Don't Be Puzzled!

Martin Henz
Programming Systems Lab
University of the Saarland
Geb. 45, Postfach 151150
D-66041 Saarbrücken, Germany
email: `henz@dfki.uni-sb.de`

July 3, 1996

Abstract

This paper is about how to solve a class of puzzles, called self-referential quizzes (`srq`), with constraint programming. An `srq` is a sequence of multiple choice questions that are about the puzzle itself. `srq`s are an attractive pastime, when they provide the possibility of drawing non-trivial conclusions on the way to the solution.

We introduce a typical `srq`, and represent it as a propositional satisfiability problem. Its straightforward clausal representation is too big for effective treatment using standard methods. Instead, we solve it with finite domain constraint programming. For this application of constraint programming, support of logic connectives such as conjunction and disjunction is crucial. With their small problem descriptions, `srq`s are ideal candidates for benchmarks covering the implementation of 0/1 variables in constraint programming languages.

1 Introduction

Around 1993, Jim Propp invented a quiz that he called "Self-referential Aptitude Test", or SRAT for short. It contains 20 quiz questions with 5 alternatives each, almost all of which refer to the quiz itself. The problem is to find unique answers for all the questions, such that they are all correct. A typical example for a question is Question 2:

2. The only two consecutive questions with identical answers are questions
   (A) 6 and 7 (B) 7 and 8 (C) 8 and 9 (D) 9 and 10 (E) 10 and 11

Distributed over the internet, this quiz attracted much attention among puzzle solvers. It is an attractive quiz, because it allows the solver to draw many non-trivial conclusions and thus reduce the search space tremendously. An experienced puzzle solver can solve SRAT within half an hour of puzzling fun.

To our knowledge, we were the first to solve SRAT automatically. We used the constraint programming language Oz to find the solution and prove it to be unique. It is surprizing that the Oz program solves SRAT almost deterministically. The constraints in SRAT are so strong that only *one single* choice-point suffices to find a solution and prove it to be unique. This nicely reflects the design goal of the author that the solution could be found "without too much trial-and-error" [6].

In this paper, we use the Self-Referential Quiz, short SRQ, instead of SRAT. We invented SRQ on the base of SRAT for the purpose of this paper. Like SRAT, SRQ has a unique solution and contains a variety of interesting self-referential questions. However, SRQ is more suitable for this presentation, since it is composed of only 10 questions, Assuming that exactly one alternative is true for every question, there is a unique solution to the following quiz. What is the solution?

1. The first question whose answer is A is question
   (A) 4 (B) 3 (C) 2 (D) 1 (E) none of the above
2. Identical answers have questions
   (A) 3 and 4 (B) 4 and 5 (C) 5 and 6 (D) 6 and 7 (E) 7 and 8
3. The next question with answer A is question (A) 4 (B) 5 (C) 6 (D) 7 (E) 8
4. The first even numbered question with answer B is question
   (A) 2 (B) 4 (C) 6 (D) 8 (E) 10
5. The only odd numbered question with answer C is question
   (A) 1 (B) 3 (C) 5 (D) 7 (E) 9
6. A question with answer D (A) comes before this one, but not after (B) comes after this one, but not before (C) comes before and after this one (D) does not occur at all (E) none of the above
7. The last question whose answer is E is question (A) 5 (B) 6 (C) 7 (D) 8 (E) 9
8. The number of questions whose answers are consonants is
   (A) 7 (B) 6 (C) 5 (D) 4 (E) 3
9. The number of questions whose answers are vowels is
   (A) 0 (B) 1 (C) 2 (D) 3 (E) 4
10. The answer to this question is (A) A (B) B (C) C (D) D (E) E

Figure 1: The Puzzle SRQ

making the presentation shorter, and since it is a harder problem than SRAT in a sense that it takes more choice-points to solve it, if the same solution technique is applied. Collected information on `srq`s can be found in [4].

2 The Self-Referential Quiz

Figure 1 states the puzzle SRQ. A few remarks may be helpful:

1. Several alternatives can be excluded immediately. For example, the answer to question 1 cannot be A, which states that question 4 is the first question whose answer is A, because then, question 1 would be the first question whose answer is A, which is a contradiction. Other such examples are the alternatives 7 C and 9 A.

