] , [ 2 , 1 ]
                 , [ 3 , 0 ] , [ 3 , 1 ] , [ 3 , 2 ] ] )
          , ( P , mkSet
                 [ [ 0 , 0 , 0 ] , [ 0 , 1 , 1 ] , [ 0 , 2 , 2 ]
                 , [ 0 , 3 , 3 ] , [ 1 , 0 , 1 ] , [ 1 , 1 , 2 ]
                 , [ 1 , 2 , 3 ] , [ 2 , 0 , 2 ] , [ 2 , 1 , 3 ]
                 , [ 3 , 0 , 3 ] ] ) ]
    }
```

Problem solution domain: sequence of steps, where (`u` is the universe)

```
data Step u = Decide Var u
     | Arc_Consistency_Deduction
         { atoms :: [ Atom ], variable :: Var, restrict_to :: [ u ]   }
     | Solved
     | Backtrack
     | Inconsistent
```

Correctness property: the sequence of steps determines a sequence of states (of the tree search) where a state is a `Stack` containing a list of domain assignments (for each variable, a list of possible values)

```
data State u = Stack [ M.Map Var [u] ]
```

A state is `Solved` if each instantiation of the current assignment (at the top of the stack) satisfies the formula. A state is *conflicting* if the current assignment contains a variable with empty domain. In a conflicting state, we can do `Backtrack` (pop the stack) or claim `Inconsistent` (if the stack has one element only).

A step `Decide v e` pops an assignment `a` off the stack, and pushes two assignments back: one where the domain of `v` is the domain of `v` in `a`, without `e` (that is where we have to continue when backtracking), and the other where the domain of `v` is the singleton `[e]`

A step `Arc_Consistency_Deduction { atoms, var, restrict }` is valid if the following holds:

- `atoms` is a subset of the formula
- for each assignment from `var` to `current-domain var` without `restrict`: it cannot be extended to an assignment that satisfies `atoms`.

This constitutes a proof that the domain of `v` can be restricted to `restrict`. We have non-determinism here, as we are not enforcing that the restricted set is minimal. If the restricted set is empty, we have detected a conflict. Since we want a minimal design, there is no other `Step` constructor for stating conflicts.

There are several arc consistency concepts in the literature. Ours has these properties:

– we allow to consider a set of atoms (its conjunction), but we can restrict its size (to one, then we are considering each atom in isolation)
– we can restrict the number of variables that occur in the set of atoms. this number is the size of the hyper-edges that are considered for hyperarc-consistency. For 1, we get node consistency; for 2, standard arc consistency.
– from this number, we omit those variables that are uniquely assigned in the current state. This allows to handle atoms of any arity: we just have to `Decide` enough of their arguments (so their domain is unit), and can apply arc consistency deduction on those remaining.

Example solution: starts like this:

```
[ Decide    x 0
, Arc_Consistency_Deduction
      { atoms = [ P ( x , x , y ) , G ( y , x ) ]
      , variable = y , restrict_to = [ ]
      }
, Backtrack
, Decide    x 1
, Arc_Consistency_Deduction
      { atoms = [ P ( x , x , y ) , G ( y , x ) ]
      , variable = y , restrict_to = [ 2 ]
      }
]
```

After the first step (`Decide x 0`), the state is

```
Stack [ listToFM [ ( x , [ 0 ] )
                 , ( y , [ 0 , 1 , 2 , 3 ] )
                 , ( z , [ 0 , 1 , 2 , 3 ] ) ]
      , listToFM [ ( x , [ 1 , 2 , 3 ] )
                 , ( y , [ 0 , 1 , 2 , 3 ] )
                 , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
```

Typical system answers for incorrect submissions: hyperarc size restriction is violated:

```
current
    Stack
        [ listToFM
              [ ( x , [ 0 , 1 , 2 , 3 ] )
              , ( y , [ 0 , 1 , 2 , 3 ] )
              , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
step
    Arc_Consistency_Deduction
        { atoms = [ P ( x , x , y ) , G ( y , x ) ]
        , variable = y , restrict_to = [ 2 ]  }
```

```
these atoms contain 2 variables with non-unit domain:
    mkSet    [ x , y ]
but deduction is only allowed for hyper-edges of size up to 1
```

elements are incorrectly excluded from domain:

