Logical Methods in Computer Science
Vol. 8(3:30)2012, pp. 1–39
www.lmcs-online.org

Submitted Mar. 8, 2011
Published Sep. 30, 2012

URSA: A SYSTEM FOR UNIFORM REDUCTION TO SAT

PREDRAG JANIČIĆ

Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia
e-mail address: janicic@matf.bg.ac.rs

Abstract. There are a huge number of problems, from various areas, being solved by reducing them to SAT. However, for many applications, translation into SAT is performed by specialized, problem-specific tools. In this paper we describe a new system for uniform solving of a wide class of problems by reducing them to SAT. The system uses a new specification language URSA that combines imperative and declarative programming paradigms. The reduction to SAT is defined precisely by the semantics of the specification language. The domain of the approach is wide (e.g., many NP-complete problems can be simply specified and then solved by the system) and there are problems easily solvable by the proposed system, while they can be hardly solved by using other programming languages or constraint programming systems. So, the system can be seen not only as a tool for solving problems by reducing them to SAT, but also as a general-purpose constraint solving system (for finite domains). In this paper, we also describe an open-source implementation of the described approach. The performed experiments suggest that the system is competitive to state-of-the-art related modelling systems.

1. Introduction

Following spectacular advances made over the last years, the SAT solving technology has many successful applications nowadays — there is a wide range of problems being solved by reducing them to SAT. Very often solving based on reduction to SAT is more efficient than using a problem-specific solution. Therefore, SAT solvers are already considered to be a *Swiss army knife* for solving many hard CSP and NP-complete problems and in many areas including software and hardware verification, model checking, termination analysis, planning, scheduling, cryptanalysis, electronic design automation, etc. [6, 3, 12, 25, 74, 48, 50]. Typically, translations into SAT are performed by specialized, problem-specific tools. However, using a general-purpose system capable of reducing a wide range of problems to SAT can simplify this task, make it less error prone, and make this approach more easily accessible and more widely accepted.

There are already a number of approaches for solving combinatorial and related problems by general-purpose systems that reduce problems to underlying theories and domains (instead of developing special purpose algorithms and implementations). A common motivation is that it is much easier to develop a problem specification for a general system than a

1998 ACM Subject Classification: F.4.1, D.3.2.
Key words and phrases: SAT problem, constraint solving, specification languages.

new, special-purpose solver. The general problem solving systems include libraries for general purpose programming languages, but also modelling and programming languages built on top of specific solvers [36, 33, 42, 14]. Most modelling languages are highly descriptive and have specific language constructs for certain sorts of constraints. Specific constraints are translated to underlying theories by specific reduction techniques. Some modelling systems use SAT as the target problem [31, 35] and some of them focus on solving NP-complete problems by reduction to SAT [10].

In this paper we present a novel approach for solving problems by reducing them to SAT. The approach can be seen also as a general-purpose constraint programming system (for finite domains). The approach consists of a new specification/modelling language URSA (from *Uniform Reduction to SAt*) and an associated interpreter. In contrast to other modelling languages, the proposed language combines features of declarative and imperative programming paradigms. What makes the language declarative is not how the constraints are expressed, but only the fact that a procedure for finding solutions does not need to be explicitly given. On the other hand, the system has features of imperative languages and may be seen as an extension of the imperative programming paradigm,[1] similarly as some constraint programming systems are extensions of logic programming paradigm. In contrast to other modelling languages, in the proposed specification language loops are represented in the imperative way (not by, generally more powerful, recursion), destructive updates are allowed, there is support for constraints involving bitwise operators and operators for arithmetic modulo 2^n. There are problems for which, thanks to these features, the modelling process is simpler and specifications are more readable and easier to maintain than for other languages and constraint systems. However, of course, the presented system does not aim to replace other constraint systems and languages, but rather to provide a new alternative with some distinctive features.

The used uniform approach enables a simple syntax and semantics of the URSA language, a simple, uniform reduction to SAT and, consequently, a simple architecture of the whole system. This enables a straightforward implementation of the proposed system and a rather straightforward verification of its correctness. This is very important because, although it is often easier for a declarative program than for a corresponding imperative program to verify that it meets a given specification, this still does not lead to a high confidence if the constraint solving system itself cannot be trusted.

The presented approach is accompanied with an open-source implementation, publicly available on the Internet. A limited experimental comparison suggest that the system (combined with state-of-the-art SAT solvers) yields good performance, competitive to other modern approaches.

Overview of the paper. In Section 2 we give relevant definitions; in Section 3 we provide motivation and basic ideas of the proposed approach. In Section 4 we describe the specification language URSA, in Section 5 its semantics, in Section 6 the corresponding interpreter, and in Section 7 pragmatics of the language. In Section 8 we discuss related techniques, languages and tools. In Section 9 we discuss directions for future work and in Section 10 we draw final conclusions.

[1]There are constraint programming libraries for imperative languages, but these still do not follow the spirit of imperative programming and are substantially declarative.

2. Background

In this section we give a brief account of the SAT and CSP problems and related notions.

Propositional logic. We assume standard notions of propositional logic: *literal*, *clause*, *propositional formula*, *conjunctive normal form* (CNF), *valuation* (or *assignment*), *interpretation*, *model*, *satisfiable formula*, etc. We denote by $\top$ and $\bot$ the Boolean constants *true* and *false* and the logical connectives by $\neg$ (*negation*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\oplus$ (*exclusive disjunction*), $\Rightarrow$ (*implication*), $\Leftrightarrow$ (*equivalence*). Two formulae A and B are said to be *equivalent* if A and B have the same truth value in any valuation. Two formulae A and B are said to be *weakly equivalent* (or *equisatisfiable*) if whenever A is satisfiable then B is satisfiable and vice versa.

Constraint Satisfaction Problem. A constraint satisfaction problem (CSP) is defined as a triple (X, D, C), where X is a finite set of variables x_1, x_2, …, x_n, D is a set of domains d_1, d_2, …, d_n for these variables, and C is a set of constraints c_1, c_2, …, c_k. In a finite-domain CSP, all sets from D are finite. Constraints from C may define combinations of values assigned to variables that are *allowed* or that are *prohibited.* A problem instance is satisfiable if there is an assignment to variables such that all constraints are satisfied. Such assignment is called a *solution.* A constraint optimization problem is a CSP in which the goal is to find a solution maximizing (or minimizing) a given *objective function* over all allowed values of the given variables.

SAT Problem and SAT Solvers. SAT is the problem of deciding if a given propositional formula in CNF is satisfiable, i.e., if there is any assignment to variables such that all clauses are true. Obviously, SAT is a special case of CSP, with all variables ranging over the domain $\{0, 1\}$ and with constraints given as clauses. SAT was the first problem shown to be NP-complete [15], and it still holds a central position in the field of computational complexity. Stochastic SAT solvers cannot prove the input instance to be unsatisfiable, but may find a solution (i.e., a satisfying variable assignment) for huge satisfiable instances quickly. On the other hand, for a given SAT instance, a complete SAT solver always finds a satisfying variable assignment or shows that there is no such assignment. Most of the state-of-the-art complete SAT solvers are CDCL (conflict-driven, clause-learning) based extensions of the Davis-Putnam-Logemann-Loveland algorithm (DPLL) [18, 17, 3]. In recent years, a tremendous advance has been made in SAT solving technology [52, 22, 75, 3]. These improvements involve both high-level and low-level algorithmic techniques. The advances in SAT solving make possible deciding satisfiability of some industrial SAT problems with tens of thousands of variables and millions of clauses.

3. Problem Specification and Problem Solving

There are two basic components of the presented approach:

- problem specification: a problem is specified by a test (expressed in an imperative form) that given values of relevant variables are indeed a solution to the problem.
- problem solving: all relevant variables of the problem are represented by finite vectors of propositional formulae (corresponding to vectors of bits and to binary representation, in case of numerical values); the specification is symbolically executed over such representation and the assertion that given values make a solution is transformed to an instance

of the SAT problem and passed to a SAT solver. If the formula is satisfiable, its model is transformed back to variables describing the problem, giving a solution to the problem.

3.1. **Problem Specification.** Let us consider problems of the following general form: *find (if it exists) a set of values S such that given constraints are met* (variations of this form include: only checking if such values exists, and finding all values that meet the given conditions). A problem of this form can be specified by a test that checks if a given set S meets the given constraints (with one assertion that combines all the constraints). The test can be formulated in a language designed in the style of imperative programming languages and such a test is often easy to formulate.

Example 3.1. Let us consider a trivial problem: if v equals $u+1$, find a value for u such that v equals 2. A simple check in an imperative form can be specified for this problem — if a value of u is given in advance, one could easily check whether it is a solution of the problem. Indeed, one would assign $u+1$ to v and finally check whether v equals 2. Such test can be written in the form of an imperative C-like code (where `assert(b)` checks whether `b` is true) as follows:

```
v = u+1;
assert(v==2);
```

The example above is trivial, but specifications may involve more variables and more complex operations, including conditional operations and loops, as illustrated by the following example.

Example 3.2. The most popular way of generating pseudorandom numbers is based on linear congruential generators. A generator is defined by a recurrence relation of the form:

$$x_{n+1} \equiv (ax_n + c) \pmod{m} \quad (\text{for } n \geq 0)$$

and x_0 is the *seed* value ($0 \leq x_0 < m$). One example of such relation is:

$$x_{n+1} \equiv (1664525x_n + 1013904223) \pmod{2^{32}} \quad (\text{for } n \geq 0)$$

It is trivial to compute elements of this sequence. The check that x_{100} is indeed equal to the given value if the seed is equal to *nseed* can be simply written in the form of an imperative C-like code as follows (assuming that numbers are represented by 32 bits):

```
nx=nseed;
for(ni=1;ni<=100;ni++)
  nx=nx*1664525+1013904223;
assert(nx==3998113695);
```

However, the following problem, realistic in simulation and testing tasks, is a non-trivial programming problem (unless problem-specific, algebraic knowledge is used): given, for example, the value x_{100} compute x_0. Still, the very same test shown above can serve as a specification of this problem. This example illustrates one large family of problems that can be simply specified using the proposed approach — problems that are naturally expressed in terms of imperative computations and that involve destructive assignments. Such problems are often difficult to express using other languages and systems. For the above specification, since in constraint programming systems the destructive assignment is not allowed, in most specification languages one would have to introduce variables for all elements of the sequence from x_0 to x_{100} and the constraints between any succeeding two. Also, other systems typically do not support modular arithmetic constraints and integers of arbitrary length.

Note that the specifications given above also cover the information on what variables are unknown and have to be determined so that the constraints are satisfied — those are variables that appear within commands before they were defined. So, the above code is a full and precise specification of the problem, up to the domains of the variables. For Boolean variables, the domain is $\{\bot, \top\}$, while for numerical variables a common domain interval (e.g., $[0, 2^n - 1]$) can be assumed and additional domain restrictions can be given within the specification.

3.2. **Problem Solving.** The described imperative tests form problem specifications and they can be used as a starting point in problem solving. Let us first describe a straightforward, naive approach.

Assume that there is a problem specification (in the form of an imperative test) and assume there is a common domain for all unknowns (except Boolean unknowns), for example, the interval $[0, 2^n - 1]$ (for a given n). Then, for all admissible values for all unknowns, the specification can be executed. All sets of values satisfying the constraints should be returned as solutions. If there are k unknown numerical variables and l unknown Boolean variables, then the search space would be of the size 2^{nk+l}.

Example 3.3. Let us consider the specification given in Example 3.1. If the domain for u and v is the interval $[0, 3]$, the specification should be executed four times and only the value 1 assigned to u leads to the constraint met, so it is the only solution of the problem.

Obviously, the above naive and brute-force approach based on systematic enumeration of all possible input values is complete (for finding all solutions), but extremely inefficient. It can be turned to a much more efficient version that takes into account given relationships between variables in order to reduce the search space. The basic idea is to represent all unknowns abstractly, in a symbolic form, as vectors of propositional formulae. Then, all steps in the specification can be performed using this abstracted form (i.e., can be symbolically executed). Finally, the assertion would generate a propositional formula for which a satisfying valuation is to be found. If there are k unknown numerical variables and l unknown Boolean variables, then the number of possible valuations would be 2^{nk+l} (if the interval $[0, 2^n - 1]$ is assumed as the domain for numerical values). Of course, instead of a brute-force search over this set of valuations, a SAT solver should be used (and it will typically perform many cut-offs and search over just a part of the whole search space).

Representation of numerical variables by propositional formulae corresponds to their binary representation. Each formula corresponds to one bit of the binary representation. If the range of a numerical variable is $[0, 2^n - 1]$, then it is represented by a vector of n propositional formulae. If a bit of the number is known to be 1, then the corresponding formula is $\top$, and if a bit of the number is known to be 0, then the corresponding formula is $\bot$. For instance, for $n = 2$, 1 is represented by $[\bot, \top]$ (where the last position corresponds to the least significant bit). If a bit of the number is not known, then it is represented by a propositional variable, or, if it depends on some conditions, by a propositional formula. We will discuss only representations of unsigned integers, but representations of signed integers can be treated in full analogy (moreover, floating point numbers can also be modelled in an analogous way). Boolean variables are represented by unary vectors of propositional formulae.

Results of arithmetic and bitwise logical operations over numbers represented by vectors of formulae can be again represented by propositional formulae in terms of formulae occurring in the input arguments. If the numbers are treated as unsigned, all arithmetic operations are performed modulo 2^n. For instance, if u is represented by $[p, q]$ and v is represented by $[r, s]$, then $v+u$ (modulo 2^2) is represented by $[(p\oplus r)\oplus(q\wedge s), q\oplus s]$. Relational operations over numbers ($=$, $<$, $>$, $\leq$, $\geq$, $\neq$, etc.) and logical operations and relations over Boolean values can also be represented. For instance, if u is represented by $[p, q]$ and v is represented by $[r, s]$, then $u = v$ is represented by the unary vector $[(p \Leftrightarrow r)\wedge(q \Leftrightarrow s)]$. Note that representations of all standard arithmetic, Boolean, and relational operations produce polynomial size formulae.