2. The questions are heavily intertwined. For example, if the answer to question 1 is C, then it follows from the first five questions only that the answer to question 2 must be A, the answer to question 3 must be B, the answer to question 4 must be B, and the answer to question 5 must be A. The argument is left to the reader.

3. The reader may verify that the following is a solution:

   1:C, 2:A, 3:B, 4:B, 5:A, 6:B, 7:E, 8:B, 9:E, 10:D

3 SRQ as a Satisfiability Problem

Every alternative of every question of SRQ contains a statement, which is either true or false. Thus, we are going to express SRQ as a formula f of propositional calculus, using

$$
\begin{array}{ll}
A_1 \equiv A_4 \wedge \neg A_1 \wedge \neg A_2 \wedge \neg A_3, & A_6 \equiv (D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5) \wedge \\
B_1 \equiv A_3 \wedge \neg A_1 \wedge \neg A_2, & \quad \neg(D_7 \vee D_8 \vee D_9 \vee D_10), \\
C_1 \equiv A_2 \wedge \neg A_1, & B_6 \equiv \neg(D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5) \wedge \\
D_1 \equiv A_1, & \quad (D_7 \vee D_8 \vee D_9 \vee D_10), \\
E_1 \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge \neg A_4, & C_6 \equiv (D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5) \wedge \\
A_2 \equiv 3 \cong 4,\ B_2 \equiv 4 \cong 5,\ C_2 \equiv 5 \cong 6, & \quad (D_7 \vee D_8 \vee D_9 \vee D_10), \\
D_2 \equiv 6 \cong 7,\ E_2 \equiv 7 \cong 8, & D_6 \equiv \sum_{i \in \{1 \ldots 10\}} D_i = 0, \\
 & E_6 \equiv D_6, \\
A_3 \equiv A_4, & A_7 \equiv E_5 \wedge \neg E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}, \\
B_3 \equiv A_5 \wedge \neg A_4, & B_7 \equiv E_6 \wedge \neg E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}, \\
C_3 \equiv A_6 \wedge \neg A_4 \wedge \neg A_5, & C_7 \equiv E_7 \wedge \neg E_8 \wedge \neg E_9 \wedge \neg E_{10}, \\
D_3 \equiv A_7 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6, & D_7 \equiv E_8 \wedge \neg E_9 \wedge \neg E_{10}, \\
E_3 \equiv A_8 \wedge \neg A_4 \wedge \neg A_5 \wedge \neg A_6 \wedge \neg A_7, & E_7 \equiv E_9 \wedge \neg E_{10}, \\
A_4 \equiv B_2, & A_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 7, \\
B_4 \equiv B_4 \wedge \neg B_2, & B_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 6, \\
C_4 \equiv B_6 \wedge \neg B_2 \wedge \neg B_4, & C_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 5, \\
D_4 \equiv B_8 \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6, & D_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 4, \\
E_4 \equiv B_{10} \wedge \neg B_2 \wedge \neg B_4 \wedge \neg B_6 \wedge \neg B_8, & E_8 \equiv \sum_{i \in \{1 \ldots 10\}} B_i + C_i + D_i = 3, \\
A_5 \equiv C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9, & A_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 0, \\
B_5 \equiv \neg C_1 \wedge C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge \neg C_9, & B_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 1, \\
C_5 \equiv \neg C_1 \wedge \neg C_3 \wedge C_5 \wedge \neg C_7 \wedge \neg C_9, & C_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 2, \\
D_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge C_7 \wedge \neg C_9, & D_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 3, \\
E_5 \equiv \neg C_1 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_7 \wedge C_9, & E_9 \equiv \sum_{i \in \{1 \ldots 10\}} A_i + E_i = 4
\end{array}
$$

Figure 2: SRQ as Satisfiability Problem

the logical connectives $\wedge$ for conjunction, $\vee$ for disjunction, $\neg$ for negation and $\equiv$ for equivalence. A solution to SRQ is then an assigment of the symbols occurring in f to truth values such that f is satisfied.

We introduce 50 truth symbols l_i stating that the answer to question i is l, for $i \in \{1 \ldots 10\}$ and $l \in \{A, B, C, D, E\}$. For example, if B_4 is true, then the answer to question 4 is B.

