DOI: 10.2507/29th.daaam.proceedings.066

MODELING CONSTRAINT SATISFACTION PROBLEM WITH MODEL CHECKER

Bruno Blaskovic, Frano Skopljanac-Macina, Petar Knezevic, Niko Palic

This Publication has to be referred as: Blaskovic, B[runo]; Skopljanac-Macina, F[rano]; Knezevic, P[etar] & Palic, N[iko] (2018). Modeling Constraint Satisfaction Problem with Model Checker, Proceedings of the 29th DAAAM International Symposium, pp.0445-0453, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-20-4, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/29th.daaam.proceedings.066

Abstract

In this paper we interpret constraint satisfaction problem (CSP) with model checker Spin and bounded model checkers from satisfiability modulo theories (SMT) solvers. Our previous experience shows that single example modelled in different ways and interpreted with the same solver can have different space and time consumption. The goal is to find a feasible translation from of CSP problem to model checker and SMT solver. For that purpose we build Promela models for Spin model checker and Z3 models for SMT solver. Domain problem used in examples is graph colouring applied to Sudoku puzzle. In the beginning we briefly introduce CSP, model checking, graph colouring and Sudoku puzzle. The central part of this paper deals with various modelling efforts of Sudoku puzzle. After that results are analysed and compared. The main benefits of this paper are twofold, as we use them for educational purposes within *Formal Methods in System Design* course and as a solution to industrial scale problems like wavelength assignment in photonic networks respectively. At the end we give a conclusion and propose future research directions.

Keywords: constraint satisfaction problem; SMT solver; model checking; Spin/Promela; sudoku

1. Introduction and problem statement

Many problems can be analysed and solved within constraint satisfaction problem paradigm [1]. Solving constraint satisfaction problems is possible due to rich set of languages, solvers and constraint libraries.

Our goal is to realize “proof-of-concept” with constraint satisfaction model for sudoku puzzle using graph colouring. There are effective implementations of algorithms for graph colouring in C or C++ language [2]. Addition of new relational constraints requires complex modifications into the source code and such approach is not suitable for our requirements. Model checker Spin [3] and satisfiability modulo theories solver Z3 [4] are used to realize solver for sudoku puzzles. Both solvers can have graph colouring schema as a basis for the constraints definition.

Model checker Spin is designed for concurrent reactive systems verification and it is definitely not natural candidate for constraint satisfaction problems or graph colouring problems. Although model checker Spin usage as constraint satisfaction problem can be good student exercise and homework assignment, we find that there are additional constraints that are hard to realize within predefined constraints from satisfiability modulo theories solvers.

Constraints are implemented through *Promela* and Z3 language, *Promela* is the input language for model checker Spin and Z3 is the input language for Z3 satisfiability modulo theories solvers, respectively. We expect that Sudoku puzzle modelling should provide enough data for two real scale problems that have graph vertex colouring as theoretical background. These problems are routing and wavelength assignment in photonic networks [5], and time table allocation consistent with automated exam questions generation which is discussed in [6] and [7]. Adding new constraints concerning students' time tables could be beneficial when planning automated assessments through e-learning system for large groups of students. System could take into account lesson classes and laboratory exercises the student has taken before automatically selecting appropriate set of questions for on-line assessment or homework assignments.

Experience shows that the definition of constraints is the most critical moment during model and solution definition. Constraints definitions are hard to automatize. If model is well-structured, built-in decision procedures can lead to quick and stable solution. If we have additional requirements, interventions involving graph colouring may be necessary. An example is defining a time table for lessons schedule that takes extra care that the students do not need to complete laboratory exercises from a new topic before they covered it on the lesson classes. From our experience, this can be a very challenging problem, especially when dealing with large groups of students. We give special emphasis to input preparation in CSP, i.e. how to define the best possible model within the tool. Such efforts are of outmost importance and if applied in suboptimal way can degrade tool's performance. In this paper main contribution is twofold, we introduce models for satisfiability modulo theories solver and model checker Spin, respectively. After that, we provide examples suitable for students' lessons. Proposed approaches will also be used for educational purposes during lectures and homework assignments in graduate course *Formal Methods in System Design*.

This paper is organized as follows: in the 2nd section we give scope and motivation; in the 3rd section we briefly cover theoretical background. In the 4th section we discuss the concept of modelling for solution. In the 5th section we describe, report and discuss about our experimental results, and in the 6th section we conclude the paper.

2. Scope and motivation

Many problems from the real world can be solved using graph colouring. At the same time, there are a lot of algorithms and their implementations that help designers come up with stable and feasible solutions.

Graph colouring provides the solution for many problem domains like scheduling and time table allocation, frequency assignment in wireless networks, routing and frequency allocation in photonic networks, register allocations, analysis of biological data and many others.

For that purpose real scale problem is modelled as a labelled graph. Graph colouring algorithms assign a colour to each vertex. Colouring schema provides the solution for our problem.