If a problem specification is executed over the variables represented by vectors of propositional variables and using the corresponding interpretation of involved operations, then the assertion of the specification generates a propositional formula. Any satisfying valuation (if it exists) for that formula would yield (ground) values for numerical and Boolean unknowns that meet the specification, i.e., a solution to the problem.

Example 3.4. Let us again consider Example 3.1. If `u` is represented by $[p, q]$ (and `1` is represented by $[\bot, \top]$), then, by the condition `v=u+1`, `v` is represented by $[(p\oplus\bot)\oplus(q\wedge\top), q\oplus\top]$, i.e., after simplification, by $[p\oplus q, \neg q]$. From the assertion `v==2`, it follows that $[p\oplus q, \neg q]$ should be equivalent to $[\top, \bot]$. In other words, the formula $((p\oplus q) \Leftrightarrow \top)\wedge(\neg q \Leftrightarrow \bot)$ should be checked for satisfiability. It is satisfiable, and in its only model p maps to $\bot$ and q maps to $\top$. Hence, the representation for a required value of `u` is $[\bot, \top]$, i.e., `u` equals 1.

3.3. **Domain of the Approach.**

A system based on the ideas presented above, could be used not only for combinatorial problems, but for a very wide range of problems — it can be used for computing x such that $f(x) = y$, given y and a computable function f with a finite domain and a finite range (i.e., for computing inverse of f). A definition of f in an imperative form can serve as a specification of the problem in verbatim. Having such a specification of the function f is a weak and realistic assumption as it is easy to make such specification for many interesting problems, including NP-problems [26]. If f is a function such that $f(x) = 1$ when x is a witness for some instance of an NP-problem, f can serve as a specification for this problem and the required answer is *yes* if and only if there is x such that $f(x) = 1$.

Concerning the type of numbers involved, the approach can be applied for any finite representation of signed or unsigned, integer or floating point numbers.

In the proposed approach, all computations (over integers) are performed modulo 2^n. In the case of non-modular constraints, the base can be set to a sufficiently large value.

The approach (in the presented form) cannot be used for computing x such that $f(x) = y$, for arbitrary computable function f. The first limitation is a finite representation of variables. The second is that conditional commands in the specification could involve only conditions with ground values at the time when the condition is evaluated. However, this restriction is not relevant for many (or most of) interesting problems. Overall, the domain of the proposed approach covers all problems with Boolean and numerical unknowns, over Boolean parameters and numerical parameters with finite domains, that can be stated in the specification language that makes the part of the approach.

4. Syntax of ursa Language

In this section we describe the language URSA that serves as a specification language in the spirit of the approach presented above. A description of the syntax of the URSA language is given, in EBNF representation, in Table 1 (⟨num var⟩ denotes the syntactical class of numerical variables, ⟨num expr⟩ denotes the syntactical class of numerical expressions, ⟨bool expr⟩ denotes the syntactical class of Boolean expressions, etc). An URSA program is a sequence of statements (and procedure definitions). There are two types of variables — numerical, with identifiers starting with `n` and Boolean, with identifiers starting with `b`. The same convention holds for identifiers of arrays. Variables are not declared, but introduced dynamically. There are functions (`bool2num` and `num2bool`) for converting Boolean values to numerical values and vice versa, and the `sgn` function corresponding to signum function. Arithmetic, bitwise, relational and compound assignment operators, applied over arithmetic variables/expressions, are written in the C-style. For example, bitwise conjunction over numerical variables `n1` and `n2` is written `n1 & n2`, bitwise left shift of `n1` for `n2` is written `n1 << n2`, and `n1 += n2` is equivalent to `n1 = n1+n2`. Logical operators, applied over Boolean variables/expressions, are written in the C-style, with additional operator `^^` for logical exclusive disjunction, in the spirit of other C logical operators. There are also compound assignment operators for logical operators, such as `&&=` (added for symmetry and convenience, although they do not exist in C). The operator `ite` is the conditional operator: `ite(b,n1,n2)` equals `n1` if `b` is true, and equals `n2` otherwise. There are no user-defined functions, but only user-defined procedures.

The role of the command `assert` is to assert that some constraint (given as the argument) is met (as in C). During the interpretation of the URSA program, this command invokes the solving mechanism and seeks for an assignment to the introduced unknowns that make this constraint true. The command `assert_all` is analogous, but it seeks for all satisfying models. One program can have several commands of this sort, but each of them works locally (as in C), i.e., it invokes the solving mechanism just for its own argument. The instructions `minimize` (and `maximize`) state that a minimal (or maximal) value within the given range for the given numerical variable should be found. The commands `assert` and `assert_all` take into account only the last `minimize`/`maximize` instruction.

There are miscellaneous commands, mostly intended to be used in interactive mode (for listing values and status of variables — `listvars`, for deleting values of all current variables — `clear`, and stopping the interpreter — `halt`).

The following two examples illustrate some constructs of the specification language — the use of procedures and the use of the operator `maximize`.

Example 4.1. For a given value k, the task is to find all values x, y, and z such that $x^k + y^k \equiv z^k (\mathrm{mod}\ 2^n)$. The following URSA code (with a procedure that computes the power function) specifies the problem for $k = 2$:

```
procedure power(na,nk) {
  np=na;
  for(ni=1;ni<nk;ni=ni+1)
     na = na*np;
}

nk=2;
nxpowernk=nx;
nypowernk=ny;
```

⟨program⟩	::=	⟨procedure def⟩* ⟨statement⟩*
⟨procedure def⟩	::=	"procedure" ⟨procedure name⟩ ("(" ")" |
		"(" (⟨num var id⟩ | ⟨bool var id⟩) ("," (⟨num var id⟩ | ⟨bool var id⟩))* ")")
		"{" ⟨statement⟩* "}"
⟨procedure name⟩	::=	⟨letter⟩(⟨letter⟩ | ⟨digit⟩)*
⟨statement⟩	::=	"{" ⟨statement⟩* "}"
		| ⟨num var⟩ ⟨assign num op⟩ ⟨num expr⟩ ";"
		| ⟨num var⟩ ⟨num op postfix⟩ ";"
		| ⟨bool var⟩ ⟨assign bool op⟩ ⟨bool expr⟩ ";"
		| "while" "(" ⟨bool expr⟩ ")" ⟨statement⟩
		| "for" "(" ⟨statement⟩ ";" ⟨bool expr⟩ ";" ⟨statement⟩ ")" ⟨statement⟩
		| "if" "(" ⟨bool expr⟩ ")" ⟨statement⟩ ["else" ⟨statement⟩]
		| "call" ⟨procedure name⟩ ("(" ")" |
		"(" (⟨num expr⟩ | ⟨bool expr⟩) ("," (⟨num expr⟩ | ⟨bool expr⟩))* ")") ";"
		| "minimize" "(" ⟨num var⟩ "," ⟨num const⟩ "," ⟨num const⟩ ")" ";"
		| "maximize" "(" ⟨num var⟩ "," ⟨num const⟩ "," ⟨num const⟩ ")" ";"
		| "assert" "(" ⟨bool expr⟩ (";" ⟨bool expr⟩)* ")" ";"
		| "assert_all" "(" ⟨bool expr⟩ (";" ⟨bool expr⟩)* ")" ";"
		| "print" (⟨num expr⟩ | ⟨bool expr⟩) ";"
		| "listvars" ";"
		| "clear" ";"
		| "halt" ";"
⟨num expr⟩	::=	⟨num const⟩
		| ⟨num var⟩
		| ⟨un num op⟩ ⟨num expr⟩
		| ⟨num expr⟩ ⟨num op⟩ ⟨num expr⟩
		| ite "(" ⟨bool expr⟩ "," ⟨num expr⟩ "," ⟨num expr⟩")"
		| "sgn" "(" ⟨num expr⟩ ")"
		| "bool2num" "(" ⟨bool expr⟩ ")"
		| "(" ⟨num expr⟩ ")"
⟨num var⟩	::=	⟨num var id⟩
		| ⟨num var id⟩ "[" ⟨num expr⟩ "]"
		| ⟨num var id⟩ "[" ⟨num expr⟩ "]" "[" ⟨num expr⟩ "]"
⟨num const⟩	::=	(⟨digit⟩)+
⟨num var id⟩	::=	"n"(⟨letter⟩ | ⟨digit⟩)*
⟨assign num op⟩	::=	"=" | "+=" | "-=" | "*=" | "&=" | "|=" | "^=" | "<<=" | ">>="
⟨num op⟩	::=	"+" | "-" | "*" | "&" | "|" | "^" | "<<" | ">>"
⟨un num op⟩	::=	"−" | "~"
⟨num op postfix⟩	::=	"++" | "--"
⟨num rel⟩	::=	"<" | ">" | "<=" | ">=" | "==" | "!="
⟨bool expr⟩	::=	⟨bool const⟩
		| ⟨bool var⟩
		| ⟨bool expr⟩ ⟨bool op⟩ ⟨bool expr⟩
		| ⟨un bool op⟩ ⟨bool expr⟩
		| ⟨num expr⟩ ⟨num rel⟩ ⟨num expr⟩
		| ite "(" ⟨bool expr⟩ "," ⟨bool expr⟩ "," ⟨bool expr⟩")"
		| "num2bool" "(" ⟨num expr⟩ ")"
		| "(" ⟨bool expr⟩ ")"
⟨bool const⟩	::=	("true" | "false")
⟨bool var⟩	::=	⟨bool var id⟩
		| ⟨bool var id⟩ "[" ⟨num expr⟩ "]"
		| ⟨bool var id⟩ "[" ⟨num expr⟩ "]" "[" ⟨num expr⟩ "]"
⟨bool var id⟩	::=	"b"(⟨letter⟩ | ⟨digit⟩)*
⟨assign bool op⟩	::=	"=" | "&&=" | "||=" | "^^="
⟨bool op⟩	::=	"&&" | "||" | "^^"
⟨un bool op⟩	::=	"!"

Table 1: EBNF description of URSA language

```
nzpowernk=nz;

call power(nxpowernk,nk);
call power(nypowernk,nk);
call power(nzpowernk,nk);

assert_all(nxpowernk+nypowernk==nzpowernk);
```

Example 4.2. Given nine points in space, no four of which are coplanar, the task is to find the minimal natural number n such that for any coloring with red or blue of n edges drawn between these nine points there always exists a triangle having all edges of the same color.[2]

We will slightly reformulate the problem: it is sufficient to find the maximal natural number n such that there is a coloring with red or blue of n edges drawn between these nine points such there is no triangle having all edges of the same color. We will assume that `nE[i][j]` is 0 if there is no edge linking i-th and j-th point, that `nE[i][j]` equals 1 if the edge linking i-th and j-th point is red, and that `nE[i][j]` equals 2 if the edge linking i-th and j-th point is blue. In the following specification, `nNumberOfEdges` stores the number of edges, `bTwoColors` stores the condition that for each pair of points, either there is no edge, or it is red or blue, and `bNoMonochromaticTriangle` stores the condition that there is no triangle having all edges of the same color. In order to use the maximal n that meets these conditions, the command `maximize(n,1,36)` is used (there are 36 edges at most).

```
maximize(n,1,36);
nPoints=9;
nNumberOfEdges=0;
bTwoColors=true;
bNoMonochromaticTriangle=true;

for(ni=1;ni<=nPoints-1;ni++)
  for(nj=ni+1;nj<=nPoints;nj++)  {
    nNumberOfEdges += sgn(nE[ni][nj]);
    bTwoColors &&= nE[ni][nj]<3;
    for(nk=nj+1;nk<=nPoints;nk++)
      bNoMonochromaticTriangle &&=
         (nE[ni][nj]==0 || nE[ni][nj]!=nE[ni][nk] || nE[ni][nj]!=nE[nj][nk]);
  }

assert(nNumberOfEdges==n && bTwoColors && bNoMonochromaticTriangle);
```

5. Semantics of ursa Language

The semantics of the ursa language is not equal, but rather parallel to the standard semantics of imperative programming languages. Namely, while in the standard semantics expressions (numerical and Boolean) are always evaluated to ground values, in ursa they may be represented in symbolic propositional form. In terms of operational semantics [59],

[2] This is one of the problems from International Mathematical Olympiad (IMO) held in 1992. Mathematical problems from IMOs are typically very challenging problems from different areas of mathematics, often coming from complex mathematical conjectures, but not requiring heavy mathematical devices themselves (as they are aimed at high-school students) [20]. A number of other IMO problems can be specified and solved using the presented system.

in the standard semantics, a *store* (intuitively, describing memory) is a function from identifiers to integers, while for the URSA language, a store is a function from identifiers to integers *or* vectors of propositional formulae. Boolean variables are represented by unary vectors, while numerical variables are represented by vectors of length n (where n is chosen in advance and then fixed for one session of the interpreter's work).

A configuration is a pair $\langle c, s\rangle$ of a command or an expression c and a store s. A one step relation $\mapsto$ maps configurations to configurations. In the following rule descriptions, $\langle skip, s\rangle$ denotes a terminal configuration — a program that has completed execution, x denotes a variable identifier, i, i_1, and i_2 denote integers, f_i, f'_i, and f''_i denote propositional formulae, e, e', e_1 and e_2 denote expressions, and $s \uplus (x \mapsto y)$ denotes a function $\hat{s}$ such that $\hat{s}(x) = y$ and $\hat{s}(z) = s(z)$ for $z \neq x$.

