Automated Reasoning in the Class

I. Drămnesc (West University of Timişoara, Romania)
E. Ábrahám (RWTH Aachen, Germany)
T. Jebelean (Johannes Kepler University of Linz, Austria)
G. Kusper (Eszterhazy Catholic University of Eger, Hungary)
S. Stratulat (Univ. de Lorraine, CNRS, LORIA, Metz, France)

isabela.dramnesc@e-uvt.ro
abraham@cs.rwth-aachen.de
tudor.jebelean@jku.at
gkusper@aries.ektf.hu
sorin.stratulat@univ-lorraine.fr

Introduction

Automated Reasoning is a field of Computer Science concerned with the automation of deduction processes in Mathematics. For this purpose, algorithms for logical deduction (Automated Theorem Proving) have been developed. Theorem provers allow to prove mathematical statements interactively by computer programs, SAT solvers are available for checking the satisfiability of propositional logic formulae, and SMT (Satisfiability Modulo Theories) solvers that combine logical procedures with methods from Computer Algebra (e.g. for the manipulation of polynomial expressions) can be used to check the satisfiability of Boolean combinations of constraints from certain theories.

Automated Reasoning finds application in numerous areas, for example for the verification of computer hardware and software, and in general of complex systems, as well as semantical information storage and retrieval. Therefore it is crucial for computer specialists to understand the theoretical basis and familiarize themselves with the underlying algorithms, in order to be able to develop Automated Reasoning tools or to use them for solving their problems. Moreover, this knowledge is also necessary for the development of semantically aware Internet repositories and tools to access them. In practice, we currently encounter numerous situations in which design or implementation errors in complex systems lead to undesired results, ranging from small nuisances to important loss of value and even to fatalities. In our opinion, an important cause for such errors is the lack of systematic use of formal modeling and verification tools, and one important way of changing this situation is the improvement of the education of students and of the academic staff in fields related to Computational Logic.

As the field of Automated Reasoning is relatively new, there is yet not enough teaching material that sufficiently addresses mathematically involved concepts for the education of students with diverse background, and in particular material that helps to increase the motivation of both the students and the teachers and supports efficient interaction.

The project *ARC (Automated Reasoning in the Class)* aimed at improving the teaching of subjects related to Automated Reasoning by producing teaching material and tools that support the activities in the class and by training academic staff on how to use them. The project was funded in the frame of the *Erasmus+* programme of the European Union as a partnership between 5 universities from Romania (coordinator), Austria, France, Germany, and Hungary, running from 2019 to 2022.

The main activities of the project have been: production of a book and related tools (available on the project home page [1]), 5 modules for training of teaching staff, a summer school for students, and an international symposium for disseminating the project results. The book *"Computational Logic: A Practical Approach"* describes the main models from Mathematical Logic and the most important algorithms from Automated Reasoning, with corresponding illustrative exercises. The tools are various programs that illustrate the main methods and can be used to support the teaching based on the various sec-

tions of the book. The training of the academic staff for using the tools and the book took place in 5 one–week modules with the subjects: *Mathematica and Theorema* (Linz, Austria), *Satisfiability Module Theories (SMT) Solving* (Aachen, Germany), *Problem Based Learning* (Timişoara, Romania), *SAT Solving* (Eger, Hungary), and *Coq* (Metz, France).

As an illustration of our approach we describe in more detail the teaching process of the DPLL algorithm [2, 3] for checking the satisfiability of propositional logic formulas. The material and the tools for the other algorithms described in our book are freely available on the home page of the project [1].

Teaching the DPLL Algorithm

Problem-based Learning

To motivate the application of logics and automated reasoning, we developed material to illustrate how certain problems can be encoded logically; later on, these logical encodings can also be used to illustrate the execution of automated reasoning algorithms and the application of relevant tools.

In this article we focus on *propositional logic* [4], whose formulas connect propositions (Boolean variables) by the unary operation of negation ($\neg$) and the binary operators of conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) etc. A *literal* is a proposition or its negation; a *clause* is a disjunction of literals; a propositional logic formula in *conjunctive normal form* (*CNF*) is a conjunction of clauses. Below we give two examples how to encode problems in propositional logic.