Input to CSP is a graph where each vertex contains colour. Colours can be encoded as integer numbers. Sudoku graph is a grid graph where cells are vertices with assigned numbers. Typical 9x9 sudoku puzzle has 9 colours denoted with integers from 1 to 9. Each row and column, and each of the nine 3x3 sub-grids must have all 9 distinct colours. Satisfiability modulo theories solvers like Z3 SMT solver have built-in "*AllDifferent*" constraint. Besides "*AllDifferent*" constraints we can achieve same effect with a new constraint "*Sum*": each row, grid or sub-grid must have *Sum == 45*.

The goal for real world problems is to find input graphs for assigning colours and optimal solution for various problems, such as:

- Modelling for routing and wavelength assignment (RWA) problem [5] in photonic networks. Wavelengths are colours. Real networks can have more than 80 colours. Colour assignment must follow additional requirements and rules, for example colours must be aware of optical signal latency, availability and reliability of optical links.
- Automated exam question generation (AEQG) with exam time table allocation. Here, the additional requirement is: "Each student must take a quiz after the lecture takes place within that same week". During the 15 weeks in semester each student takes 3-5 quizzes. For that purpose we use automated exam question generation. Problem is twofold: optimal time table for 30-40 groups of 16 students, and avoiding situation when student takes test with questions that were not yet taught in the lectures.

In this paper we are introducing additional not-built-in constraints on Sudoku puzzle example. Such requirements are expressed through additional constraints using available logical and relational constructs. There is no fixed list of such constraints and no possibility to find fully suitable CSP language and suitable CSP solver. There is still open question, whether to analyse available CSP solvers with above problems. Even with this approach it is still difficult to define additional constraints when constraints are dynamically changed. For that purpose we suggest to connect CSP solvers with planning and scheduling tools [8].

3. Theoretical background

In this section we briefly introduce most important definitions and facts from background theories.

Definition 1 – Labelled graph is a triple $G(V, E, L_V)$, where V is the set of vertices, E is the set of edges and L_V is the set of vertex labels or colours.

Definition 2 – Colouring of a labelled graph $G(V, E, L_V)$ assigns colours to vertices of G such that each two adjacent vertices are assigned different colour $c \in L_V$.

We must note that in the case of sudoku puzzles we will use integers instead of colours. In that case the set of vertex labels will be: $L_V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Definition 3 – Constraint satisfaction problem is a quadruple $CSP = (X, D, C, R)$, where X is the set of variables, D is the set of variable domains, C is the set of constraints and R is the set of relational constraints.

Definition 4 – Graph colouring with constraints (for 9x9 sudoku puzzle example) is defined as a $CSP = (X, D, C, R)$ with: $X = \{x_1, \ldots, x_n\}$, where $n = 9 \times 9 = 81$; $D = \{1, \ldots i \ldots, k\}$, where $k = 9$; C: $x_i \neq x_j$, $\forall (i, j) \in E$ where (i, j) is an edge and $AllDiferent(x_i \mid i \in C)\ \forall c \in C$; and $R = \sum_{i=1}^{k} \frac{k}{2}(1 + k)$.

Definition 5 – Sudoku square of order n^4, with $n^2 \times n^2$ gird with values 1 to n^2, such that each entry in each row, each column and in each of the n^2 major $n \times n$ blocks are different [9].

We must note that the Definitions 4 and 5 are identical for the graph grid with $n \times n$ vertices.

4. Modelling for solution

In the last sections we introduced concepts that are phases of modelling and solving CSPs:

- pre-processing (reduction): eliminate non-feasible values (reducing reachability tree - problems' search space)
  - Z3 SMT solver uses built-in construct for constraints with underlying logic.
  - Model checker Spin operates on Kripke structure with constraints expressed in propositional logic or in temporal linear (LTL) logic.
- search:
  - SMT solvers have built-in highly optimized algorithms for search.
  - Model checker Spin has built-in searching algorithm, and uses various optimisations such as bitstate-hashing or parallel execution to additionally reduce search space.
- assigning values:
  - Z3 SMT solver in the last step gives solution if and only if a solution is found. Other possibilities are infinite loop for ill-structured constraints or messages that there is no solution for the given set of constrains.
  - Model checker Spin produces solution as a counterexample.

Designing process models can be simplified if the solution for the problem can be reduced to a problem with known complexity and tool support. In this section we will describe search for a solution that consists of subsequent steps of problem reductions. As a working example Sudoku puzzle is used [10]. In first step Sudoku puzzle can be solved with reduction to:

- constraint satisfaction programming [11],
- graph colouring problem (gcol),
- model checking [12],
- SAT (Boolean SATisfiability problem) [13],
- SMT (Satisfiability Modulo Theory problem) [14].