In the standard semantics, for example, the assignment operator = is defined by the rules:

(1) $\langle x = i, s\rangle \;\mapsto\; \langle skip, s \uplus (x \mapsto i)\rangle$

(2) $\dfrac{\langle e, s\rangle \;\mapsto\; \langle e', s'\rangle}{\langle x = e, s\rangle \;\mapsto\; \langle x = e', s'\rangle}$

Since in the URSA language a variable can be assigned both a ground integer value or a vector of propositional formulae, the semantics of the assignment operator in URSA is defined by the rules (the rule (1) from above is split into two rules):

(1'a) $\langle x = i, s\rangle \;\mapsto\; \langle skip, s \uplus (x \mapsto i)\rangle$

(1'b) $\langle x = [f_1, \dots, f_n], s\rangle \;\mapsto\; \langle skip, s \uplus (x \mapsto [f_1, \dots, f_n])\rangle$

(2') $\dfrac{\langle e, s\rangle \;\mapsto\; \langle e', s'\rangle}{\langle x = e, s\rangle \;\mapsto\; \langle x = e', s'\rangle}$

If, in an assignment command, the right hand side is evaluated to a ground integer (i.e., if the rule (1'a) has been applied), then the variable on the left hand side gets the status *ground*. If the right hand side is evaluated to a vector of formulae (i.e., if the rule (1'b) has been applied), then the variable on the left hand side gets the status *symbolic* and *dependent*.

In the standard semantics, for an operator $\diamond$ over integers, the relation $\mapsto$ is defined for expressions by the following four rules:

(3) $\langle x, s\rangle \;\mapsto\; \langle s(x), s\rangle$

(4) $\langle i_1 \diamond i_2, s\rangle \;\mapsto\; \langle i, s\rangle$ (where i equals $i_1 \diamond i_2$)

(5) $\dfrac{\langle e_1, s\rangle \;\mapsto\; \langle e'_1, s'\rangle}{\langle e_1 \diamond e_2, s\rangle \;\mapsto\; \langle e'_1 \diamond e_2, s'\rangle}$

(6) $\dfrac{\langle e_2, s\rangle \;\mapsto\; \langle e'_2, s'\rangle}{\langle i \diamond e_2, s\rangle \;\mapsto\; \langle i \diamond e'_2, s'\rangle}$

The corresponding rules for the URSA language are substantially different. Regarding the rule (3), in standard imperative programming languages, a statement attempting to evaluate an undefined variable (one that is not in the domain of the current store) results in a runtime error. On the other hand, in the URSA language, undefined variables can be used in expressions, so this introduces a non-standard rule (instead of the rule (3))[3]:

[3]More precisely, rather than a rule, this is a rule-schema, with instances for each n.

$$\text{(3')}\quad \langle x, s\rangle \;\mapsto\; \begin{cases} \langle s(x), s\rangle & \text{if } x \text{ is defined in } s \\ \langle [v_1,\dots,v_n], s \uplus (x \mapsto [v_1,\dots,v_n])\rangle & \text{otherwise } (v_i\text{'s are fresh} \\ & \text{propositional variables)} \end{cases}$$

If, for a variable x, the second case of the above rule has been applied, x gets the status *symbolic* and *independent* (and is represented by a vector of propositional formulae). In other words, this status is associated to each variable that is for the first time used not on the left hand side of an assignment operator. For example, if a variable `nX` was not defined beforehand, after the statement `nY=nX;`, both variables are to be represented by the same sequence of propositional variables, but `nX` will internally have the status *independent*, while `nY` will have the status *dependent*. Variables with the status *symbolic* and *independent* are those that should be determined in order to solve the given problem.

Example 5.1. Let us consider the specification for the trivial problem from Example 3.1:

```
nv = nu+1;
assert(nv==2);
```

The variable `nu` is used for the first time within the first command, so it is symbolic (internally represented by a vector of propositional formulae) and independent. The variable `nv` becomes symbolic and dependent (it depends on `nu`).

Since in URSA expressions (numerical and Boolean) may be represented in symbolic propositional form, the rule (4) is replaced by the following four rules:

(4'a) $\langle i_1 \diamond i_2, s\rangle \;\mapsto\; \langle i, s\rangle$ (where i equals $i_1 \diamond i_2$)
(4'b) $\langle i \diamond [f'_1,\dots,f'_n], s\rangle \;\mapsto\; \langle [f_1,\dots,f_n] \diamond [f'_1,\dots,f'_n], s\rangle$
(where $[f_1,\dots,f_n]$ is a binary representation of i)
(4'c) $\langle [f_1,\dots,f_n] \diamond i, s\rangle \;\mapsto\; \langle [f_1,\dots,f_n] \diamond [f'_1,\dots,f'_n], s\rangle$
(where $[f'_1,\dots,f'_n]$ is a binary representation of i)
(4'd) $\langle [f_1,\dots,f_n] \diamond [f'_1,\dots,f'_n], s\rangle \;\mapsto\; \langle [f''_1,\dots,f''_n], s\rangle$

where, in the last rule, $[f''_1,\dots,f''_n]$ is a vector of propositional formulae corresponding to $[f_1,\dots,f_n] \diamond [f'_1,\dots,f'_n]$, assuming that all numerical expressions are considered unsigned[4] and all arithmetic operations are performed modulo 2^n. For example, assuming that numbers are represented by vectors of length 2 (i.e., $n = 2$), the rule (4'd) for the operator $+$ is as follows:

$$\langle [f_1, f_2] + [f'_1, f'_2], s\rangle \;\mapsto\; \langle [(f_1 \oplus f'_1) \oplus (f_2 \wedge f'_2), f_2 \oplus f'_2], s\rangle$$

As another example, assuming that numbers are represented by vectors of length 2 (i.e., $n = 2$), the operator $>$ is defined as follows (in accordance with lexicographic ordering):

$$\langle [f_1, f_2] > [f'_1, f'_2], s\rangle \;\mapsto\; \langle [(f_1 \wedge \neg f'_1) \vee ((f_1 \Leftrightarrow f'_1) \wedge (f_2 \wedge \neg f'_2)], s\rangle$$

By the above semantics, an expression involving symbolic expressions is also a symbolic expression. However, for efficiency reasons, if it is possible, an expression will be computed to a ground value. For instance, after the statement `nA = nB * 0;`, the variable `nA` will have a value `0` and after the statement `bX = bY && false;`, the variable `bX` will have the

[4] Dealing with representation of signed integers or floating point numbers can be, in principle, described and implemented by analogy (`http://www.informatik.uni-bremen.de/~florian/sonolar/`, `http://www.cprover.org/SMT-LIB-Float/` [13, 7]). Still, as discussed by Brillout et al. [7], the bottleneck of a reducing floating point operations to propositional logic would be in the complexity of the resulting propositional formulae.

ground value *false*, even if the variables `bB` and `bY` were symbolic. This is described by the modified version of the rule (4'd):

$$\text{(4"d)}\quad \langle [f_1,\dots,f_n]\diamond[f'_1,\dots,f'_n],s\rangle \;\mapsto\; \begin{cases} \langle i,s\rangle & \text{if all } f''_1,\ \dots,\ f''_n \text{ are Boolean constants, where } i \text{ is a number with the binary representation } [f''_1,\dots,f''_n] \\ \langle [f''_1,\dots,f''_n],s\rangle & \text{otherwise} \end{cases}$$

While the rule (5) from above is kept unchanged for the URSA language, the rule (6) is split into two rules, one for ground argument and one for symbolic argument:

$$\text{(5')}\quad \frac{\langle e_1,s\rangle \;\mapsto\; \langle e'_1,s'\rangle}{\langle e_1\diamond e_2,s\rangle \;\mapsto\; \langle e'_1\diamond e_2,s'\rangle}$$

$$\text{(6'a)}\quad \frac{\langle e_2,s\rangle \;\mapsto\; \langle e'_2,s'\rangle}{\langle i\diamond e_2,s\rangle \;\mapsto\; \langle i\diamond e'_2,s'\rangle}$$

$$\text{(6'b)}\quad \frac{\langle e_2,s\rangle \;\mapsto\; \langle e'_2,s'\rangle}{\langle [f_1,\dots,f_n]\diamond e_2,s\rangle \;\mapsto\; \langle [f_1,\dots,f_n]\diamond e'_2,s'\rangle}$$

The semantics of `while` is defined by the following standard rules:

$$\text{(7)}\quad \frac{\langle b,s\rangle \;\mapsto\; \langle \top,s'\rangle}{\langle \texttt{while}\ b\ \texttt{do}\ c,s\rangle \;\mapsto\; \langle c;\texttt{while}\ b\ \texttt{do}\ c,s'\rangle}$$

$$\text{(8)}\quad \frac{\langle b,s\rangle \;\mapsto\; \langle \bot,s'\rangle}{\langle \texttt{while}\ b\ \texttt{do}\ c,s\rangle \;\mapsto\; \langle \mathit{skip},s'\rangle}$$

If, for the statement `while` b `do` c, neither of $\langle b,s\rangle \mapsto \top$ and $\langle b,s\rangle \mapsto \bot$ holds (i.e., if b is evaluated to a propositional formulae), then a run-time error is raised. The semantics of `for` is defined by analogy. This is a restriction of the specification language and it is difficult to overcome, as it would require indefinite loop unrolling. Recursion is also not supported and an index for accessing an array element has to be a ground number. A condition for the `if` statement has to evaluate to a ground Boolean value, too (otherwise a run-time error is raised). However, as a substitute, there is a conditional statement `ite` (for *if-then-else*) and its condition (the first argument) can be either ground or symbolic. Its semantics, if the last two arguments are of the Boolean type, is defined in the following way (if the arguments are of the numerical type, it is defined for each vector element by analogy):

$$\langle \texttt{ite}(b,[b_1],[b_2],s\rangle \;\mapsto\; \langle (b\Rightarrow b_1)\wedge(\neg b\Rightarrow b_2),s\rangle$$

Arguments to procedures are passed by name if they are variables, and otherwise, they are passed by value.

The semantics of other commands in the URSA language and its relationship with the standard semantics is obvious or analogous to the semantics of the commands given above.

The above, clear and relatively simple semantics, enable a rather straightforward (although still tedious) verification (of correctness) of the proposed system: if the system returns a solution, then this solution indeed meets the given specification, and — if the system does not return a solution, then the specification has no solutions. The correctness property is given by the following theorem ($s_\emptyset$ denotes a function not defined for any argument and $\mapsto$ correspond both to the standard and the URSA semantics since there are no unknowns in the relevant specification).

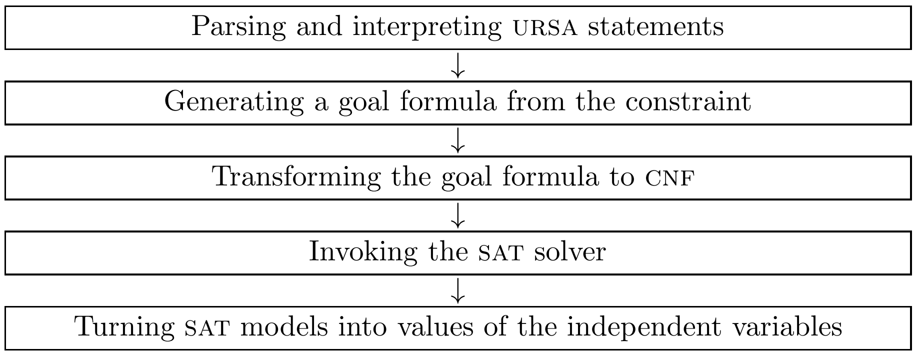

Parsing and interpreting URSA statements
Generating a goal formula from the constraint
Transforming the goal formula to CNF
Invoking the SAT solver
Turning SAT models into values of the independent variables

Figure 1: Overall architecture of URSA system

Theorem 5.2. *If the variables v_1, v_2, …, v_n are (the only) unknowns in an* URSA *specification $S; assert(b);$, then it leads to a solution $(v_1, v_2, \ldots, v_n) = (c_1, c_2, \ldots, c_n)$, (i.e. this solution leads to the model of the formula $s(b)$) iff $\langle v_1 = c_1; v_2 = c_2; \ldots; v_n = c_n; S; assert(b), s_\emptyset\rangle \mapsto \langle skip, s\rangle$ where $s(b)$ is true.* $\square$

The above semantics also ensure *faithfulness* of problem specifications — each model meeting the specification leads to one model of the generated propositional formula, and each model of the generated propositional formula leads to one model of the input specification. This property is essential for constructing (or counting) all solutions of a problem.

6. Interpreter for URSA Language

In this section we describe the implementation of the interpreter for the URSA language.[5]

The interpreter is implemented in the programming language C++. The whole system has a simple architecture and is relatively small (around 100Kb of source code, not counting SAT solver used). The overall architecture of the system is illustrated in Figure 1.

6.1. **Parsing and Interpreting URSA Statements.**

Statements can be entered in an interactive mode and processed one by one or provided to the URSA system within a file. Statements are interpreted according to the semantics described in Section 5.

The table of symbols (i.e., the *store*) stores values of numerical and Boolean variables. Numerical variables are represented by a class (`Number`) whose objects can be either symbolic numbers (i.e., vectors of propositional formulae) or concrete numbers. For this class all standard arithmetic (except division[6]), bitwise logical, and relational operators are defined. In principle, these operators return ground values for ground operands and vectors of formulae for symbolic operands (but, if possible, return ground values for symbolic operands). Boolean variables are represented by analogy (by the class `Boolean`).