The first sentence in SRQ states that exactly one alternative must be true for every question. Thus, for every $j \in \{1 \ldots 10\}$, the formula

$$
\begin{array}{l}
A_j \wedge \neg B_j \wedge \neg C_j \wedge \neg D_j \wedge \neg E_j \ \vee \ \neg A_j \wedge B_j \wedge \neg C_j \wedge \neg D_j \wedge \neg E_j \ \vee \\
\neg A_j \wedge \neg B_j \wedge C_j \wedge \neg D_j \wedge \neg E_j \ \vee \ \neg A_j \wedge \neg B_j \wedge \neg C_j \wedge D_j \wedge \neg E_j \ \vee \\
\neg A_j \wedge \neg B_j \wedge \neg C_j \wedge \neg D_j \wedge E_j
\end{array} \tag{1}
$$

must hold. As an abbreviation for this formula, we introduce

$$A_j + B_j + C_j + D_j + E_j = 1 \tag{2}$$

We allow the generalization to any integer instead of 1, and the usual summation notation $\sum$. As another convenient abbreviation, let us introduce:

$$i \cong j \ ::= \ (A_i \equiv A_j) \wedge (B_i \equiv B_j) \wedge (C_i \equiv C_j) \wedge (D_i \equiv D_j) \wedge (E_i \equiv E_j) \tag{3}$$

Using this notation, Figure 2 gives the translation of the further information provided by Questions 1 through 9. Note that Question 10 provides no further information.

The problem can now be stated as follows: Find an assignment of the variables

$A_i, B_i, C_i, D_i, E_i,\ i \in \{1 \dots 10\}$, to truth values such that the conjunction of all formulae (1) for $j \in \{1 \dots 10\}$ and all formulae in Figure 2 holds. The general problem of satisfiability of formulae in propositional calculus (SAT) is the first problem, for which NP-completeness has been shown [1].

The representation of f in clausal form as required by the standard algorithms for satisfiability [3, 2] becomes so big that it is questionable, whether these algorithms can be used here: A clausal form of the formula associated with alternative 8 A (see Figure 2) alone contains at least 2.035.800 clauses. Thus, we did not attempt to run a conventional algorithm on our formula f. Instead, we translate f to a constraint program, making use of the fact that some constraint languages provide for propagation of arbitrary combinations of conjunction, disjunction, equivalence, and summation.

4 SRQ as a Constraint Program

We know that humans can solve SRQ; after distributing SRQ on the internet, several people reported the correct answer and a solving time of 10 minutes to half an hour. A human problem solver usually employs the following strategy:

Propagation: Try to exclude as many alternatives as possible.

Distribution: Choose one of the still possible alternatives and assume it to be true, and continue with Propagation.

When a contradiction is encountered, jump to a branch of the search tree, where there are still alternatives.

For example, a human problem solver can (among others) immediately exclude the alternative A for question 1 (see Remark 1 on page 2). Therefore, let us assume she picks alternative B for question 1. Now, 1 B states 3 A, which states 4 A, which implies 2 A. This is a contradiction, since question 2 states that there are only two consecutive questions with identical answers and we have 1 A, 2 A, and 3 A.

Thus she tries alternative C for question 1, leading to the conclusions given in Remark 2 on page 2.

Constraint programming turns exactly this principle of "propagation and distribution" into a programming paradigm.

An obvious approach to translate SRQ into a constraint program is to model every truth variable l_i from the previous section as a finite domain variable ranging over values from $\{0, 1\}$, where 0 represents "false" and 1 represents "true". Such variables, we call *0/1 variables*. In the following, we are going to use the constraint language Oz as implementation language [8, 9]. In the following Oz program, we are defining a list `Answers` of five tuples `A`, `B`, `C`, `D`, and `E`, each containing ten 0/1 variables:

```
declare Answers=[A B C D E]
{ForAll Answers fun {$} {FD.tuple q 10 0#1} end}
```

The variable B_4 in the previous section corresponds to `B.4`. Note the dot syntax for feature access and `{...}` for procedure application. In the rest of this paper, we are going to discuss the implementation of a problem solver for SRQ given in Figure 3. In comments `%%` we indicate the section that explains the respective part of the program. We refer to lines in the program by using marks of the form `%1C`, implementing the corresponding alternative, here alternative C of Question 1.