The main goal of the first step is to find feasible approach for reduction that will produce solution within required space/time limits. Our solution is to define constrains as graph colouring problem and implement them in Z3 SMT solver and model checker Spin. In the next section we introduce some examples to illustrate these introduced concepts.

5. Experiments and results

In our experiments we focused on modelling and solving graph colouring CSP problems. As a domain of graph colouring problems, we chose sudoku puzzles of various dimensions. These types of logical puzzles are popular and well-known, and we believe they can serve as a useful learning tool when introducing students to the concepts of model checking and automated solving methods on the course *Formal methods in system design*.

5.1. Brief introduction to sudoku puzzles

Sudoku (translated from Japanese: *single digit*) is a logic puzzle defined as a 9x9 grid with 81 places (comprised of 9 smaller 3x3 grids, each with 9 places) that is partially filled with digits between 1 and 9 (*initial clues*). The goal of the puzzle is to completely fill the rest of the grid using digits from 1 to 9, while ensuring there are no multiple same digits in each row, column and small grid. The problem can be trivial, e.g. if there are many initial clues, but generally it can be quite challenging. The initial clues should be carefully selected, so that the sudoku puzzle has only one solution.

Nevertheless, there can be sudoku puzzles which have multiple correct solutions – especially when the number of initial clues is too small. We experimented with the automated solving of regular 9x9 sudoku puzzles (Fig. 1a), but also sudoku type puzzles of smaller dimensions: 6x6 (Fig. 1b) and 4x4 (Fig. 1c), which use digits 1 to 6 and 1 to 4, respectively. Also, we initially started with a simple 3x3 grid (Fig. 1d), which uses digits 1 to 3. But, we must note that this is a small Latin square and not a sudoku puzzle because it cannot be divided into three smaller grids of equal size.

It is straightforward to find that the number of all combinations of the digits 1 to 9 in a 9x9 grid (81 places) is $9^{81} \approx 1.97 \cdot 10^{77}$. Of course, we can assume that the great majority of those grids are not proper complete sudoku grids because they violate the row, column and small grid constraints. However, the authors in [15] found that the number of completed sudoku 9x9 grids is still very high: $6.67 \cdot 10^{21}$.

When we consider sudoku puzzles as a form of graph colouring problem, we must find a 9-colouring of a graph which is initially only partially coloured. There are 81 vertices in the graph (one vertex for each place in a sudoku grid), and each two vertices are connected by an edge if their places are in the same row, column or small grid. For practical reasons colours are encoded with the digits from 1 to 9.

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

a) 9x9 sudoku

	3	4			
	4		2	5	3
2		3			6
	1				2
		5	4	6	

b) 6x6 sudoku

4			
	2	1	
	4	3	
			1

c) 4x4 sudoku

1		2
		3

d) 3x3 grid

Fig. 1. Examples of sudoku puzzles

We used two different approaches in experimenting with modelling and solving CSP problems. Firstly, we considered CSP problem to be an SMT problem, and we used the Z3 SMT solver to find problem solution. In the second approach we used model checking techniques, namely model checker Spin and its modelling language *Promela*. We performed both sets of experiments on a standard average desktop computer (3 GHz Intel Core i3 CPU 540 and 4 GB of RAM) running *Linux Debian* operating system.

5.2. SMT solver experiments

One way for solving a CSP problem such as graph colouring is to model it as an SMT problem. Satisfiability modulo theories (SMT) problem is a generalization of the Boolean satisfiability problem (SAT). SAT solvers try to find an interpretation of a Boolean propositional logic formula that is satisfiable (*sat*), i.e. they search for a set of Boolean (binary) variable values that will evaluate the Boolean formula as *true*. Otherwise, if the SAT solver cannot find an appropriate set of Boolean variables values the Boolean formula is unsatisfiable (*unsat*). SMT problems use predicates instead of binary variables, and therefore they are more expressive than SAT problems. SMT solvers try to find a model in which all interpretations of a first-order logic formula are *true*.

In our experiments we used the Z3 SMT Theorem Prover developed by *Microsoft Research* [16]. More specifically, we used z3 and z3-solver libraries developed for the *Python* programming language. In practice, SMT solvers analyse input data, set of given rules and constraints and try to decide if the defined problem is satisfiable. If the problem is satisfiable the solver responds with *sat* and returns a model of a possible solution, and if the problem is not satisfiable the solver responds with *unsat* (Fig. 2).