[5]The source code of the interpreter and example specifications are available within the distribution of the URSA tool, available online from: `http://www.matf.bg.ac.rs/~janicic/ursa.zip`.

[6]It is possible to define division in symbolic terms, but for its complexity, it is not implemented in the current version of the system. Still, division can be modelled within the system by using multiplication as shown in the following example, giving `nD` as a results of integer division `nX/nY` and `nR` as a remainder (the constraint on `nD` restricts the number of solutions, because of the modular arithmetic): `assert(nX==nY*nD+nR;nR<nY;nD<nX);`

Symbolic values of variables are represented by vectors of propositional formulae. There are classes for dealing with propositional formulae (`Formula`) and vectors of propositional formulae (`FormulaVector`). The length of vectors of propositional formulae representing numerical variables is given as a parameter to the interpreter (the default value is 8, corresponding to integers from 0 to 255).

In the current implementation, all numerical variables and constants are treated as unsigned integers (represented by binary representation). For the class `FormulaVector` all standard arithmetic (except division), bitwise logical, and relational operators are defined. They deal only with symbolic numerical and Boolean values (in contrast to the class `Number` and `Boolean`). The implementations of bitwise logical operators are simple and straightforward. The implementations of the arithmetic operators modulo 2^n and the relational operators are more complex because resulting bits depend on all the previous bits of the operands. Figure 2 shows the implementation of the relational operator $>$ (it processes propositional formulae corresponding to bits from the least significant one and returns a unary vector). Note that the given function is not computing a value for the relational operator $>$ for concrete numbers, but it sets a formula that corresponds to $>$ for two symbolic input numerical expressions.

```
FormulaVector1 FormulaVector::operator > (const FormulaVector &fv) {
  FormulaVector1 result, r1,r2;
  r1 = (*this)[size-1];
  r2 = fv[size-1];
  result = (r1 & ~r2);
  for(int i=size-2;i>=0;i--) {
    r1 = (*this)[i];
    r2 = fv[i];
    result = (r1 & ~r2) | (result & (r1==r2));
  }
  return result;
}
```

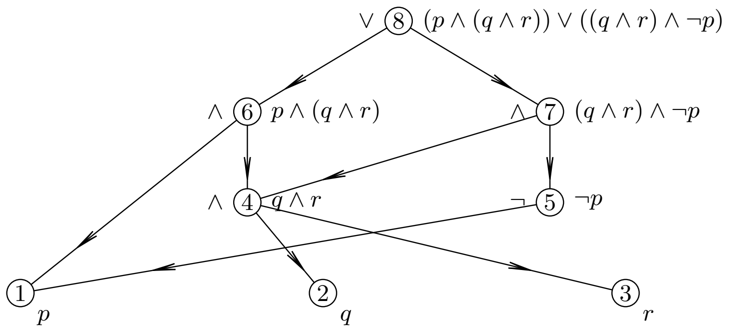
Figure 2: Implementation of the operator $>$.

For the sake of efficiency (both time and space), the technique of shared expressions was used. So, each subformula is stored only once, but there can be more references to it (from different formulae). All formulae generated during the interpretation of the program are stored by the class `FormulaFactory` in this way. Hence, the formulae are not stored individually, but in a form of a directed graph that stores links between them. Each formula that is not a propositional variable is represented by its connective and by references to its subformulae. Each formula is assigned a unique (numerical) identifier.

Example 6.1. Figure 3 illustrates how the (artificial) formula $(p \wedge (q \wedge r)) \vee ((q \wedge r) \wedge \neg p)$ and its subformulae are stored internally.

6.2. **Generating a Goal Formula from the Constraint.**

When executing the command `assert`, the interpreter invokes the underlying solving process. For the given constraint (given as an argument to `assert`), a corresponding (single) propositional formula is generated, transformed to CNF and, by a SAT solver, it is checked whether the formula is true in some valuation. If yes, then the system lists values of all independent variables in that



$\vee$ 8 $(p \wedge (q \wedge r)) \vee ((q \wedge r) \wedge \neg p)$
$\wedge$ 6 $p \wedge (q \wedge r)$
$\wedge$ 7 $(q \wedge r) \wedge \neg p$
$\wedge$ 4 $q \wedge r$
$\neg$ 5 $\neg p$
1 p
2 q
3 r

Figure 3: Internal representation of formulae

valuation. The command `assert_all` is similar, but it lists values of independent variables in all satisfying valuations.

Transforming the formula F that corresponds to the constraint to CNF is performed using Tseitin transformation [70] which is linear in both space and time. The central idea of the transformation is to introduce new („definitional“) variables for all subformulae of the input formula, to replace the subformulae with the corresponding variables, and to add conjuncts (in the form of clauses) representing „definitions“ of newly introduced variables. This yields a resulting formula that is not logically equivalent to F (because of the newly introduced variables) but is *equisatisfiable* to it (i.e., the resulting formula is satisfiable if and only if F is satisfiable). Still, each model for the resulting formula gives one model for F and vice versa (thanks to the nature of definitional variables). This approach is suitable for the representation of formulae described in Section 6.1 since all subformulae of the formula representing the constraint are already generated and assigned unique identifiers that correspond to definitional variables.

The problem with the Tseitin transformation is that it introduces many new variables, and consequently many clauses. There are techniques that can reduce the number of variables and clauses, e.g., by using implications instead of equivalences for subformulae that occur in one polarity only [23]. In the current implementation of the URSA system, for reducing (in some cases) the number of variables and clauses, associativity and commutativity of the connectives $\wedge$ and $\vee$ is used.[7] If a formula A is of the form $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ there is no need to introduce new variables for each of $n-1$ conjunctions. Indeed, the standard Tseitin set of resulting clauses for this case can be replaced by:

$$(p_A \vee \neg p_{A_1} \vee \neg p_{A_2} \vee \ldots \vee \neg p_{A_n}) \wedge (\neg p_A \vee p_{A_1}) \wedge (\neg p_A \vee p_{A_2}) \wedge \ldots (\neg p_A \vee p_{A_n})$$

Disjunctions are treated by analogy. In addition, if the initial formula is a conjunction, its conjuncts are put into CNF separately and conjoined. If any of those conjuncts is a clause, it is directly used (without transformation and definitional variables) at the conjunction top level [32]. The above modifications still keep the transformation linear.

[7] The fact that the connectives $\wedge$ and $\vee$ are associative and commutative is used only when the formula is already generated. An alternative would be to deal with n-ary conjunctions and disjunctions in earlier stages, along the generation process.

Thanks to the above modifications, input instances of the SAT problem itself (represented simply, in a way that is close to the DIMACS format) are reduced by the URSA system to the same instances, up to renaming variables.[8]

Typically, reducing the size of a formula is an important objective, but a smaller number of clauses does not necessarily mean an easier formula and sometimes adding the right clauses (e.g., those corresponding to *symmetry breaking*) can have a positive effect to performance of the solver. Still, guessing and adding clauses that could speed-up the solving process is is not performed in the current version of URSA.

6.3. **Invoking the SAT Solver.** The URSA system (currently) uses state-of-the-art SAT solvers ArgoSAT[9] and CLASP.[10] The solvers are called directly through appropriate function calls. Alternatively, the solvers could accept inputs through files in the standard DIMACS form[11] (URSA can be used just to translate the input problem to SAT, without solving it). In that case, the solvers can be used as a black-box, and could be replaced by any SAT solver that accepts this input format (and has an option to generate all models of the input propositional formula).

Since during the transformation to CNF (described above) some variables are eliminated, it is necessary to rename all remaining variables (and update the numbers of variables). Namely, if some variables do not occur in the generated CNF, the solver would consider as different all models that differ only in the values of such variables. The basic, independent variables (those that correspond to independent URSA variables) are never eliminated, even if they don't occur in the generated formula (because their values in all models are relevant).

Along with the generated SAT instance, URSA can also pass to the solver the information on which propositional variables are independent and which are dependent. Some SAT solvers can use this information to guide (and hopefully speed-up) the solving process. If the constraint is given by the `assert_all` command, the solver is invoked to generate all satisfying valuations.[12] When multiple invocations of the solver are used (for finding all solutions), the solving process can benefit from clauses learnt in previous invocations.

If there is a statement `minimize` (or `maximize`) used, in the current implementation, the problem is solved sequentially for all values from the given range assigned to a relevant

[8] Because of the possible renaming of the variables, solving the same SAT instance by URSA and by the underlying solver would not necessarily take the same time as the solving process can take different routes.

[9]ArgoSAT (`http://argo.matf.bg.ac.rs/downloads.html`) is an open-source, flexible, and verified SAT solver [45, 44].

[10]CLASP (`http://www.cs.uni-potsdam.de/clasp/`) is a solver for (extended) normal logic programs [27]. It combines high-level modelling capacities of ASP with state-of-the-art techniques from the area of SAT solving and it can be used as an ASP solver or a SAT solver. CLASP was a winner at the ASP Competition 2009 in several categories (`http://www.cs.kuleuven.be/~dtai/events/ASP-competition/Results.shtml`) and a winner at the SAT Competition 2009 (`http://www.satcompetition.org/`) in categories crafted SAT +UNSAT and Crafted SAT.

[11]In DIMACS form, the number of variables N and the number of clauses L are given first, and are followed by the list of clauses. The variables are represented by natural numbers (from 1 to N) and their negations are represented by corresponding negative numbers (from -1 to $-N$). More details about the DIMACS format can be found online: `http://www.satlib.org/Benchmarks/SAT/satformat.ps`.

[12]If the solver does not have a feature for generating all satisfying valuations, a simple approach with *blocking clauses* and successive calls to the solver can be used (a blocking clause, added to the set of initial clauses, consists of all literals defining one model negated — this way, a model once generated will not be repeated.

variable — from the minimal (maximal) element in the range onwards, seeking for a minimal (maximal) value that meets the constraint.

6.4. **Turning SAT Models into Values of the Independent Variables.** If the SAT solver finds a satisfying valuation for the input formula, that valuation is used for computing values of independent URSA variables. For this, only the basic variables are relevant (and not the definitional variables introduced during transformation to CNF). Each satisfying valuation determines a vector of fixed Boolean values that correspond to an independent variable. The numerical (and Boolean) values are trivially computed from such representations and returned by the URSA system.

7. Pragmatics of URSA Language

Representation of symbolic values natively used in URSA corresponds to binary representation of unsigned integers, but the specification language is expressive enough and leaves enough freedom for modelling problems in substantially different ways and in different encoding styles, leading to simpler or more efficient solutions. For illustrating this, a prototypical CSP example — the queens problem is considered. The problem is to place N chess queens on an $N \times N$ chessboard such that none of them is able to capture any other (following the standard chess queen's moves). For modelling this problem, in the following text we do not use symmetry breaking or other similar additional constraints, but only the basic formulations of the problem.

Since each row (denoted by numbers 0 to $N-1$) of the board can have exactly one queen on it, the problem is, simply reformulated, to determine, for each row, a column for one queen to be placed. Hence, a solution to the problem is a sequence r_0, r_1, …, r_{N-1} that meet the given constraints. If such sequence is given in advance, it can be simply checked whether it is indeed a solution of the problem. The next URSA program, followed by a fragment of the output, specifies the problem (for the problem size 8; in this and other URSA specifications that follow, the dimension of an instance can be trivially changed in one line).

```
/* *****  queens-1 ***** */
nDim=8;
bDomain = true;
bNoCapture = true;

for(ni=0; ni<nDim; ni++) {
  bDomain &&= (n[ni]<nDim);
  for(nj=ni+1; nj<nDim; nj++)
    bNoCapture &&= n[ni]!=n[nj] && ni+n[nj]!=nj+n[ni] && ni+n[ni]!=nj+n[nj];
}
assert_all(bDomain && bNoCapture);
```

```
************************************
*********  URSA Interpreter *********
************************************

--> Solution 1
n[0]=0
n[1]=6
```

```
n[2]=3
n[3]=5
n[4]=7
n[5]=1
n[6]=4
n[7]=2

...

--> Solution 92
n[0]=3
n[1]=1
n[2]=4
n[3]=7
n[4]=5
n[5]=0
n[6]=2
n[7]=6

[Formula generation: 0s; conversion to CNF: 0.01s; total: 0.01s]
[Solving time: 0.08s]
[Formula size: 841 variables, 3352 clauses]
```

In the above specification (referred to as to `queens-1` in the following text), i-th row ($i = 0, \ldots, N-1$) is associated with the i-th element the array `n`. In each row there should be one queen and `n[i]` is equal to the column in which that queen is placed. The variable `bDomain` encodes the condition that each `n[i]` is between 0 and $N-1$ and the variable `bNoCapture` encodes the condition that there are no two queens that attach each other. If the numbers are represented by vectors of length l, for each problem instance there are lN basic variables in the propositional formula generated. For $N < 16$, the numbers (and intermediate results such as `ni+n[nj]`) in the above specification can be represented by 5 bits. Table 2 shows experimental results for the above specification for the instance sizes N = 1, 2, 3, ..., 12 (instances for which all solutions were found within 600s), including the number of solutions, the number of variables and clauses in the generated propositional formula, and the time spent[13] for finding all solutions. The time spent for finding the first solution was less than $0.05s$ for each problem instance. The time spent for generating the formulae was less than $0.01s$ for each problem instance and was negligible compared to the solving time. This shows that the used generation mechanism is rather efficient. For these problems (as well as for many other CSP problems), the ratio of the number of clauses and the number of variables in the generated formulae, gets rather stable (as the size of the instance increases) and it reflects the problem *constrainedness* [29].

dimension	1	2	3	4	5	6	7	8	9	10	11	12
number of solutions	1	0	0	2	10	4	40	92	352	724	2680	14200
number of variables	5	44	115	209	331	480	667	841	1052	1286	1560	1819
number of clauses	5	149	418	794	1274	1869	2612	3352	4217	5179	6295	7390
all solutions	0	0	0	0	0.01	0.02	0.14	0.08	0.61	3.12	19.25	116.78

Table 2: Experimental results for the `queens-1` specification applied for $N = 1, \ldots, 12$

[13] Reported experimental results were obtained on a PC computer with Intel Celeron 420 1.60Ghz, 1Gb RAM, for URSA using CLASP as the underlying SAT solver.