For each formula in f, we are going to install a number of *propagators*, whose declarative semantics is given by the formula. The propagators concurrently inspect a *constraint store*, which in our case contains only basic constraints of the form `X = 0`, `X = 1`, `X` $\in \{0, 1\}$, and `X = Y`. During search, these propagators may amplify the store by adding basic constraints to it, which possibly triggers further propagation.

Consider for example line *%1C* in Figure 3

```
C.1 = A.2 * ~A.1
```

In slight abuse of notation, we redefined the arithmetic operators `*`, `+`, and `~` in the third line in Figure 3 to propagators `FD.conj` (conjunction), `FD.disj` (disjunction), and `FD.nega` (negation). Thus, the above line stands for

```
local N in
   {FD.nega A.1 N}
   {FD.conj A.2 N C.1}
end
```

The propagator `{FD.nega A.1 N}` is straightforward: It waits until one of its arguments becomes determined (bound to 0 or 1) and binds the other one to the other value (1 or 0).

The propagator `{FD.conj A.2 N C.1}` can propagate in several ways (see [5]):

1. If `A.2` or `N` is determined to 0, then `C.1` is determined to 0,
2. If `A.2` and `N` are determined to 1, then `C.1` is determined to 1,
3. if `A.2` is determined to 1, then the constraint `N = C.1` is added, and vice versa for `N`,
4. if `C.1` is determined to 1, then both `A.2` and `N` are determined to `1`, and
5. if `C.1` is determined to 0 and `A.2` determined to 1 then `N` is determined to 0, and vice versa for `N`.

The other formulae in Figure 2 are implemented in a similar way. Note that Question 2 uses the equivalence propagator `=:`. Among other propagations, the propagator `B = A.I=:A.J` imposes the constraint `A.I=A.J`, if `B` becomes bound to 1. Question 6 makes use of the disjunction propagator `FD.disj`, denoted by `+`. Questions 6, 8 and 9 make use of the procedure `Sm`, which computes the sum over all finite domain variables in a given tuple or list, using the propagator `FD.sum`. Thus, `SmBCD` is the number of questions whose answers are consonants.

Note that adding `A.1=1` will immediately result in a failure, since the conjunction propagators in *%1A* will trigger `~A.1 = 1`, which will try to bind `A.1` to 0, contradicting its previous binding to 1. Compare this result with Remark 1 on page 2.

5 The General Condition

The general condition (1) on page 3 has the potential for strong constraint propagation: If one p_i is 1, it allows to set all values l_i with $l \neq p$ to 0.

As suggested by the notation (2) on page 3, we represent the general condition by arithmetic propagators of the form

```
{FD.sum [A.i B.i C.i D.i E.i] ´=:´ 1}
```

As soon as one of the five variables becomes bound to 1, this propagator will bind all others to 0. The predefined procedure `For` allows to iterate from 1 to 10. Thus, the program

```
{For 1 10 1
 proc {$ I} {FD.sum [A.I B.I C.I D.I E.I] ´=:´ 1} end}
```