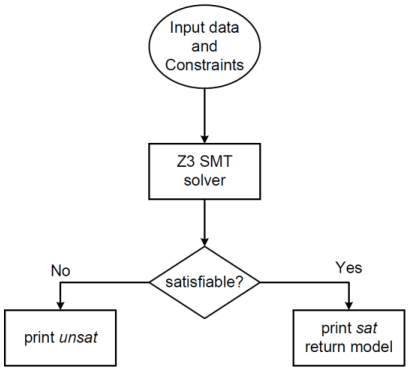

Fig. 2. Overview of the Z3 SMT solver system

To solve regular sudoku puzzles (9x9 grid) using Z3 SMT solver, we need to define the format of the sudoku puzzle and its initial known data, as well as all the constraints. At the beginning of the python script we must import modules from the z3 library. Then, we specify sudoku's 9x9 grid as a two-dimensional list with 81 integer elements, and we instantiate the solver with *s = Solver()*. Afterwards, the starting grid information must be given to the solver. Those elements are invariants – they should not be changed during the problem-solving phase and they must appear in their positions also in the model of the possible solution. Next, using function *s.add()* we add constraints to the defined problem. First, we give the solver an important constraint that each element of the list must be in the integer interval from 1 to 9. If we omit this constraint the solver will unnecessarily use the entire integer range while searching for a solution. Now, we can also add key sudoku constraints which state that the elements of each row, column and a small grid must be distinct using *s.add(Distinct(list_of_elements))* command. With that, we have fully defined the given sudoku problem and we can instruct solver to find possible solutions by calling the method *s.check()*. If the solution has been found the solver returns string *sat*, and *unsat* otherwise. And most importantly, if the problem is satisfiable we want to know the solution of the sudoku problem. Therefore, we obtain the solution model with the command *model = s.model()*, and after some formatting we print out the found solution of the problem. As we have stated earlier, sometimes this solution will not be unique – there can also be some other correct solutions for the given sudoku puzzle.

5.3. Results of SMT solver experiments

Based on the numerous runs of the Python script executions using different examples of various sudoku puzzles found online and in print magazines we found that the Z3 SMT solver is efficient, fast and reliable. Though, it must be noted that the first installation of the Z3 SMT solver's Python library can take a considerable time to complete. After executing shell command *pip install z-3 solver Z3* the download and installation process begins, which includes compiling of many required programs and modules. From our experience this process can take more than 30 minutes to finish.

As expected, easier sudoku puzzle problems are solved very quickly (Fig. 3a). Using standard *Linux* shell tool *time* we found that the majority of the scripts in this case finished just under 0.2 s. When using moderate sudoku puzzles the execution time increased to around 0.2 s (Fig. 3b). For some harder sudoku problems we have observed somewhat longer script execution time, but still it was very fast – around 0.25 s (Fig. 3c). As expected, if the sudoku puzzle is not properly given (error in the puzzle itself or a mistype during the input of the puzzle) the Z3 SMT solver will find that there is no solution (*unsat*). In that case the script execution will also be very fast – around 0.15 s (Fig. 3d – there is an error in the initial grid, there are two digits 7 in the upper left small grid). We also tested our script repeatedly with sudoku puzzles that have multiple correct solutions (Fig. 3e). It was interesting to observe that the solver in those cases always returned only one same solution to a particular problem.

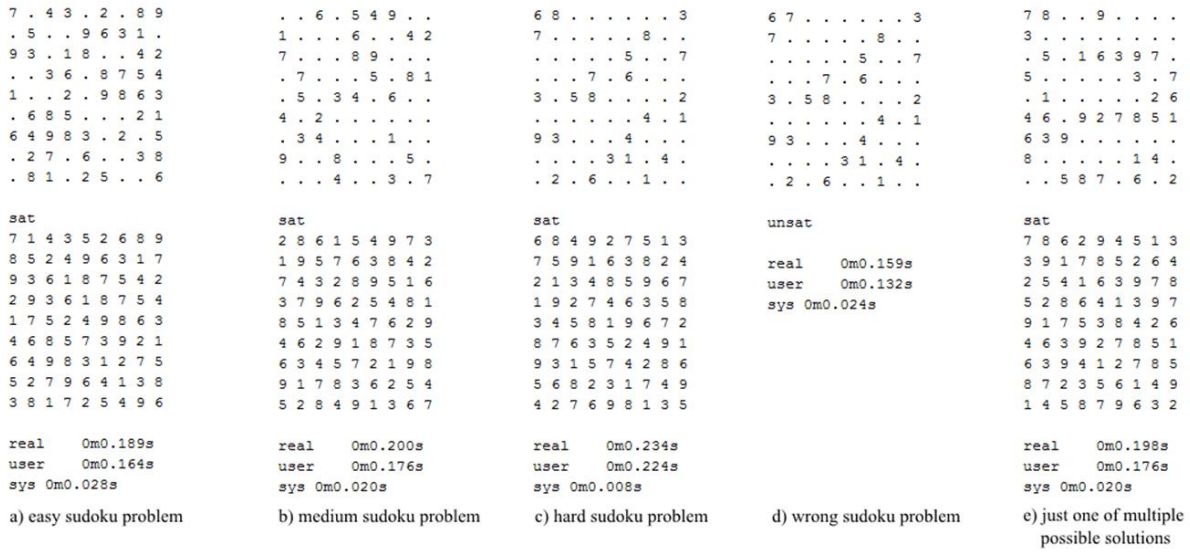

Fig. 3. SMT solver results for different sudoku problems

5.4. Spin model checker experiments

For this approach we used model checking methods and techniques to model and solve CSP problems. Model checking is a formal method for automatic verifying if a model of a system satisfies given properties. This technique helps to ensure that the hardware or software systems avoid critical errors, such as deadlocks and systems crashes. One of the important model checking tools is the Spin model checker. It is predominately used for formal verification of complex concurrent software systems, such as concurrent reactive software.