The above data are obtained using numbers represented by vectors of length 5. However, using the minimal vectors length is not critical — if a larger vector size is used, only trivial constraints are added and they have a very small impact on the solving process. Table 3 shows results for the above specification of the queens problem for instance size 10, for vectors length increasing from 5 to 12. The time for generating formulae slightly increases, the number of variables and clauses increases significantly, but the time spent for solving remains about the same for all vector lengths.

number of bits	5	6	7	8	9	10	11	12
variables	1286	1611	1936	2261	2586	2911	3236	3561
clauses	5179	6604	8119	9724	11419	13204	15079	17044
generating	0.02	0.02	0.02	0.03	0.03	0.04	0.04	0.04
solving	3.12	3.26	3.38	3.19	3.37	3.13	3.22	3.23

Table 3: Experimental results for the `queens-1` specification applied (for finding all solution) for $N = 10$ and for number representations using $5, 6, \dots, 12$ propositional formulae

The same problem can be represented in URSA in other ways as well. For instance, the above specification can be slightly modified so it uses bit-wise operators, instead of arithmetic operators. Each row of the table can be represented as a N-digits number, i.e., by an element of an array `n`. Each of these numbers should have exactly one 1 in its binary representations (which can be checked in the way described in Example 8.1), and at each position exactly one of `n[i]` should have 1. The remaining (diagonal) no-attack conditions (for pairs of positions represented by indexes (`Ax`,`Ay`) and (`Bx`,`By`)) can be also expressed by using bit-wise operations. A corresponding URSA specification (referred to as to `queens-2` in the following text) is as follows (it should be used with numbers represented as vectors of length N):

```
/* *****  queens-2 ***** */
nDim = 8;

bHorizontal = true;
for(ni=0; ni<nDim; ni++)
  bHorizontal &&= ((n[ni] & n[ni]-1)==0) && (n[ni]!=0);

nVertical = 0;
for(ni=0; ni<nDim; ni++)
  nVertical |= n[ni];
bVertical = (nVertical+1 == 0);

bDiagonal = true;
for(nAi=0; nAi<nDim-1; nAi++)
  for(nAj=0; nAj<nDim; nAj++)
    for(nBi=nAi+1; nBi<nDim; nBi++)
      for(nBj=0; nBj<nDim; nBj++)
        if (nBi-nAi==nBj-nAj || nBi-nAi==nAj-nBj)
          bDiagonal &&= (((n[nBj]<<(nBi-nAi)) & n[nAj])==0);

assert_all(bHorizontal && bVertical && bDiagonal);
```

The queens problem can be represented also in the spirit of the *direct encoding*. Namely, each position in the table is associated with one Boolean variable, so `b[i][j]` is set if and only if the position (i, j) is occupied by a queen. It has to be ensured that in each row there is exactly one queen and it has to be ensured that in each column there is at least one queen (this is sufficient if the former condition is satisfied). No-attack conditions are expressed in a straightforward manner. A corresponding URSA specification is as follows (the numerical values could be represented by any vector length that can accommodate `nDim`):

```
/* *****  queens-3 ***** */
nDim = 8;

bHorizontal = true;
for(ni=0; ni<nDim; ni++) {
  bOne = false;
  bMoreThanOne = false;
  for(nj=0; nj<nDim; nj++)  {
    bMoreThanOne ||= bOne && b[ni][nj];
    bOne ||= b[ni][nj];
  }
  bHorizontal &&= bOne && !bMoreThanOne;
}

bVertical = true;
for(ni=0; ni<nDim; ni++) {
  bOne = false;
  for(nj=0; nj<nDim; nj++)
    bOne ||= b[nj][ni];
  bVertical &&= bOne;
}

bDiagonal = true;
for(nAi=0; nAi<nDim-1; nAi++)
  for(nAj=0; nAj<nDim; nAj++)
    for(nBi=nAi+1; nBi<nDim; nBi++)
      for(nBj=0; nBj<nDim; nBj++)
        if (nBi-nAi==nBj-nAj || nBi-nAi==nAj-nBj)
          bDiagonal &&= (!b[nAi][nAj] || !b[nBi][nBj]);

assert_all(bHorizontal && bVertical && bDiagonal);
```

It seems that the four-fold loop in the above specification can be a source of inefficiency. Indeed, instead of going through all possible values for `nBi` and only then checking if the corresponding positions should be tested for attack condition, one can calculate and consider only relevant coordinates `nBj`, as in the following modified specification:

```
/* *****  queens-4 ***** */
nDim = 8;

bHorizontal = true;
for(nx=0; nx<nDim; nx++) {
  bOne = false;
  bMoreThanOne = false;
  for(ny=0; ny<nDim; ny++)  {
    bMoreThanOne ||= bOne && b[nx][ny];
    bOne ||= b[nx][ny];
  }
  bHorizontal &&= bOne && !bMoreThanOne;
}
```

```
bVertical = true;
for(ny=0; ny<nDim; ny++) {
  bOne = false;
  for(nx=0; nx<nDim; nx++)
    bOne ||= b[nx][ny];
  bVertical &&= bOne;
}

bDiagonal = true;
for(nAx=0; nAx<nDim-1; nAx++)
  for(nBx=nAx+1; nBx<nDim; nBx++)  {
    for(nAy=0; nBx-nAx+nAy<nDim; nAy++)  {
      nBy=nBx-nAx+nAy;
      bDiagonal &&= (!b[nAx][nAy] || !b[nBx][nBy]);
    }
    for(nAy=nBx-nAx; nAy<nDim; nAy++)  {
      nBy=nAy-(nBx-nAx);
      bDiagonal &&= (!b[nAx][nAy] || !b[nBx][nBy]);
    }
  }

assert_all(bHorizontal && bVertical && bDiagonal);
```

However, this modification does not improve efficiency of the solving process. Namely, all constraints that are generated by the former and not by the later specification are trivially discarded and the final generated formulae are identical in the two cases. The formula generation is, still, somewhat more efficient in the latter case, but that gain gets insignificant as the problem instance grow.

Table 4 shows experimental results for the four given representations of the queens problem. For the first representation numbers are represented by vectors of length 5, for the second, numbers are represented by vectors of length equal to the instance size, and for the last two, numbers are represented by vectors of length 4. The table shows times (in seconds) for finding all solutions, while the timeout was set at 600s. The first problem representation was clearly shown to be the least efficient, while modified version (the second representation) was the best. As expected, the third and the fourth specification were almost equal. According to the data, it appears that the second specification was most efficient. However, dealing with output of large number of variables takes a larger portion of time for the last two specifications than for the second one. If used in a “quiet” mode (without listing values of independent variables), the last two specifications are slightly more efficient than the second one. In summary, the specifications based on the bitvector operators and on the direct encoding were more efficient. Of course, even a conclusion about the most efficient sort of encoding for the queens problem should rely on a wider set of specifications and experiments, not to mention considering most efficient sorts of encodings generally. Although some encodings typically perform better than some other ones, a full picture of relative quality of encodings is very complex and each popular encoding has problems where it performs well. Hence, one of the points of presenting different specifications of the queens problem is not to locate the best encodings, but rather to demonstrate that in the URSA system one can make significantly different representations of the same problem.

Dimension	solutions	queens-1	queens-2	queens-3	queens-4
8	92	0.08	0.08	0.09	0.03
9	352	0.61	0.16	0.15	0.09
10	724	3.12	0.37	0.32	0.27
11	2680	19.25	1.18	1.17	0.92
12	14200	116.78	4.92	5.52	5.15
13	73712	–	26.47	30.85	30.48
14	365596	–	164.19	198.06	190.45

Table 4: Experimental results of URSA applied on the N queens problem for $N = 1, \dots, 14$ (for finding all solutions) and for four different specifications

8. Comparison to Related Techniques, Languages and Tools

In this section we discuss tools and techniques related to the presented approach. We comment on symbolic execution, on constraint solvers, and on reducing problems to SAT. Finally, we present results of a limited experimental comparison between URSA and several other systems.

8.1. Symbolic execution.

Operation of URSA is related to symbolic execution. In symbolic execution [40, 57], program inputs are represented by symbolic values rather than by concrete data and the values of program variables are represented as symbolic expressions. The program is executed by manipulating expressions involving the symbolic values and, as a result, the output values are expressed as a function of the input symbolic values. Symbolic execution has been proposed for software verification over three decades ago, and recently it gained a renewed interest. The verification tools using symbolic execution include systems like Java Pathfinder[14] [56], Pex[15] [68], Vigilante[16] [16]. Some of these tools use SAT and SMT solvers, but they typically handle only machine data-types (and not arbitrary bit-widths). Also, their purpose is generating test suites and finding (single) models that lead to bugs (rather than enumerating all solutions of combinatorial problems).

8.2. Modelling Languages and Systems.

General modelling systems are used for specifying problems in corresponding modelling languages and solving them by various techniques. There are several dominating approaches for constraint programming including: constraint logic programming over finite domains (CLP(FD); combines two declarative programming paradigms – logic programming and constraint solving) [37], answer set programming (ASP; a form of declarative programming with the roots in nonmonotonic reasoning, deductive databases and logic programming with some ASP systems using SAT solvers) [30], and disjunctive logic programming [42]. There are hybrid systems that use custom specification languages and provide support for constraint programming (e.g., IBM ILOG OPL, COMET, G12). Also, there are libraries for constraint programming and combinatorial optimisation for general purpose programming languages: Ilog solver[17] is a C++ library for

[14]http://babelfish.arc.nasa.gov/trac/jpf

[15]http://research.microsoft.com/en-us/projects/pex/

[16]http://research.microsoft.com/en-us/projects/vigilante/

[17]http://ilog.com/products/

constraint programming, Numberjack[18] is a Python-based constraint satisfaction and optimisation library with support for several underlying combinatorial solvers, $Scala^{Z3}$ is a SCALA library for checking satisfiability and solution enumeration with support for the SMT solver Z3 [41].

Programs in specification languages used by the modelling systems are generally not directly executed. Rather, they describe a problem at a high-level, descriptive way and the specification does not say how the problem is to be solved. In most of modelling languages all solving aspects are ignored (are stored only in the underlying solver). Modelling systems typically use custom, different and incompatible modelling languages. There is no standard modelling language for constraint programming problems. A language XCSP 2.1 is an XML-based format to represent various CSP instances [61]. The main objective of this language is to ease the effort required to test and compare different algorithms by providing a common test-bed of constraint satisfaction instances. XCSP is already used in CSP competitions as an standard input format.[19] A high-lever language MINIZINC (a subset of a language ZINC [47]) also aims at becoming a standard specification language [54]. MINIZINC models can be translated to FLATZINC, a low-level solver input language. There are a number of differences between the languages XCSP, MINIZINC and URSA. XCSP representations are low-level, while MINIZINC and URSA representations are high-level — the former is rather a machine-oriented, interchange format, while the latter two provide high-level, human-readable specifications. In contrast to MINIZINC and URSA, XCSP has no arrays and looping constructs and in XCSP, for each instance, domains, variables, relations, predicates and constraints are exhaustively listed.

All specification languages used in the above systems are based on some form of declarative/logic programming, while URSA uses a novel combination of declarative and imperative paradigms. For some problems, URSA specifications may be longer than of other systems, but some problems naturally expressed in URSA (for instance, problems that involve bit-wise operations, arithmetic modulo 2^n, or software verification problems that involve destructive assignments) are very difficult (or practically impossible for large scale instances) to express in other specification languages. Loop constructs are naturally expressed in URSA, while their absence in some declarative languages (some specification languages do admit complex looping structures) may cause a range of difficulties [62].[20] The URSA language is expressive enough to enable substantially different encodings of the same problem, which is often not easy with other systems. Learning the URSA language should be trivial to someone familiar with some widely used imperative programming language such as C or Java, as there are no specific commands or flow-controls aimed at constraint solving. Concerning the solving mechanism, URSA, like other systems, builds up a formula representing a constraint and then posts the constraint to an underlying solver. URSA's specific is that building this formula is completely specified by URSA's simple semantics, corresponding to binary representation of numbers. Concerning underlying solvers, URSA uses SAT solvers and differs from the above described tools that use mathematical programming, constraint logic programming, and other techniques (with exception of ASP systems such as CMODELS that also uses SAT solvers).

[18]`http://4c110.ucc.ie/numberjack/`

[19]`http://cpai.ucc.ie/09/`

[20]There are recent ideas for introducing loops in constraint programming in an imperative language style in order to enable prototyping new constraints with less effort [19].

Example 8.1 illustrates one family of problems that can be simply solved by the URSA system. Specifications of functions in C (analyzed if they are equivalent) can be almost in verbatim used within URSA, while it would be extremely difficult to represent this kind of problems in existing declarative programming languages — consider, for instance, complex functions (say, like cryptographic functions) that involve a large number of destructive assignments in their specifications. Additional problem is that the bitwise operators are not supported in other modelling systems.