installs a propagator of the above form for each of the ten questions.

```
proc {SRQ Answers}
  !Answers=[A B C D E]
  `*`=FD.conj `+`=FD.disj `~`=FD.nega
  {ForAll Answers
   fun {$} {FD.tuple q 10 0#1} end}
  % General Condition (Section 5)
  {For 1 10 1
   proc {$ I}
      {FD.sum [A.I B.I C.I D.I E.I]
       ´=:´ 1}
   end}
  % Question 1 (Section 4)
  A.1=A.4*~A.1*~A.2*~A.3  %1A
  B.1=A.3*~A.1*~A.2
  C.1=A.2*~A.1            %1C
  D.1=A.1
  E.1=~A.1*~A.2*~A.3*~A.4
  %% Question 2 (Section 4)
  fun {Eq I J} (A.I=:A.J)*(B.I=:B.J)
   *(C.I=:C.J)*(D.I=:D.J)*(E.I=:E.J)
  end
  A.2 = {Eq 3 4}          %2A
  B.2 = {Eq 4 5}
  C.2 = {Eq 5 6}
  D.2 = {Eq 6 7}
  E.2 = {Eq 7 8}
  %% Question 3
  A.3=A.4
  B.3=A.5*~A.4            %3B
  C.3=A.6*~A.4*~A.5
  D.3=A.7*~A.4*~A.5*~A.6
  E.3=A.8*~A.4*~A.5*~A.6*~A.7
  %% Question 4
  A.4=B.2                 %4A
  B.4=B.4 *~B.2
  C.4=B.6 *~B.2*~B.4
  D.4=B.8 *~B.2*~B.4*~B.6
  E.4=B.10*~B.2*~B.4*~B.6*~B.8
  %% Question 5
  A.5=C.1                 %5A
  B.5=C.3 C.5=C.5 D.5=C.7 E.5=C.9
  %% Question 6 (Section 4)
  Bef=D.1+D.2+D.3+D.4+D.5
  Aft=D.7+D.8+D.9+D.10
  A.6=Bef*~Aft
  B.6=Aft*~Bef
  C.6=Bef*Aft
  proc {Sm L S}
     {FD.decl S} {FD.sum L ´=:´ S}
  end
  D.6={Sm D} =: 0
  E.6=D.6
  %% Question 7
  A.7=E.5*~E.6*~E.7*~E.8*~E.9*~E.10
  B.7=E.6*~E.7*~E.8*~E.9*~E.10
  C.7=E.7*~E.8*~E.9*~E.10
  D.7=E.8*~E.9*~E.10
  E.7=E.9*~E.10
  %% Question 8 (Section 4)
  SmBCD={Sm [{Sm B} {Sm C} {Sm D}]}
  A.8 = SmBCD=:7  B.8 = SmBCD=:6
  C.8 = SmBCD=:5  D.8 = SmBCD=:4
  E.8 = SmBCD=:3
  %% Question 9 (Section 4)
  SmAE={Sm [{Sm A} {Sm E}]}
  A.9 = SmAE=:0  B.9 = SmAE=:1
  C.9 = SmAE=:2  D.9 = SmAE=:3
  E.9 = SmAE=:4
in
  {FD.distribute %% (Section 7)
   generic(value:max
           order:constraints)
   {FoldR
    {Map Answers Record.toList}
    Append nil}}
end
```

Figure 3: SRQ as a Constraint Program

6 Don't Be Puzzled!

With this machinery in place, we will show in this section, why determining the answer to question 1 to C leads to a propagation strong enough to yield the conclusions in Remark 2 on page 2.

What happens if we assume the answer to question 1 to be C by imposing the constraint `A.1 = 3`? The constraint `%5A` binds `A.5` to 1. The constraint `A.1 = 3` triggers the propagator `%1C`, resulting in propagation 4 of `FD.conj` (see Section 4), which binds `A.2` to 1. Propagation of the general condition (see Section 5) binds `B.2` to 0, which also binds `A.4` to 0, due to `%4A`. Since `A.2 = 1`, the conjunction propagators in `Eq` for `%2A` result in imposing the constraints `A.3=A.4`, and `B.3=B.4`, among others. Since `A.4=0`, `A.3` is also bound to 0. Since `A.5=1` and `A.4=0`, the conjunction propagator in `%3B` fires by propagation 2 in Section 4 and binds `B.3` to 1. Since `B.3=B.4`, `B.4` is also bound to 1.

Thus, we have shown how only assuming the answer to question 1 to be C results in the conclusions given in 2, using only simple propagators for negation, conjunction, equivalence, and summation. The argument also nicely demonstrates the necessity of concurrency for propagators: It is impossible to determine statically the order in which they do their job.

7 The Solver for SRQ

The program in Figure 3 implements a *solver*, which is a unary procedure, here `SRQ`, abstracting from a *root variable*, here `Answers`. In the solver, constraints on this root variable are established. In our case, `Answers` is bound to the list `[A B C D E]`.

The solver also determines a *distribution strategy*. The line

```
{FD.distribute generic(value:max order:constraints) ...}
```

indicates that we chose the following predefined distribution strategy: If no more propagation is possible, choose the variable on which most propagators depend, and try its maximal value first, which is always 1 in our example. This heuristic roughly corresponds to the most-constrained-first heuristic used in [2]. We choose first 1 instead of 0, since the general condition can propagate much better in this case.

The solver can be given to an *inference engine*, which performs the search. A particular inference engine is the Explorer tool [7]. It allows to visualize the search tree, comprized of choice nodes (circles), failure nodes (squares) and solution nodes (diamonds). The left tree in Figure 4 shows the search tree of the program in Figure 3 for all-solution search. It contains 7 choice nodes; its exploration took 117 milliseconds cpu-time on a Sparc 4 Workstation.