Spin has its own modelling language called *Promela* (Process meta language) for creating Spin models that can be then automatically analysed and verified. When creating Spin models, we define process models (*proctype* code blocks) that describe the behaviour of the system. Process models can use global and local variables and data structures and they can use message channels for communication. To check system properties, we can use assertions (*assert* command) or linear temporal logic (LTL) formulas.

Spin can be used in two different ways – in the simulation mode and in the verification mode. Simulation mode performs a non-deterministic run of the Spin model. Furthermore, we can use guided interactive simulation where the user can choose execution path at every decision point during the simulation run. Verification mode performs a full exhaustive search of all the execution paths of the Spin model. A few extra steps are needed before performing verification. First, Spin tool automatically translates defined Spin model in *Promela* to the process analyser program in C (*pan.c*). Then we must use a standard C compiler (*gcc*) to compile *pan.c*. Finally, we start the verification by running the compiled *pan* program. This will use more system resources, but it is necessary when trying to prove if the system satisfies a certain property. When the checked property is violated (e.g. a variable is out of allowed bounds) verification fails and its execution trail is automatically generated. Finally, we can use the Spin tool to replay this execution trail that leads us to the error in the Spin model.

In experimenting with solving sudoku puzzles of various complexity and grid dimensions we used both the simulation mode and the verification mode of the Spin tool. We developed two basic solving strategies, for simulation runs we used Spin models with loops (Fig. 4), and for verification process we used Spin models without loops (Fig. 5).

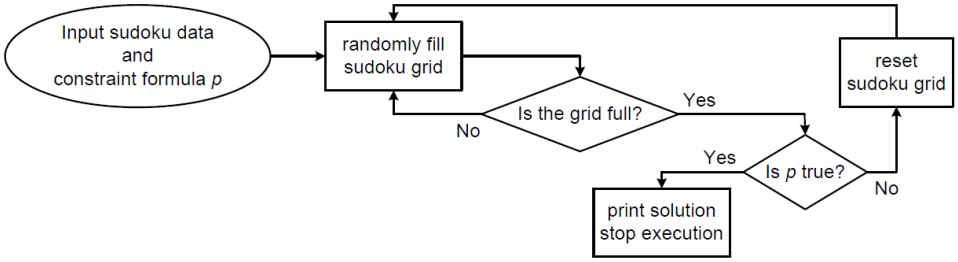

Fig. 4. Process of solving sudoku puzzles using Spin model checker (Spin model with loops)

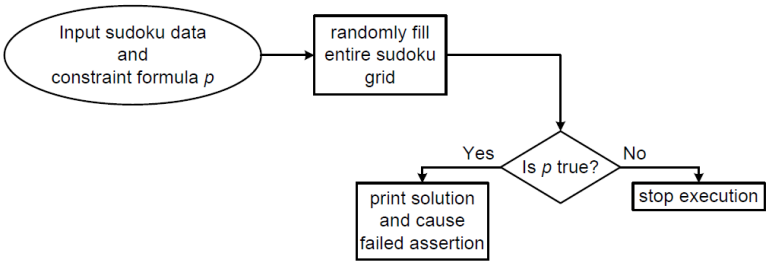

Fig. 5. Process of solving sudoku puzzles using Spin model checker (Spin model without loops)

In the both approaches, we must first define the starting sudoku grid as a list of integers (initially unknown grid places are set to 0). Then we define a conjunctive logical formula that connects all the constraints: allowed interval of sudoku grid elements, and distinct values in each row, column or small grid. Afterwards the strategies diverge slightly. Spin model for simulation mode randomly fills in each loop iteration one missing grid element until the whole sudoku grid is full. Spin model for verification randomly fills all the missing elements of the sudoku grid as listed in their ascending order. Then the constraints' formula is checked. In the simulation mode if the formula is not satisfied the sudoku grid is reset (*backtracking*) and it will be filled randomly again. But, if the constraints' formula is satisfied then the solution is found, the resulting sudoku grid is printed out and the execution stops. On the contrary, in the verification mode if the formula is not satisfied the execution just stops, and if the formula is satisfied the sudoku puzzle solution is printed out and an intentionally placed false assertion is executed. In the verification process all possible execution paths are explored, and when the assertion fails an error is reported by the verification process and its entire execution run trail is generated. This is standard procedure with "counterexamples" which is built into every model checker. Counterexamples are generated when assertions or LTL formulae fail.