Example 8.1. One of the common problems in software verification is ensuring that two implementations are equivalent. The URSA system can be suitable for such tasks. Consider, for example, two implementations (both based on C-style bitwise operators) of the check that the input number `n` has in its representation exactly one bit set: the first (encoded by `b1`) is rather compact, while the second (encoded by `b2`) is more elaborated but simpler. URSA (for the vector length for representation of numbers equal `nLen`), verifies that the condition `b1^^b2` is unsatisfiable, hence the two implementations give the same result for any input number.

```
b1 = (nv!=0) && ((nv & (nv-1))==0);

nLen=8;
bOne = false;
bMoreThanOne = false;
for(ni=0; ni<nLen; ni++)  {
  bMoreThanOne ||= bOne && ((nv & 1)!=0);
  bOne ||= ((nv & 1)!=0);
  nv >>= 1;
}
b2 = bOne && !bMoreThanOne;

assert(b1^^b2);
```

8.3. **Reduction to SAT and SAT-Based Constraint Solvers.** There are a huge number of problems solved by reduction to SAT, in a range of domains (e.g., in scheduling [74], termination analysis [25], cryptanalysis [48, 50], model checking [12], to name just a few). One of the reasons for such a wide application range are tremendous advances in SAT technology over the last years and a number of efficient solvers. The URSA system does not introduce or promote one SAT solver. Rather, URSA can use any SAT solver in the solving phase. Moreover, it is fruitful to have, within URSA, a number of different SAT solvers, appropriate for different sorts of input problems, following ideas of SAT portfolio solvers [55].

There are several approaches for encoding CSP problems into SAT [60]. Probably the most popular basic types of encodings into SAT are: the *sparse encoding*, the *compact encoding*, and the *order encoding* [34, 66]. In the sparse encoding, a propositional variable $x_{v,i}$ is defined as true iff the integer variable v has a value i assigned to it. Examples of the sparse encoding are the *direct encoding* and the *support encoding* [60]. In the compact encoding (or *log encoding*), a propositional variable is assigned to each bit of each integer variable (within a finite domain) [60]. In the order encoding (also known as *regular encoding*), related to many-valued logics and often used for the finite-domain linear problems, an inequality $v \leq i$ is encoded by a different propositional variable for each integer variable v and integer value i [66, 1]. Even within one encoding style, modelling of a problem can take significantly different forms [63]. There are a number of both theoretical and practical studies and comparisons between different encoding schemes. Since the log encoding lacks the propagation power of the direct and support encodings, it typically leads to less efficient solutions, compared to these two [72, 28]. The order encoding gives better performance compared with the direct encoding and the support encoding for some CSP problems [66]. Within the URSA system, numerical variables are represented using binary representation that corresponds to the compact encoding, but still, thanks to the expressiveness of the specification language, other encoding styles can be simulated and used (as shown by the presented examples).

The URSA system is related to the special-purpose system for transforming cryptanalysis of hash functions into the SAT problem [39]. In that approach, implementations of hash functions in C++ were used and, by overloading the standard arithmetic and logical operators in C++ and by running the code of the hash functions within such framework, propositional formulae corresponding to cryptanalysis tasks were generated (and then solved by a SAT solver). Similar approach was also used for cryptanalysis of DES [71]. URSA is a general framework aimed not only to cryptanalysis tasks, but to a much wider range of problems. The framework consists of both a modelling language (instead of C++) and the solving machinery, tightly integrated. The system is stand-alone, does not involve the C++ language in the modelling process, and the language itself defines the modelling power of the approach. The related, C++ based approach, has an advantage that the specification can be used directly as a C++ code, but URSA specifications can often be used as C code in verbatim.

There are several general-purpose constraint solving systems based on SAT. Thanks to the advances in SAT technology, such systems can be very efficient, despite the main weakness of this approach: a domain knowledge and a global structure of the problem are lost when it is reduced to the simple propositional logic. SUGAR[21] is a SAT-based constraint solver that uses the order encoding [66]. It is focused on compiling finite linear CSP into SAT. SUGAR uses a syntax of CSP that is designed to cover the notation of the XCSP 2.1 format. FznTini[22] is a general constraint solver that solves constraint satisfaction and optimization problems (not involving floating point numbers) given in the general constraint language FlatZinc (typically produced from MiniZinc specifications) by translating them to SAT and calling a SAT solver Tinisat [35]. The system can be also used for translating to SAT only (so, it can be used by independent SAT solvers). FznTini uses a fixed encoding directly related to binary representation of integers (the two's complement representation) that corresponds to the compact encoding. NPSpec is a modelling language for constraint problems, going with a tool Spec2Sat that compiles specifications into SAT instances [11]. NPSpec uses a highly declarative style of programming, similar to Datalog (a query language for deductive databases). The semantics of NPSpec is based on the model minimality, an extension of the least fixed point semantics of the Horn fragment of first order logic [11]. The system MXG[23], focused on NP-hard problems, uses a modelling language based on classical first order logic, and for a given specification produces a propositional formula (and passes it to a SAT solver) [51, 58]. The system can also translate problem specification to SAT extended with cardinality constraints.

[21]`http://bach.istc.kobe-u.ac.jp/sugar/`

[22]`http://users.cecs.anu.edu.au/~jinbo/fzntini/`

[23]`http://www.cs.sfu.ca/research/groups/mxp/`

Apart from systems that can translate problem specifications to SAT, there are also systems that translate such specifications to SMT. For instance, FZN2SMT [5] translates from the FlatZinc language and the system SIMPLY [4] translates from a declarative modelling language similar to, but simpler than MiniZinc. Both systems translate to the standard SMT-LIB format[24] and can use various underlying SMT theories and available SMT solvers.

The system URSA and the above systems share the underlying solving technology, but their specification languages are very different. In contrast to URSA, all of the above tools use declarative languages, and they don't have features of imperative languages (e.g., destructive assignments), as discussed in Section 8.2. Also, the languages used by the above tools provide support for various operators and global constraints, but typically do not support bit-wise operators and constraints involving modular arithmetic, which can be essential in many cases. This way, the input languages restrict the tools from having the full power of modelling in propositional logic. Concerning expressiveness, SUGAR and FznTini use rather simple, low-level specification language without flow control structures and other features of programming languages that URSA has. Using representation based on binary representation of integers is very similar in FznTini and URSA (with a minor difference that FznTini uses the two's complement representation).

8.4. **Experimental Comparison.** It is very difficult to make a fair and thorough comparison of the modelling systems: they were not built with the same motivation and purposes; it is not only performance that is important but also expressiveness and ease of acquiring a modelling language; some systems perform better on some sorts of problems (and worse on the other); a single problem can be often specified in different ways; most systems do not share the same input language (even if they do, some types of specifications may be better suited to some systems); the leading systems are under ongoing developments, and there are new features being added constantly; there are new emerging systems, etc. Still, with all of the above cautious, several (a very few, likely due to the above difficulties) existing reports give some general picture of efficiency of systems for constraint solving over finite domains [24, 43, 21] while new insights are provided by a number of system competitions recently initiated.

For experimental comparison with the URSA system, we used the following state-of-the-art systems (and versions), including several industrial ones: constraint logic programming systems B-Prolog 7.4[25] and SICStus 4.1.1[26] (with the `clpdf` module), a deductive database system (using Disjunctive Datalog language that combines databases and logic programming) DLV[27] [42], ASP solvers (used with Smodels specifications and a grounder Lparse 1.1.2) CLASP 1.3.2[28] [27], Cmodels 3.79[29] [31] and Smodels 2.34[30] [65], hybrid optimization system (using a custom object-oriented programming language) COMET 2.1.1[31] [49], a hybrid optimization system (using a custom declarative modelling language) IBM

[24]`http://www.smtlib.org/`

[25]`http://www.probp.com/`

[26]`http://www.sics.se/isl/sicstuswww/site/`

[27]`http://www.dbai.tuwien.ac.at/proj/dlv`

[28]`http://www.cs.uni-potsdam.de/clasp/`

[29]`http://www.cs.utexas.edu/~tag/cmodels/`

[30]`http://www.tcs.hut.fi/Software/smodels/`

[31]`http://dynadec.com`

ILOG OPL 6.3[32], a default finite-domain G12/FD solver Mercury (using MiniZinc as its input language) from MiniZinc 1.1.1[33] [64], and SAT-based systems SUGAR 1.14.6 and FznTini.[34] The URSA system was used with CLASP 1.2.0 as an underlying SAT solver.

We performed experimental comparison between all of the above tools on a prototypical CSP problem — the queens problem (that involves different sorts of constraints), and additional comparison between the systems that performed the best in the first phase. Most of the used specifications are given as part of the system distributions; they are typically straightforward for their systems and symmetry breaking conditions or some other additional constraints were not used. Experiments were performed on a PC computer with Intel Celeron 420 1.60Ghz, 1Gb RAM and the time threshold for finding all solutions was 600s. The measured times are reported by the systems themselves or by adding the spent "user" and "system" time.[35]

The Queens problem. Table 5 shows results of experiments for the queens problem for instance sizes from 8 to 14. The URSA system was used with the specification `queens-2` (see page 19). The most efficient system on this benchmark was B-Prolog, followed by SICStus, and then by URSA, that had similar performance as G12/FD and CLASP. It is interesting to notice that the URSA system using CLASP as an underlying SAT solver was around the same efficiency as the CLASP solver used as ASP solver, which suggests that the reduction to SAT used by URSA is very efficient.

N	solutions	Cmodels	Smodels	DLV	OPL	COMET	CLASP	G12/FD	URSA	SICStus	B-PROLOG
8	92	0.11	0.09	0.04	0.08	0.02	0.02	0.03	0.08	0.01	0.01
9	352	0.75	0.35	0.15	0.30	0.09	0.07	0.08	0.16	0.03	0.01
10	724	7.42	1.41	0.45	0.63	0.34	0.22	0.22	0.37	0.09	0.02
11	2680	132.20	10.51	2.31	2.28	1.64	0.89	1.12	1.18	0.50	0.10
12	14200	>600	44.37	12.67	12.56	9.34	4.51	5.91	4.92	2.53	0.45
13	73712	>600	331.54	83.36	62.89	48.73	25.50	31.48	26.47	14.36	2.85
14	365596	>600	>600	479.91	301.88	246.90	190.48	188.43	164.19	76.89	17.64

Table 5: Results of experimental comparison of ten tools (including URSA) applied on the N queens problem for $N = 8, \ldots, 14$

The above result don't include the systems that translate problem specifications to SAT and which are the systems closest to URSA. Namely, these systems translate inputs to SAT (so it can be considered that they share the solving mechanism), but they use different SAT solvers. A fair comparison would be thus to use these systems only as translators to SAT and then use the same SAT solver (for instance, CLASP) for finding all models of the generated SAT formulae. It is interesting to consider size of generated formulae and solving times (of course, smaller formulae does not necessarily lead to shorter solving times). FznTini was used with FlatZinc specifications obtained from the MiniZinc specification used by G12/FD (with

[32]`http://www-01.ibm.com/software`

[33]`www.g12.cs.mu.oz.au/minizinc/`

[34]We didn't include the systems MXG and NPSpec/Spec2Sat in this evaluation: MXG was not publicly available and NPSpec/Spec2Sat was not maintained since 2005, and also its reported performance [10], especially for the SAT formulae generation phase, is significantly poorer than of URSA and other considered systems.

[35]For some recommendations on how to benchmark constraint solving systems visit `http://www.dbai.tuwien.ac.at/proj/dlv/bench/`.

integers encoded with 5 bits) and with FlatZinc specifications obtained from a MiniZinc specification made in the style of the direct encoding,[36] we will denote them by 1 and 2. sugar was used only with a specification that employs the order encoding. ursa was used with the specifications `queens-1` (with integers encoded with 5 bits), `queens-2` (with the number of bits equal the instance dimension), and `queens-3`, (with integers encoded with 4 bits). Table 6 presents the obtained experimental results. All recorded times were obtained for the "quiet" mode of the sat solver (without printing the models). Times for generating formulae were negligible (compared to the solving phase) for all systems, so we don't report them here.

For related specifications, ursa's `queens-1` gave much smaller formulae (probably thanks to techniques mentioned in Section 6.2) and somewhat better performance than FznTini 1, which suggests that FznTini does not benefit much from information about the global structure of the problem. The formulae generated by sugar were significantly smaller than in the above two cases, and led to much better solving efficiency. However, it was outperformed by the remaining entrants. The ursa's specifications `queens-2` and `queens-3` gave similar results. The specification `queens-3` produced formulae with the smallest number of clauses. FznTini 2 produced formulae with the smallest number of variables. The best results in terms of the solving times were obtained also by FznTini 2. It can be concluded that ursa can produce, with suitable problem specifications, propositional formulae comparable in size and in solving times with formulae produced by related state-of-the-art systems.

Additional Experiments. In additional experiments, only the systems that performed the best on the queens problem were considered (with only one constraint logic programming system kept): B-prolog, clasp, FznTini, g12/fd, and ursa. The following problems were considered (for all problems all solutions were sought):

Golomb Ruler: The problem (actually, one of its variation) is as follows, given a value L check if there are m numbers a_0, a_1, …, a_{m-1} such that $0 = a_0 < a_1 < ... < a_{m-1} = L$ and all the $m(m-1)/2$ differences $a_j - a_i$, $0 \leq i < j \leq m$ are distinct (problem 6 at CSPlib[37]). The experiments were performed for $m = 5, \dots, 11$, with the largest L that make the problem unsatisfiable and with the smallest L that make the problem satisfiable.

Magic Square: A magic square of order N is a $N \times N$ matrix containing the numbers from 0 to $N^2 - 1$, with each row, column and main diagonal equal the same sum. The problem is to find all magic squares of order N (problem 19 at CSPlib). The experiments were performed for $N = 3$ and $N = 4$.