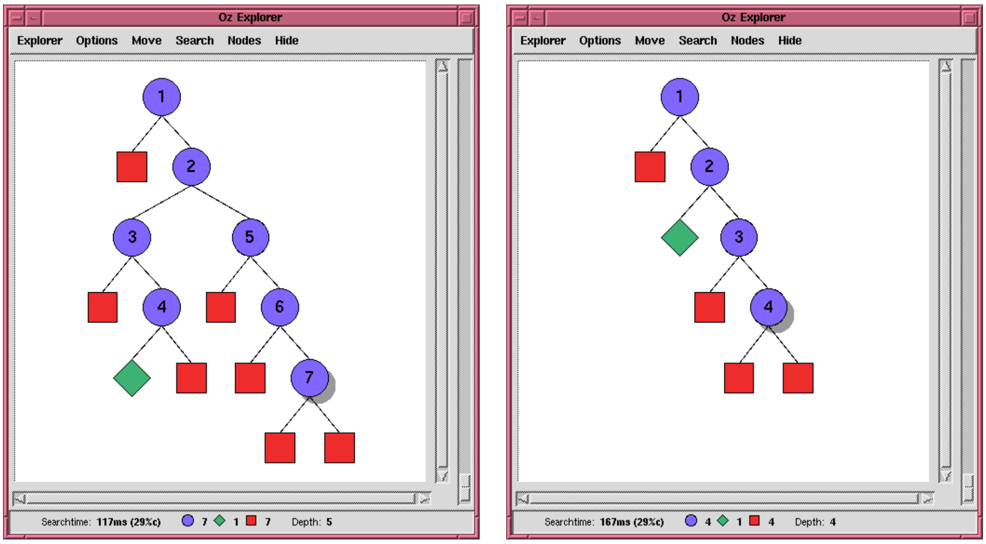

Figure 4: The Oz Explorer

An important programming technique in constraint programming is to allow for further pruning in the search with *redundant constraints*. An obvious redundant constraint is `SmAE`+`SmBCD` $= 10$. The right tree in Figure 4 shows the tree after inserting the propagator `{FD.sum [SmAE SmBCD] '=:' 10}`, which only contains 4 choice nodes.

Using the Explorer, it is straightforward to invent new puzzles of this kind, judge their difficulty by analysing the search tree and verify that they have unique solutions. In fact, we invented a sequence of three puzzles with increasing difficulty (as measured by the size of the search tree), each having 10 questions and a unique solution [4]. We propose to create a test-suite of `srq`s so that we can compare the propagation involving 0/1 variables provided by different implementations of constraint languages.

8 Conclusion

We showed how a formulation of `srq` as a satisfiability problem can be translated to a constraint program and solved efficiently using propagators for 0/1 variables provided by Oz. To evaluate the competitiveness of constraint programming in the satisfiability domain, an evaluation of other SAT-algorithms with respect to `srq`-like problems is necessary. For a comparison of the implementation of propagators for 0/1 variables in different constraint languages, a more comprehensive suite of `srq`-like problems is needed.

Acknowledgements

I would like to thank Jim Propp for inventing SRAT and giving me background information on the puzzle. I thank Gert Smolka for pointing out the importance of consistently using 0/1 variables, Jörg Würtz and Tobias Müller, the designers and implementors of the finite domain system of Oz, for always lending me an ear for my questions, Joachim Walser for commenting on an earlier version of this paper, and last but not least the folks on the newsgroup `rec.puzzles` for solving my puzzles, pointing out bugs and problems and always asking for more.

References

[1] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. Association for Computing Machinery, New York, 1971.

[2] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. of AAAI-93*, Washington, DC, 1993.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[4] M. Henz. Collected information on SRATs, available via WWW from `http://ps-www.dfki.uni-sb.de/~henz/oz/puzzles/srat/`, Apr. 1996.

[5] T. le Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16:319–359, 1993.

[6] J. Propp. private communication, Apr. 1996.

[7] C. Schulte. Solver—a search debugger for Oz. In *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 29 November–1 December 1995.

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

[9] G. Smolka and R. Treinen. DFKI Oz documentation series. Available on paper or via WWW from `http://ps-www.dfki.uni-sb.de/oz/`, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.