```
current
    Stack [ listToFM [ ( x , [ 0 ] )
              , ( y , [ 0 , 1 , 2 , 3 ] )
              , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
step
    Arc_Consistency_Deduction
        { atoms = [ P ( x , x , y ) ]
        , variable = y , restrict_to = [ 1 ]
        }
these elements cannot be excluded from the domain of the variable,
because the given assignment is a model for the atoms:
    [ ( 0 , listToFM [ ( x , 0 ) , ( y , 0 ) ] ) ]
```

Instance generator: uses the same idea as for DPLL: generate a random instance, solve it breadth-first, and check for reasonable solution length.

9 Related Work and Conclusion

We have shown automated exercises for constraint programming, and also presented the intentions behind their design. In particular, we described how to test the student's understanding of constraint solving algorithms by making use of non-determinism, similar in spirit to the inference systems (proof rules) in [Apt03]. These exercise types are part of the `autotool` framework for generating exercise problem instances, and grading solutions semantically.

There are several online courses for constraint programming, Few of them seem to contain online exercises. In all cases, computerized exercises (offline or online) focus on teaching a specific constraint language, as a means of modelling, e.g., Gnu-Prolog [Sol04], ECLIPSe [Sim09], CHR [Kae07].

The exercises from the present paper do not focus much on modelling, and learning a specific language. The aim is to teach the semantics of logical formulas, and fundamental algorithms employed by constraint solvers. One could say that each exercise uses a different problem-specific language. Each exercise is graded automatically, and immediately, while giving feedback that helps the student.

So, the approaches are not competing, but complementary.

Acknowledgments: Many thanks to the students of my Constraint Programming course (Sommersemester 2014) for working on these exercises, fighting for high-scores, and reporting bugs in the software and in a draft of this report; and to Alexander Bau, Carsten Fuhs, and Sibylle Schwarz for helpful comments on exercise design and implementation.

References

Apt03. Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

Cla11. Edmund M. Clarke. Assignment 2. `http://www.cs.cmu.edu/~emc/15414-f11/assignments/hw2.pdf`, 2011.

DLL62. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

DP60. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

EB05. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

GLSW11. Hans-Gert Gräbe, Frank Loebe, Sibylle Schwarz, and Johannes Waldmann. autotool und autotool-Netzwerk. `http://www.imn.htwk-leipzig.de/~waldmann/talk/11/hds/`, 2011. HDS-Jahrestagung, TU Dresden, November 4.

Kae07. Martin Kaeser. WebCHR examples. `http://chr.informatik.uni-ulm.de/~webchr/`, 2007.

KcSFS05. Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 192–203. ACM, 2005.

KS08. Daniel Kroening and Ofer Strichman. *Decision Procedures, an Algorithmic Point of View*. Springer, 2008.

NOT06. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

RRW08. Mirko Rahn, Alf Richter, and Johannes Waldmann. The Leipzig autotool E-Learning/E-Testing System. `http://www.imn.htwk-leipzig.de/~waldmann/talk/08/ou08/`, 2008. Symposium on Math Tutoring, Tools and Feedback, Open Universiteit Nederland, September 19.

SDH02. Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT 2002 Competition (preliminary draft). `http://www.satcompetition.org/2002/onlinereport.pdf`, 2002.

Sim09. Helmut Simonis. Lessons learned from developing an on-line constraint programming course. `http://4c.ucc.ie/~hsimonis/lessons_abstract.pdf`, 2009. 14th Workshop on Constraint Solving and Constraint Logic Programming CSCLP 2009, Barcelona.

Sol04. Christine Solnon. An on-line course on constraint programming. `http://www.ep.liu.se/ecp/012/001/ecp012001.pdf`, 2004. First International Workshop on Teaching Logic Programming TeachLP 2004.

Tse70. G. S. Tseitin. On the complexity of derivation in propositional calculus. Leningrad Seminar on Mathematical Logic, 1970.

Wal14a. Johannes Waldmann. Skript Constraint-Programmierung. `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss14/cp/folien/`, 2014. lecture slides (in German).

Wal14b. Johannes Waldmann. The Haskell OBDD Package. `https://hackage.haskell.org/package/obdd`, 2014.