At first, we used these two basic strategies to develop simple Spin models for solving 3x3 grids and 4x4 sudoku puzzles. However, when we tried to use them for solving larger, 6x6 and 9x9 sudoku grids we encountered problems. In those cases the search space grew exponentially and it was impossible to find a solution even for moderate 9x9 sudoku puzzles in an acceptable amount of time. Therefore, we needed to improve our basic strategies, and we did that by imposing additional rules and constraints in the sudoku grid filling phase of the Spin models. First, we added a new constraint for random selection of values for sudoku grid elements.

Now, all the values that are already in some empty element's row, column or small grid are skipped. This helped to solve most 6x6 and easier 9x9 sudoku puzzles much quicker, but it still was not enough for moderate or hard 9x9 sudoku puzzles. The second optimisation step was intended for simulation model only. We added a new rule for selecting the order by which the empty grid elements are filled. Before each iteration of the main loop, system first calculates which rows, columns or small grids have fewest empty places. Then, the each iteration of the loop will only select randomly one of those empty elements. This strategy was not as helpful as we hoped, especially for harder sudoku problems which have small number (20 or less) initially known grid elements.

Finally, we decided to remove the second optimisation and to approach the problem differently. We added a 9 element array storing all the possible candidate digits for each grid element. System checks what values can be assigned to each element before it enters the main loop. Then, before each iteration of the main loop the system finds a subset of empty elements which have the fewest available values left, and randomly selects one of them. After the element is filled the system updates possible candidates for all remaining empty places in that element's row, column and small grid. The similar strategy improvement was successfully used for verification model, but there we just skip the search for the empty elements with the minimal number of available values. Using these improvements we managed to reduce the search space drastically, which means that our simulation model and the verification model solve even hard 9x9 sudoku puzzles in very short time almost as quick as the SMT solver. We made one additional important optimisation – grid elements were changed from an integer list to a byte list. This made sense because the allowed values are only in the range 1 to 9. This was especially useful for the verification model, because it reduced memory consumption, the state-vector size (contains all the variables used in the model), and the overall duration of the verification run by a factor of ~4. With such approach, we have hybrid solution that mimics imperative algorithms and connects procedural and declarative programming in a feasible way.

5.5. Results of Spin model checker experiments

Spin model checker has proved to be very efficient with small sudoku problems: 3x3 grids, 4x4 sudoku and 6x6 sudoku were all solved in simulation mode very quickly, almost instantly (in ~0.1 s or less) as it is shown in Fig. 6a, 6b and 6c respectively. Using described model optimisations we also achieved simulation execution times for the same harder 9x9 sudoku problems of around 0.3 s, comparable to 0.2-0.25 s for SMT solver experiments (Fig. 6d and 6e). If the sudoku does not have a unique solution the Spin model checker is able to find different correct solution every time run the simulation is performed (e.g. two possible solutions shown on Fig. 6f – compare with Fig. 3e).