Linear Recurrence Relations: Linear homogeneous recurrence relations of degree k are of the form: $T_{n+k+1} = a_k T_{n+k} + \ldots + a_1 T_{n+1}$ for $n \leq 0$. Given T_1, …, T_k and n, T_n can be simply calculated, but finding explicit formula for T_i requires solving a nonlinear characteristic equation of degree k, which is not always possible. So, the following problem is nontrivial: given T_1, …, T_{k-1} and T_n, find T_k. For the experiment, we used the relation $T_{n+3} = T_{n+2} + T_{n+1} + T_n$, $T_1 = T_2 = 1$. We generated instances with the (only) solution $T_3 = 1$ and the systems were required to seek all possible values for T_3. Additional constraints (used explicitly or implicitly) for all considered systems)

[36]Therefore, these translations to sat are rather by two systems: the MiniZinc to FlatZinc converter and FznTini.

[37]`http://www.csplib.org/`

Dimension	8	9	10	11	12	13	14
FznTini 1:							
variables	3012	3825	4735	5742	6846	8047	9345
clauses	9128	11628	14460	17567	21000	24713	28770
all solutions	0.15	0.79	3.20	14.53	111.78	>600	>600
URSA (queens-1):							
variables	841	1052	1286	1560	1819	2139	2468
clauses	3352	4217	5179	6295	7390	8712	10089
all solutions	0.08	0.53	2.83	17.60	98.04	>600	>600
SUGAR:							
variables	220	284	356	436	524	620	724
clauses	1138	1653	2253	3012	3924	5003	6263
all solutions	0.02	0.06	0.31	1.58	9.59	68.07	411.15
URSA (queens-2):							
variables	542	739	978	1263	1598	1987	2434
clauses	3319	5008	7280	10258	14077	18884	24838
all solutions	0.01	0.04	0.12	0.70	4.01	23.17	138.55
URSA (queens-3):							
variables	176	225	280	341	408	481	560
clauses	800	1110	1490	1947	2488	3120	3850
all solutions	0.01	0.03	0.12	0.69	3.82	21.09	136.45
FznTini 2:							
variables	128	162	200	242	288	338	392
clauses	872	1236	1690	2244	2908	3692	4606
all solutions	0.01	0.02	0.07	0.33	1.52	8.39	52.12

Table 6: Data on SAT formulae generated by FznTini, SUGAR and URSA for the N queens problem for $N = 8, \dots, 14$ and solved by the CLASP SAT solver

were $T_i > 0$ and $T_i \leq T_n$, for $i = 1, \dots, n$. URSA and FznTini were used with 32 bit length for numerical values.

Non-linear recurrence relations: In non-linear homogenous recurrence relations of degree k, the link between T_{n+k} and T_{n+k-1}, T_{n+k-2}, …, T_n, is not necessarily linear. For the experiment we used the relation $T_{n+4} = T_{n+3} \cdot T_{n+2} + T_{n+1}$, $T_1 = T_2 = 1$. We generated instances with the (only) solution $T_3 = 1$ and the systems were required to seek all possible values for T_3. Additional constraints (used explicitly of implicitly) for all considered systems) were $T_i > 0$ and $T_i \leq T_n$, for $i = 1, \dots, n$. For all problem instances, the size of all relevant values T_i were smaller than 2^{32}. URSA and FznTini were used with 32 bit length for numerical values.

The URSA was used with the following specification for the Golomb ruler problem (for the instance $m = 7$, $L = 25$):

```
nM=7;
nL=25;
bRulerEndpoints = num2bool(nRuler & nRuler >> nL & 1);

nMarks=2;
bDistanceDiff=true;
for(ni=1; ni<=nL-1; ni++) {
  nMarks += (nRuler >> ni) & 1;
  n = (nRuler & (nRuler << ni));
  bDistanceDiff &&= (n & (n-1))==0;
}

assert_all(bRulerEndpoints && nMarks==nM && bDistanceDiff);
```

The above specification employs a binary representation of the ruler (`nRuler`) where each bit set denotes a mark. The value `nRuler & nRuler >> nL & 1` equals `1` if and only if the first and `nL`th bit are set (as the ruler endpoints). The value `nMarks` counts the bits set (e.g., the marks) and it should equal `nM`. If the ruler `nRuler` is a Golomb ruler, then whenever it is shifted left (for values `1`, . . ., `nL-1`) and bitwise conjunction is performed with the original ruler giving the value `n`, there will be at most one bit set in `n` (since all the differences between the marks are distinct). There is at most one bit set in `n` if and only if the value `n & (n-1)` equals 0. This specification, employing a single loop, illustrate the expressive power of bitwise operations supported in the URSA language.

For the magic square problem, URSA was used with the following specification:

```
nDim=4;
nN=nDim*nDim;
bCorrectSum = (2*nSum*nDim == nN*(nN-1));

bDomain=true;
bDistinct=true;
for(ni=0;ni<nDim;ni++) {
  for(nj=0;nj<nDim;nj++) {
    bDomain &&= (nT[ni][nj]<nN);
    for(nk=0;nk<nDim;nk++)
      for(nl=0;nl<nDim;nl++)
        bDistinct &&= ((ni==nk && nj==nl) || nT[ni][nj]!=nT[nk][nl]);
  }
}

bSum=true;
nSum1=0;
nSum2=0;
for(ni=0;ni<nDim;ni++) {
  nSum1 += nT[ni][ni];
  nSum2 += nT[ni][nDim-ni-1];
  nSum3=0;
  nSum4=0;
  for(nj=0;nj<nDim;nj++) {
    nSum3 += nT[ni][nj];
    nSum4 += nT[nj][ni];
  }
  bSum &&= (nSum3==nSum);
  bSum &&= (nSum4==nSum);
}
bSum &&= (nSum1==nSum);
bSum &&= (nSum2==nSum);

assert_all(bCorrectSum && bDomain && bDistinct && bSum);
```

For the linear recursive relation, URSA was used with the following specification (the specification for the non-recursive relations is analogous):

```
n  = 30;
ny = 20603361;
```

```
n1=1;
n2=1;
n3=nx;

for(ni=4; ni <= n; ni++) {
  ntmp=n1+n2+n3;
  n1=n2;
  n2=n3;
  n3=ntmp;
  bDomain &&= (n3<=ny);
}

assert_all(bDomain && n3==ny);
```

Table 7 shows experimental results (with translation times included). FznTini was used with CLASP as an underlying solver (the built-in solver gave poorer results). The number of variables and clauses generated by URSA were, in these benchmarks, smaller than for FznTini. For LPARSE/CLASP, the translation time was significant, and some of the poor results for some benchmarks are due to large domains (while the system works with relations rather than functions). For recurrence relations, G12/FD reported model inconsistency when it approached its limit for integers, while B-PROLOG just failed to find a solution for larger instances. Overall, on this set of benchmarks, URSA gave better results than CLASP and FznTini and on some benchmarks outperformed all other tools.

Discussion. The described limited experiments cannot give definite conclusions or ranking of the considered systems, as discussed above. In particular, one may raise the following concerns, that can be confronted with the following arguments:

- URSA *was used with a good problem specifications, and there may be specifications for other systems that lead to better efficiency.* However, almost all specifications were taken from the system distributions, given there to illustrate the modelling and solving power of the systems. Also, the problem specifications used for URSA are also probably not the best possible, but are rather straightforward, as specifications for other systems. In addition, in contrast to URSA, other modelling systems typically aim at liberating the user of thinking of details of internal representations and are free to perform internal reformulations of the input problem. Making specifications in URSA may be somewhat more demanding than for some other systems, but gives to the user a fuller control of problem representation.
- *Some specifications used for* URSA *are related to the direct encoding (known to be efficient, for example, for the queens problem), while this is not the case with other systems.* What is suitable for SAT-based systems is not necessarily suitable for other systems. For instance, G12/FD gives significantly poorer results with the specification of the queens problem based on the direct encoding, than with the one used in the experiment. This is not surprising, because systems that are not based on SAT does not necessarily handle efficiently large numbers of Boolean variables and constraints (in contrast to SAT solvers) and lessons from the SAT world (e.g., that for some sort of problems, some encoding scheme is the most efficient) cannot be *a priori* applied to other solving paradigms.
- URSA *uses bit-wise logical operators, while other systems do not (as they don't have support for them).* Bit-wise logical operators make one of advantages of URSA, while in the same time, some other systems use their good weapons (e.g., global constraints such as all-different).

Dimension	B-prolog	clasp	FznTini	g12/fd	ursa
Golomb ruler					
5/10	0.01	5.36	0.20	0.06	0.01
6/16	0.01	44.68	1.16	0.08	0.02
7/24	0.01	350.11	9.53	0.10	0.10
8/33	0.08	>600	111.90	0.24	0.69
9/43	0.69	>600	>600	1.18	4.89
10/54	5.34	>600	>600	7.84	35.55
11/71	105.54	>600	>600	93.2	571.90
5/11	0.01	6.33	0.32	0.07	0.01
6/17	0.01	57.40	1.43	0.08	0.01
7/25	0.01	429.98	14.15	0.10	0.13
8/34	0.09	>600	106.26	0.26	0.87
9/44	0.78	>600	>600	1.27	5.87
10/55	6.86	>600	>600	6.44	37.28
11/72	115.81	>600	>600	125.50	450.41
Magic square					
3	0.01	0.05	0.04	0.01	0.01
4	4.74	462.21	>600	10.26	93.01
Linear recurrence relations					
4	0.00	0.01	0.01	0.01	0.00
5	0.00	0.06	0.01	0.02	0.00
6	0.00	1.18	0.01	0.02	0.01
7	0.00	25.49	0.01	0.02	0.01
...					
28	43.83	>600	0.17	143.54	0.31
29	84.92	>600	0.68	incons	0.33
30	158.78	>600	1.02	incons	0.47
Non-linear recurrence relations					
4	0.00	0.00	0.01	0.01	0.00
5	0.00	0.01	0.01	0.01	0.02
6	0.00	0.42	0.22	0.02	0.05
7	0.00	126.05	0.36	0.02	0.08
8	0.00	>600	0.51	0.02	0.13
9	fail	>600	0.76	0.02	0.28
10	fail	>600	0.88	0.03	0.53
11	fail	>600	0.97	incons	0.77

Table 7: Results of experimental comparison of five tools

In summary, the presented experiments suggest that the ursa system (although it is not primarily a csp solver but a general system for reducing problems to sat) is competitive to the state-of-the-art, both academic and industrial, modelling systems — even if they can encode high-level structural information about the input problem and even if they involve specialized underlying solvers (such as support for global constraints like all-different). A wider and deeper comparison between these (and some other) constraint solvers (not sharing input language) and with different encodings of considered problems, would give a better overall picture but is out of scope of this paper.

9. Future Work

The current system (with the presented semantics and the corresponding implementation) uses one way (binary representation) for representing (unsigned) integers but (as shown in the given examples), it still enables using different encoding styles in specifications. For further convenience, we are planning to natively support other representations for integers, so the user could choose among several encodings. Also, signed integers and floating point numbers could be supported.

The language URSA (and the interpreter) can be extended by new language constructs (e.g., by division). A new form of the `assert` can be added, such that it propagates intermediate solutions to subsequent commands of the same sort. Support for global constraints can also be developed, but primarily only as a ,,syntactic sugar“ — the user could express global constraints more easily, but internally they would be expanded as if they were expressed using loops (i.e., as in the current version of the system). Alternative forms of support for global constraints would require substantial changes in the SAT-reduction mechanism.

On the lower algorithmic and implementation level, we are planning to further improve the current version of transformation to CNF.

In the current version, ground integers are represented by built-in fixed-precision integers, which is typically sufficient. However, in order to match symbolic integers, ground integers should be represented by arbitrary (but fixed) length integers and we are planning to implement that.

Concerning the underlying SAT solvers, currently only two complete SAT solvers are used. We are planning to integrate additional solvers, since some solvers are better suited to some sorts of input instances, as the SAT competitions show. Within this direction of work, we will also analyze performance of stochastic solvers within URSA. In addition, we are exploring potentials of using non-CNF SAT solvers [67, 53, 38] within URSA, which avoids the need for transformation to CNF [69]. Choosing among available solvers can be automated by using machine learning techniques for analysis of the generated SAT instances (or even input specifications) [73, 55]. For solving optimization problems, instead of the existing naive implementation, we are planning to implement more advanced approaches and to explore the use of MaxSAT and pseudo-Boolean solvers [3].

On the theoretical side, the full operational semantics outlined in this paper can be formally defined and it could be proved that solutions produced by the URSA system indeed meet the specifications and if there are no produced solutions, then the specifications is inconsistent. Along with the formal verification (i.e., verification within a proof assistant) of the SAT solver ArgoSAT used [45, 44], that would provide a formal correctness proof of the URSA system (which would make it, probably, the first *trusted* constraint solver).

In the presented version of URSA, reducing to SAT is tightly integrated (and defined by the semantics of the system) in the program execution phase. An alternative would be as follows: during the program execution phase, a first-order formula is generated and only before the solving phase it is translated to a propositional formula. Moreover, the generated formula would not need to be translated to a propositional formula, but could be tested for satisfiability by using SMT (satisfiability modulo theory) solvers (e.g., for linear arithmetic, equality theory, alldifferent theory etc.) [2]. In particular, symbolic computations employed by the URSA system are closely related to the theory of bit-vector arithmetic and to decision procedures for this theory based on “bit-blasting” [8, 9]. Since solvers for this theory typically cover all the operators used in URSA, the theory of bit-vector arithmetic can be used as an underlying theory (instead of propositional logic) and any solver for bit-vectors arithmetic can be used as a solving engine. Generally, the reduction could be adaptable to SMT solvers available — if some solver is available, then its power can be used, otherwise all generated constraints are exported to propositional logic. This would make the approach more powerful and such development is the subject of our current work — the system URSA MAJOR will be able to reduce constraints not only to SAT but to different SMT theories. That system will be a general constraint solver and also a high-level front-end to the low level SMT-LIB interchange format, and, further, to all SMT solvers that supports it. Reduction to the theory of bit-vector arithmetic is firstly explored in this context [46] and it shows that reducing to bit-vector arithmetic does not necessarily lead to more efficient solving process than reducing to SAT (and the same holds for other SMT theories).