Example: *The pigeon hole problem*. Let $n \in \mathbb{N}\setminus\{0\}$. The *pigeon hole problem* is the problem to decide whether $n+1$ pigeons fit into n holes, if no two pigeons fit into one hole. This problem is an example which can be easily solved by humans but which often challenges automated reasoning tools. We can encode this problem in propositional logic as follows, where $x_{i,j}$ stands for pigeon i being in hole j ($1 \le i \le n+1$, $1 \le j \le n$):

$$\left(\bigwedge_{i=1}^{n+1}\left(\bigvee_{j=1}^{n} x_{i,j}\right)\right) \wedge \left(\bigwedge_{j=1}^{n}\bigwedge_{i1=1}^{n}\bigwedge_{i2=i1+1}^{n+1}\left(\neg x_{i1,j} \vee \neg x_{i2,j}\right)\right)$$

Above, the left operand of the outmost conjunction encodes that each pigeon is in at least one hole, and the right operand encodes for each pair of pigeons that they cannot be in the same hole. Note that we break the symmetry for the pigeon pairs in the second block, ordering the pairs by increasing indices. Furthermore, we require only that each pigeon is in at least one hole; one could also encode the fact that each pigeon is in at most one hole, but since we only want to decide the existence of a solution we know that if there is a solution in which a pigeon uses more than one hole than there is also a solution in which it uses exactly one hole (the others being empty). On this example we can illustrate the relevance of how we encode problems.

Example: *Sudoku Light*. Assume an $n \times n$ square grid for some $n \in \mathbb{N} \setminus \{0\}$. Each square in the grid is either empty or it contains a natural number from $\{1, \ldots, n\}$. We want to fill each empty square in the grid with numbers such that each row and each column contain exactly one occurrence of each number from $\{1, \ldots, n\}$. This problem can be encoded by the propositional logic formula

$$\varphi_{init} \wedge \varphi_{squares} \wedge \varphi_{rows} \wedge \varphi_{columns}$$

using propositions $g_{i,j,k}$ for $i, j, k \in \{1, ..., n\}$ to encode that the square in row i and column j contains the number k; φ_{init} is a conjunction stating for each initially non-empty field in row i and column j with number k the truth of $g_{i,j,k}$; and the following sub-formulas:
Each square has at most one number:

$$\varphi_{squares} := \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n}\bigwedge_{k1=1}^{n-1}\bigwedge_{k2=k1+1}^{n}\left(\neg g_{i,j,k1} \vee \neg g_{i,j,k2}\right)$$

Each row contains each number:

$$\varphi_{rows} := \bigwedge_{i=1}^{n}\bigwedge_{k=1}^{n}\left(\bigvee_{j=1}^{n} g_{i,j,k}\right)$$

Each column contains each number:

$$\varphi_{columns} := \bigwedge_{j=1}^{n}\bigwedge_{k=1}^{n}\left(\bigvee_{i=1}^{n} g_{i,j,k}\right)$$

Teaching the DPLL Algorithm with Mathematica

The Davis–Putnam–Logemann–Lovelace (DPLL) algorithm [2, 3] solves SAT problems in conjunctive normal form by investigating on a search tree a certain subset of the possible assignments to the propositional variables and by unit propagation.

A *unit* clause consists of a single literal – note that such a clause determines the truth assignment of the corresponding variable, since each of the clauses of a CNF formula need to be satisfied in order to satisfy the whole formula. Unit propagation uses this information to simplify the set of clauses: all instances of the opposite of this literal (whose truth value is *false*) are removed from the corresponding clauses (*unit resolution*) and all clauses containing an instance of this literal (whose truth value is *true*) are deleted (*unit subsumption*). (The unit clause itself is also removed by this rule, but the corresponding assignment is kept in case we want to actually find the solution[s].) Note that unit resolution may also produce new units – in this case they are also propagated (*Boolean constraint propagation*) – as well as an empty clause or an empty set of clauses.

When there is no unit clause in the current set, then the algorithm proceeds by *branching*: it chooses one of the variables and it produces two branches for the possible truth assignments and propagates these. A heuristics decides which branch to process first.

The search on a branch finishes in two possible ways. If unit resolution produces the empty clause (a *contradiction* is found), then the formula is not satisfied on the

current branch of the search tree; the algorithm backtracks by moving up along the current path to the most recent branching point with a yet unexplored child and continues with processing that child. If unit subsumption empties the current clause set, then the formula is satisfied for the current assignment of the variables. In case we are only interested in satisfiability, then the search can stop at the first such satisfying situation.