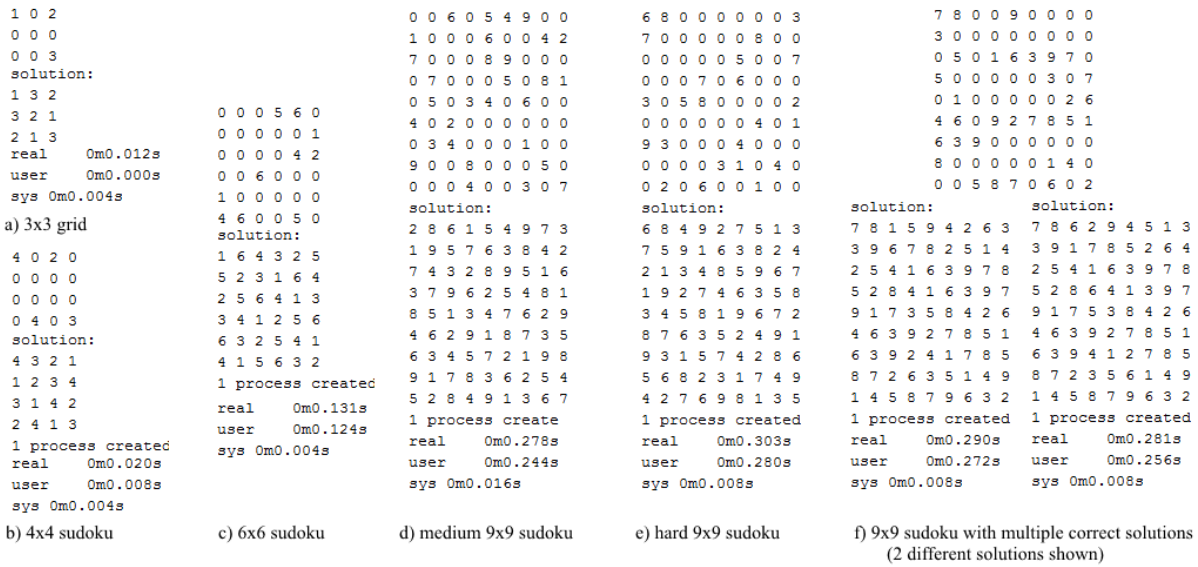

Fig. 6. Spin model checker simulation results for various sudoku puzzles

Verification process performs an exhaustive search of all execution paths, and it usually takes more time than a simulation run. We must note, that using special flag *-e* during the verification process we can find almost all the solutions (shell command *./pan -e*) for a given sudoku problem. Therefore, we made a *Perl* script which automatically post-processes and stores the results of the verification. It checks all the generated execution run trails and extracts and counts all the different solutions from those trails. For an example in Fig. 7a the whole verification process took 9 s, but most of that time is used for compiling verification program (~7 s) and for post-processing (~1 s). We see in the Fig. 7b that the verification process found that the sudoku problem (also shown in Fig. 3e) has exactly 28 different possible correct solutions. This verification run took ~16 s, because it took longer to execute all generated error trails and to post-process all the results.

For a harder sudoku problem shown on Fig. 7c the verification process took ~26 s, but this time the majority of the time was indeed spent during the actual verification run (~18 s), and the rest was used for verification program compilation (~7 s) and results post-processing (~2 s). Described verification process can verify if the sudoku puzzle has only one unique solution, as shown in Fig. 7a and 7c.

```
Sudoku problem:
0 0 6 0 5 4 9 0 0
1 0 0 0 6 0 0 4 2
7 0 0 0 8 9 0 0 0
0 7 0 0 0 5 0 8 1
0 5 0 3 4 0 6 0 0
4 0 2 0 0 0 0 0 0
0 3 4 0 0 0 1 0 0
9 0 0 8 0 0 0 5 0
0 0 0 4 0 0 3 0 7
-----------------------
2 8 6 1 5 4 9 7 3
1 9 5 7 6 3 8 4 2
7 4 3 2 8 9 5 1 6
3 7 9 6 2 5 4 8 1
8 5 1 3 4 7 6 2 9
4 6 2 9 1 8 7 3 5
6 3 4 5 7 2 1 9 8
9 1 7 8 3 6 2 5 4
5 2 8 4 9 1 3 6 7
Found solution 1 !
VERIFICATION SUMMARY:
One unique solution!
```

a) medium sudoku puzzle

```
Sudoku problem:
7 8 0 0 9 0 0 0 0
3 0 0 0 0 0 0 0 0
0 5 0 1 6 3 9 7 0
5 0 0 0 0 0 3 0 7
0 1 0 0 0 0 0 2 6
4 6 0 9 2 7 8 5 1
6 3 9 0 0 0 0 0 0
8 0 0 0 0 0 1 4 0
0 0 5 8 7 0 6 0 2
-----------------------
7 8 6 5 9 4 2 1 3
3 9 1 7 8 2 5 6 4
2 5 4 1 6 3 9 7 8
5 2 8 4 1 6 3 9 7
9 1 7 3 5 8 4 2 6
4 6 3 9 2 7 8 5 1
6 3 9 2 4 1 7 8 5
8 7 2 6 3 5 1 4 9
1 4 5 8 7 9 6 3 2
Found solution 28 !
VERIFICATION SUMMARY:
28 different correct solutions!
```

b) sudoku puzzle with multiple solutions (only the last, 28th solution shown)

```
Sudoku problem:
6 8 0 0 0 0 0 0 3
7 0 0 0 0 0 8 0 0
0 0 0 0 0 5 0 0 7
0 0 0 7 0 6 0 0 0
3 0 5 8 0 0 0 0 2
0 0 0 0 0 0 4 0 1
9 3 0 0 0 4 0 0 0
0 0 0 0 3 1 0 4 0
0 2 0 6 0 0 1 0 0
-----------------------
6 8 4 9 2 7 5 1 3
7 5 9 1 6 3 8 2 4
2 1 3 4 8 5 9 6 7
1 9 2 7 4 6 3 5 8
3 4 5 8 1 9 6 7 2
8 7 6 3 5 2 4 9 1
9 3 1 5 7 4 2 8 6
5 6 8 2 3 1 7 4 9
4 2 7 6 9 8 1 3 5
Found solution 1 !
VERIFICATION SUMMARY:
One unique solution!
```

c) hard sudoku puzzle

Fig. 7. Spin model checker verification results for different sudoku puzzles

5.6. Discussion

As it can be seen from the results, in the end both the SMT solver and Spin model checker solved various sudoku puzzles in a quick and an efficient way. Nevertheless, execution run-time of the SMT solver examples was shorter compared to the Spin simulation runs (around 0.1 s slower), and especially shorter when compared to Spin verification runs. However, an exhaustive verification process using Spin model checker can give us valuable information about the sudoku problem, namely if it has one unique solution or multiple correct solutions. Earlier, we have seen that the SMT model checker will only return one possible solution of a sudoku puzzle.