With the increased power of the presented system (by using both SAT and SMT solvers), we are planning to further consider a wide range of combinatorial, NP-complete problems, and potential one-way functions and also to apply the URSA system to real-world problems (e.g., the ones that are already being solved by translating them to SAT). Some applications in synthesis of programs are already the subject of our current work.

For the sake of easier practical usability of the URSA system, we are planning to develop a support for integration of URSA with popular imperative languages (C, C++, Java).

10. Conclusions

In this paper we described a novel approach for uniform representation and solving of a wide class of problems by reducing them to SAT. The approach relies on:

- a novel specification language that combines features of imperative and declarative languages — a problem is specified by a test, expressed in an imperative form, that some values indeed make a solution to the problem;
- symbolic computations over unknowns represented by (finite) vectors of propositional formulae.

The approach is general and elegant, with precisely defined syntax and induced (by the concept of symbolic execution) semantics of the specification language. This enables straightforward implementation of the system and it works as a "clear box". The proposed language is a novel mixture of imperative and declarative paradigms, leading to a new programming paradigm. Thanks to the language's declarative aspects — the problem is described by what makes a solution and not by describing how to find it — using the system does not require human expertise on the specific problem being solved. On the other hand, specifications are written in imperative form and this gives the following advantages compared to other modelling languages (all of them are declarative):

- problem specifications can involve destructive assignments, which is not possible in declarative languages and this can be essential for many sorts of problems (e.g., from software verification);
- modelling problems that naturally involve loops (and nested loops) is simple and the translation is straightforward;
- for users familiar with imperative programming paradigm, it should be trivial to acquire the specification language URSA (since there are no specific commands or flow-controls aimed at constraint solving);

- the user has a fuller control of internal representation of the problem, so can influence the efficiency of the solving phase.
- a specification can be taken, almost as-is, from and to languages such as C (within C, such code would check if some given concrete values are indeed a solution of the problem).
- the system can smoothly extend imperative languages like C/C++ or Java, (as constraint programming extends logic programming).
- the system can be verified to be correct in a straightforward manner.

In addition, the system URSA, in contrast to most of (or all) other modelling systems, supports bit-wise logical operators and constraints involving modular arithmetic (for the base of the form 2^n), which is essential for many applications, and can also enable efficient problem representation and problem solving.

The proposed approach can be used for solving all problems with Boolean and numerical unknowns, over Boolean parameters and numerical parameters with finite domains that can be stated in the specification language (e.g., the domain of the system is precisely determined by expressiveness of its specification language).

The search for required solutions of the given problem is performed by modern SAT solvers that implement very efficient techniques not directly applicable to other domains. While SAT is already used for solving a wide range of various problems, the proposed system makes these reductions much easier and can replace a range of problem-specific tools for translating problems to SAT. The tool URSA can be used not only as a powerful general problem solver (for finite domains), but also as a tool for generating SAT benchmarks (both satisfiable and unsatisfiable).[38]

Concerning weaknesses, URSA is not suitable for problems where knowledge of the domain and problem structure are critical and can be efficiently tackled only by specialized solvers, and this holds for reduction to SAT generally. Due to its nature, by interfacing URSA with standard specification languages like XCSP or MiniZinc, most of the URSA modelling features and power would be lost (e.g., bit-wise logical operators and destructive updates), while global constraints supported by these languages would be translated in an inefficient way. Therefore, it makes no much sense to enable conversion from these standard languages to URSA and this makes URSA a bit isolated system in the world of constraint solvers or related systems.

In this paper we do not propose:

- a new SAT-encoding technique — rather, the URSA specification language can be used for different encoding styles;
- a technique for transforming a SAT formula to conjunctive normal form — this step is not a part of the core of the URSA language and is not covered by its semantics, so any approach (meeting the simple specification) can be used; the current technique seem to work well in practice, while it can still be a subject of improvements.
- a SAT solver — rather, URSA can use any SAT solver (that can generate all models for satisfiable input formulae); moreover, having several SAT solver would be beneficial, since some solvers are better suited to some sorts of problems.

In this paper we also described the system URSA that implements the proposed approach and provided some experimental results and comparison with related systems. They

[38]A large number of SAT instances obtained from cryptanalysis of DES [71] (by using simpler version of the system proposed in this paper), were part of the SAT Competition corpus (http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf).

suggest that, although URSA is not primarily a CSP solver (but a general system for reducing problems to SAT), the system is, concerning efficiency, competitive to state-of-the-art academic and industrial CSP tools. URSA is also competitive to other system that translate problem specifications to SAT. In contrast to most of other constraint solvers, the system URSA is open-source and publicly available on the Internet.

For future work, we are planning to extend the system so it can use not only complete SAT solvers, but also stochastic SAT solvers, non-CNF solvers, and SMT solvers. We will also work on formal (machine-checkable by a proof assistant) verification of the system (i.e., show that the URSA system always gives correct results) and on extensions of the system relevant for practical applications.

Acknowledgement

This work was partially supported by the Serbian Ministry of Science grant 174021 and by SNF grant SCOPES IZ73Z0_127979/1.

The author is grateful to Dejan Jovanović whose ideas on overloading C++ operators influenced development of the system presented in this paper; to Filip Marić for valuable discussions on the presented system and for using portions of his code for shared expressions; to Milan Šešum for portions of his code for performing operations over vectors of propositional formulae; to Ralph-Johan Back and to the anonymous reviewers for very detailed and useful comments on a previous version of this paper.

References

[1] Josep Argelich, Alba Cabiscol, Inês Lynce, and Felip Manyà. Regular encodings from MAX-CSP into partial MAX-SAT. In *ISMVL 2009, 39th International Symposium on Multiple-Valued Logic*, pages 196–202. IEEE Computer Society, 2009.

[2] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 75–97 [3], 2009.

[3] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009.

[4] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *Eighth International Workshop on Constraint Modelling and Reformulation – ModRef 2009*, pages 30–44, 2009.

[5] Miquel Bofill, Josep Suy, and Mateu Villaret. A system for solving constraint satisfaction problems with SMT. In *Theory and Applications of Satisfiability Testing - SAT 2010*, LNCS 6175, pages 300–305. Springer, 2010.

[6] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006.

[7] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 69–76. IEEE, 2009.

[8] Raik Brinkmann and Rolf Drechsler. Rtl-datapath verification using integer linear programming. In *Proceedings of the VLSI Design 2002*, pages 741–746. IEEE Computer Society, 2002.

[9] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):95–104, 2009.

[10] Marco Cadoli, Toni Mancini, and Fabio Patrizi. SAT as an Effective Solving Technology for Constraint Problems. In *ISMIS*, LNCS 4203, pages 540–549. Springer, 2006.

[11] Marco Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1-2):89–120, 2005.

[12] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[13] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS 2004*, pages 168–176. LNCS 2988, Springer, 2004.

[14] Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM Press, 1971.

[16] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony I. T. Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Transactions on Computer Systems*, 26(4), 2008.

[17] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[18] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of Association for Computing Machinery*, 7(3):201–215, 1960.

[19] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. An abstract interpretation based combinator for modelling while loops in constraint programming. In *CP'07: Proceedings of the 13th international conference on Principles and practice of constraint programming*, LNCS 4741. Springer, 2007.

[20] Dušan Djuki¸ Vladimir Janković, Ivan Matić, and Nikola Petrović. *The IMO Compendium 1959-2004*. Springer, 2005.

[21] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Jorunal of Experimental and Theoretical Artificial Intelligence*, 21(2):79–121, 2009.

[22] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, LNCS 2919, pages 502–518. Springer, 2004.

[23] Uwe Egly. On different structure-preserving translations to normal form. *Journal of Symbolic Computation*, 22(2):121–142, 1996.

[24] Antonio J. Fernández and Patricia M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 5(3):275–301, 2000.

[25] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Sat solving for termination analysis with polynomial interpretations. In *Theory and Applications of Satisfiability Testing - SAT 2007*, LNCS 4501, pages 340–354. Springer, 2007.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.

[27] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, LNCS 4483, pages 260–265. Springer, 2007.

[28] Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

[29] I.P. Gent, E. Macintyre., P. Prosser, and T. Walsh. The constraidness of search. In *Proceedings of AAAI-96*, pages 246–252, Menlo Park, AAAI Press/MIT Press., 1996.

[30] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-Based Answer Set Programming. In *The Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pages 61–66. AAAI Press / The MIT Press, 2004.

[31] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[32] John Harrison. Meta theory and reflection in theorem proving: a survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.

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

[34] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, pages 296–303. Morgan Kaufmann, 1999.

[35] Jinbo Huang. Universal booleanization of constraint models. In *Principles and Practice of Constraint Programming – CP 2008*, LNCS 5202, pages 144–158. Springer, 2008.

[36] ILOG-S.A. Ilog solver 6.0: Reference manual. 2003.

[37] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[38] Himanshu Jain and Edmund M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In *Design Automation Conference*, pages 563–568. ACM, 2009.

[39] Dejan Jovanović and Predrag Janičić. Logical analysis of hash functions. In *Frontiers of Combining Systems (FroCoS)*, LNCS 3717, pages 200–215. Springer, 2005.

[40] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[41] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of z3: Integrating smt and programming. In *23rd International Conference on Automated Deduction – CADE-23*, LNCS 6803, pages 400–406. Springer, 2011.

[42] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[43] Toni Mancini, Davide Micaletto, Fabio Patrizi, and Marco Cadoli. Evaluating ASP and Commercial Solvers on the CSPLib. *Constraints*, 13(4):407–436, 2008.

[44] Filip Marić. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.

[45] Filip Marić and Predrag Janičić. Formal Correctness Proof for DPLL Procedure. *Informatica*, 21(1):57–78, 2009.

[46] Filip Marić and Predrag Janičić. Urbiva: Uniform reduction to bit-vector arithmetic. In *Automated Reasoning (IJCAR 2010)*, LNCS 6173, pages 346–352. Springer, 2010.

[47] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13(3), 2008.

[48] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, 2000.

[49] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming - CP 2005*, LNCS 3709, pages 881–881. Springer, 2005.

[50] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *Theory and Applications of Satisfiability Testing - SAT 2006*, LNCS 4121, pages 102–115. Springer, 2006.

[51] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving np search problems. In *National Conference on Artificial Intelligence – AAAI 2005*, pages 430–435. AAAI Press / The MIT Press, 2005.

[52] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th Conference on Design Automation*, pages 530–535. ACM Press, 2001.

[53] Rafiq Muhammad and Peter J. Stuckey. A Stochastic Non-CNF SAT Solver. In *PRICAI 2006: Trends in Artificial Intelligence*, LNCS 4099, pages 120–129. Springer, 2006.

[54] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, LNCS 4741, pages 529–543. Springer, 2007.

[55] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-based selection of policies for SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2009*, LNCS 5584, pages 326–340. Springer, 2009.

[56] Corina S. Pasareanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Model Checking Software – SPIN*, LNCS 2989, pages 164–181. Springer, 2004.

[57] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[58] Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In *International Conference on Logic Programming – ICLP 2005*, LNCS 3668, pages 221–234. Springer, 2005.

[59] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark., 1981.

[60] Steven David Prestwich. CNF encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 75–97 [3], 2009.

[61] Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format xcsp 2.1. *CoRR*, abs/0902.2362, 2009.

[62] Joachim Schimpf. Logical loops. In *Logic Programming, 18th International Conference, ICLP 2002*, LNCS 2401, pages 224–238. Springer, 2002.

[63] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *AAAI/IAAI*, pages 182–187. AAAI Press / The MIT Press, 2000.

[64] Peter J. Stuckey, Maria J. García de la Banda, Michael J. Maher, Kim Marriott, John K. Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The g12 project: Mapping solver independent models to efficient solutions. In *Principles and Practice of Constraint Programming - CP 2005*, LNCS 3709, pages 13–16. Springer, 2005.

[65] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*, LNCS 2173, pages 434–438. Springer, 2001.

[66] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.

[67] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving Non-clausal Formulas with DPLL Search. In *Principles and Practice of Constraint Programming - CP 2004*, LNCS 3258, pages 663–678. Springer, 2004.

[68] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs – TAP 2008*, LNCS 4966, pages 134–153. Springer, 2008.

[69] Milan Todorović. Applications of non-CNF solvers. Master's thesis, Faculty of Mathematics, University of Belgrade, 2011.

[70] G. S. Tseitin. On the complexity of derivations in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic (Part II)*, pages 115–125. Consultants Bureau, 1968. (Also in *The Automation of Reasoning*, Springer-Verlag, 1983.).

[71] Milan Šešum. Logical cryptoanalysis of des. Master's thesis, Faculty of Mathematics, University of Belgrade, 2009.

[72] Toby Walsh. SAT v CSP. In *Principles and Practice of Constraint Programming - CP 2000*, LNCS 1894, pages 441–456. Springer, 2000.

[73] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[74] Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT based scheduler for tournament schedules. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing - SAT 2004*, 2004. Online Proceedings.

[75] Lintao Zhang and Sharad Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Automated Deduction (CADE 2002)*, LNCS 2392, pages 295–313. Springer, 2002.