Some variants of DPLL also use the *pure literal rule*: if a literal occurs with only one sign in the problem, then all the corresponding clauses can be removed without changing the satisfiability of the problem.

The formula is unsatisfiable if contradiction occurs on all branches. Otherwise, each satisfiable branch produces one or more solutions (the variables that have not been assigned can have any value).

Example: *DPLL solving*. Fig.1 shows the search tree for the set of clauses listed in node 0.

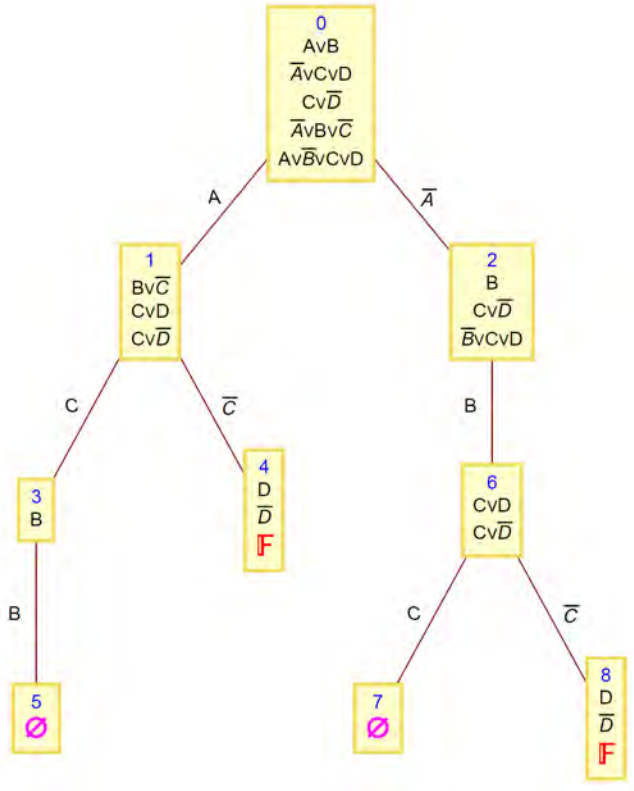


Figure 1: *Example of DPLL solving.*

Because there is no unit clause in the original set, the first step is a branching one, using the variable A and producing the two new sets 1 and 2. (One can use any of the other variables and then the sets would be different.) On the branch corresponding to A the first and the last clause are removed, while the second and the fourth clause loose the negation of A. On the branch corresponding to the negation of A the second and the third clause are removed, while the first and the last clause loose A. On both branches the third clause remains unchanged.

Set 1 again does not contain any unit, thus branching is applied to C producing sets 3 and 4. In set 3 the second and the third clause are removed and the first clause looses B. In set 4 the first clause is removed and the last two clauses loose the negation of C.

Set 3 becomes empty either by unit propagation or by the pure literal rule, thus the assignment of true to A, B and C satisfies the formula. On can check that this assignment satisfies the original set by observing that each of the original clauses contains at least one of these three variables. The two possible assignments to D give two satisfying full assignments.

Set 4 gives a contradiction, thus we can infer no solution here.

Set 2 has pure literal C, thus true assignment to this satisfies the last two clauses. If we use the pure literal rule then we can eliminate them, and then we have again pure literal B that finishes the search. This gives the solution assignments that have negative A and positive B and C, with D of any value. However note that by using the pure literal rule we could ignore some of the solutions, because in principle also the negative C may be part of some solutions (however this is not the case here).

If we do not use the pure literal rule, then unit B is propagated and we obtain the set 6, in which branching is applied on C and we obtain a satisfying assignment on set 7 and contradiction on set 8.

The Chaff implementation

Chaff [5] is an implementation of the DPLL algorithm that is *lazy*: it avoids some unnecessary operations. For instance when a clause is deleted, all previous updates of this clause by unit resolution are useless.

This implementation does not use recursion, but it explores the search tree in an explicit way. However the amount of backtracking data is very small: only the assignments to the variables and the branching variables need a stack. This is because the information about the watched literals (details follow) does not need to be backtracked: the algorithm runs correctly no matter which are the watched literals.