Concerning usability, we can clearly see that the SMT solver is more user-friendly, and that is already has various useful methods and functions. To achieve similar results using Spin model checker, we needed to write quite complex Spin models in *Promela*, which has a limited expressive power compared to C and *Python*. For computer engineering education in the field of formal methods these examples can be useful learning tools, which help students to understand the problems of combinatorial explosion and to find the procedures for reducing problem's search space. Also, the students could see what would happen if we tried to verify Spin simulation model – because of the loops the search space rises dramatically and the verification process would run out of memory fast without finding a satisfiable solution. Similarly, if the students would run the simulation of the verification model it would stop immediately – either the solution was randomly found (highly unlikely), or, more likely, the solution was not found.

6. Conclusion and future work

In our paper we presented an approach for modelling constraint satisfaction problem using Spin model checker and compared its results with the established SMT solver Z3. We tested proposed methods on a graph colouring problem in a form of sudoku puzzles of various sizes and difficulties. When using Spin model checker in the simulation mode we obtained results comparable with Z3 SMT solver performance on the same set of sudoku puzzles. The verification mode of the Spin model checker is more resource demanding and slower than the Z3 SMT solver. However, using Spin verification mode we could prove that a sudoku puzzle has one unique solution or multiple correct solutions. If a sudoku puzzle does not have a unique solution Spin model checker also provides an option for generating almost all correct solutions, whereas SMT solver by default only returns one possible correct solution.

We plan to use these findings as valuable learning tools in teaching *Formal Methods in System Design* course, and also explore possibilities of using this modelling technique for solving problems in photonic networks and creating time tables for students.

7. References

[1] Dechter, R. (2003) Constraint Processing, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[2] Johnson, D.S.; Mehrotra, A. & Trick M.A. (2008). Special issue on computational methods for graph coloring and its generalizations. Discrete Applied Mathematics, 156(2):145–146.

[3] Holzmann, G. (2004). Spin Model Checker, the: Primer and Reference Manual. Addison-Wesley Professional.

[4] De Moura, L. & Bjørner, N. (2008). Z3: An efficient smt solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pp. 337–340, Springer-Verlag, Berlin, Heidelberg.

[5] Mikac, B.; Inkret, R. & Tillerot, F. (1998). Wavelength assignment in WDM networks - A comparison of approaches. In Optical Networks: Design and Modeling, IFIP TC6 Second International Working Conference on Optical Network Design and Modeling (ONDM '98), February 9-11, 1998, Rome, Italy, pp. 37–44.

[6] Škopljanac-Mačina, F. & Blašković, B. (2013). Formal Concept Analysis – Overview and Applications, 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, Zadar, Procedia Engineering 69 (2014) pp. 1258-1267, DOI: https://doi.org/10.1016/j.proeng.2014.03.117

[7] Skopljanac-Macina, F; Blaskovic, B. & Pintar, D. (2016). Automated Generation of Questions for Basic Electrical Engineering Education, Proceedings of the 27th DAAAM International Symposium, pp.0377-0385, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-08-2, ISSN 1726-9679, Vienna, Austria, DOI: 10.2507/27th.daaam.proceedings.056

[8] Uyar, A.S.; Ozcan, E. & Urquhart, N. (2015). Automated Scheduling and Planning: From Theory to Practice. Springer Publishing Company, Incorporated.

[9] Simonis, H. Sudoku as a constraint problem, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.2964, unpublished.

[10] Lorch, J. (2012). Magic squares and sudoku. The American Mathematical Monthly, 119(9):759–770.

[11] Tsang, E. (1996). Foundations of constraint satisfaction. Academic Press Limited, London.

[12] Clarke, E.M.; Emerson, E.A. & Sifakis, J. (2009). Model checking: Algorithmic verification and debugging. Commun. ACM, 52(11):74–84.

[13] Kautz, H. & Selman, B. (2007). The state of sat. Discrete Applied Mathematics, 155(12):1514 – 1524, 2007. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[14] Barrett, C.W.; Sebastiani, R.; Seshia, S.A. & Tinelli, C. (2009). Satisfiability modulo theories. In Handbook of Satisfiability, 825–885.

[15] http://www.afjarvis.staff.shef.ac.uk/sudoku/, (2006). Sudoku enumeration problems, Accessed on: 2018-09-24

[16] https://z3prover.github.io/api/html/, (2006). Z3 Theorem Prover, Accessed on: 2018-09-26