In order to avoid some unnecessary operations one uses the concept of *watched literals*. Namely, at the beginning, two literals from every clause are setup to be watched, and a clause is updated (*visited*) only when one of the watched literals must be removed. Otherwise, in the process of unit propagation the clauses are neither removed and the literals are also not deleted, but of course the necessary information (a clause or a literal should have been deleted) is used by just checking the current assignment of the corresponding variables. This method allows to avoid many operations, however the fact that a clause becomes unit is detected immediately as explained below.

We describe here a particular version of Chaff, having all the essential features. At the beginning the watched literals are established in an arbitrary way, and for each variable we instantiate a record containing:

- the truth assignment (initially unassigned);
- two lists of clauses on which the variable occurs as positive, respectively as negative literal, each

composed of two sublists (occurrence as the first, respectively the second watched literal).

Additionally the algorithm uses a *backtrack stack* for the branching variables, a *variable stack* for currently assigned variables, and a *unit queue* for the units that have been found but not yet propagated. All these are initially empty.

The main loop has three kinds of steps:

- Branching: If the unit queue is empty:
  - if all variables are assigned we have a solution, store it and go to backtracking.
  - otherwise choose an unassigned variable, assign it true, put it in the backtrack stack and in the variable stack, and go to unit propagation for the positive literal of this variable.
- Continue unit propagation: If the unit queue is not empty, choose one of the variables and propagate the literal corresponding to its current assignment.
- Backtracking:
  - If the backtrack stack is empty then stop, the search tree is exhausted.
  - Otherwise pop the variables from the variable stack and unassign them one by one, until the same variable occurs in the backtrack stack. Remove the variable from the backtrack stack, assign the variable to false, and propagate the negative literal of this variable.

Unit propagation of a literal consists in scanning the lists of clauses corresponding to the opposite of this literal and visiting each of them.

Visiting a clause consists in the following:

- If the other watched literal is assigned true, end the visit (the clause should have been removed).
- Otherwise, scan the list of the literals that are not watched:
  - If the scan ends then either there are non watched literals, or they should all have been removed: this clause is a unit - namely the other watched literal, try to assign the corresponding value to the variable.
    - If the variable has no value: assign the value found and put the variable in the unit queue.
    - If the variable already has the same value: end the visitation.
    - If the variable already has the opposite value: contradiction, end the visitation, end the unit propagation, and go to backtracking in the main loop.
  - If the scanned literal has no value: in this clause switch this literal with the watched literal that triggered the visitation and update the list of clauses that contain watched literals in the records corresponding the two variables involved, then end the visitation.
  - If the scanned literal is positive: end the visitation (the clause should have been removed).
  - If the scanned literal is negative: continue to scan (the literal should have been deleted).

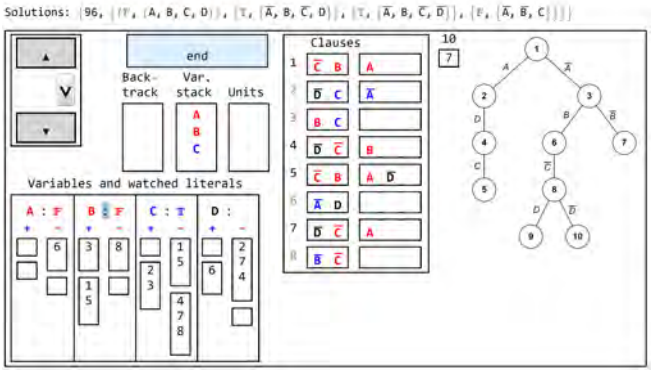


Figure 2: *Demo of the Chaff algorithm.*

We developed a Mathematica implementation of the DPLL algorithm; our implementation can generate and display the corresponding search trees, as show in Figure 1. Furthermore, in Mathematica we developed another interactive simulation tool for the Chaff algorithm, whose functionalities we illustrate next on an example.

Example: *Chaff*. Fig.2 presents the screen shot at the end of the interactive animation demonstrating the Chaff algorithm on the set of clauses shown in the table "Clauses". The box on the LHS of each clause contains the watched literals. The buttons on the LHS upper corner can be used for navigating forward and backward through the demo. The upper box indicates the state of the main loop. The search tree is displayed on the RHS only for a better understanding by the user, it is not actually necessary for the running of the algorithm. Over the box we see the final result: after 96 elementary steps we have two contradictions (nodes 5 and 7) and two satisfying assignments (nodes 9 and 10).

Teaching: After presenting the main structure of the algorithm as above, we show the students several examples as the previous ones and explain how the algorithm is applied. The pictures are produced dynamically using our tools implemented in Mathematica, and the students can themselves experiment interactively using different examples.

Exercise Generation for the DPLL Algorithm

Interaction during lectures enable the students' active participation and increases the effectiveness of learning. We made especially good experience with *bonus questions*. These are posed during the lectures (in average one

question per 45 minutes lecture). The teacher first poses a small *example* problem and gives the students some fixed time (3-5 minutes) to try to solve it. After that, the teacher presents the solution and gives the students the opportunity to ask questions. Successively, each student gets an *individual* problem to solve within a fixed time window (3-8 minutes). These individual problems should differ from the example problem just in certain parameters, such that the solutions for the individual problems can be achieved through the same steps as for the example problem. If time permits, the students should be given the possibility to discuss (face-to-face or online) or even to check each other's solutions. Solutions can be submitted on an online learning platform like Moodle. For each correct solution, the students can earn *bonus points* for the final exam. According to the short time frame per task, the posed problems should be small and cover a central concept.

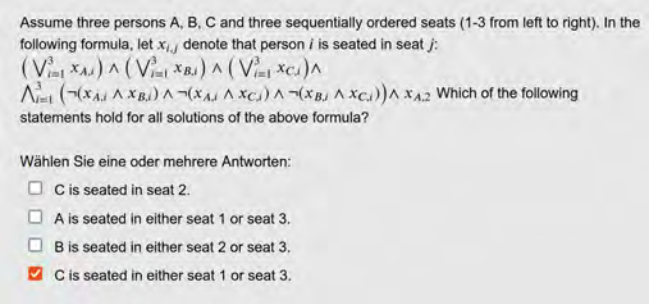
Besides bonus questions, similar problems are needed to design weekly exercise sheets, intermediate tests, and written exams. Also here, it is advantageous to have *parametric* problems, whose instances share a common solution scheme and a comparable solution effort. Optimally, also the problem sizes should be *scalable*, as the students have more time to solve the exercise sheets in home work than the bonus questions during the lectures.

The definition of such parametric problems and the automated generation of their instances is highly challenging, but once implemented, also highly rewarding. Individual problem instances assigned to the students have a strong motivating effect, they harden cheating and they can be evaluated automatically, which is very important when teaching large classes.

However, for topics related to computer algebra and symbolic computation, it is hard to come up with a clear measure that defines the quality of such parametric problems. We developed a catalogue of quality criteria, covering aspects of (i) which problems to choose, (ii), how to formulate the problems, (iii) which questions to ask and (iv) how to give feedback. We do not discuss the above catalogue here in detail, but present some aspects on a few examples.

In our project, we developed 36 such parametric problems for different topics covered in an elective lecture on *Satisfiability Checking* taught at RWTH Aachen University by Ábrahám, with 200–550 registered students in the previous years. In the following we present three of these problems that are related to the Chaff algorithm.

Example: *Propositional logic*. The following problem instance should check the understanding of the syntax and semantics of propositional logic, as well as aspects of encoding real-world problems in propositional logic.



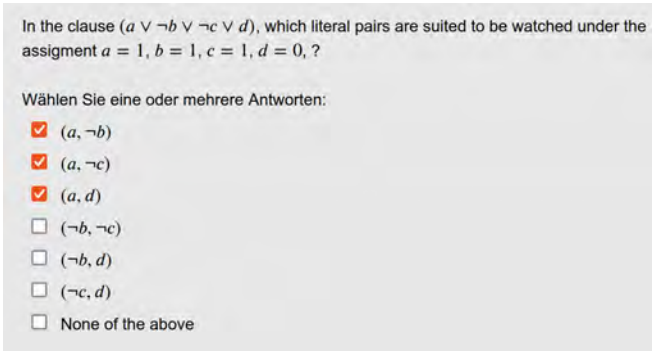
In the problem statement, we could ask e.g. to encode by a propositional logic formula the problem to seat three people in three sequentially ordered seats under certain side conditions. However, this would require a free textual answer, which we want to avoid for several reasons: it would require to fix the input syntax; reading this input syntax specification takes valuable time; it might be inconvenient to answer on cell phones; it is unclear how to evaluate incorrect syntax; the answer would not be unique, such that students might be uncertain which answer to give, furthermore some of the encodings might be easier to find than others, and the time needed to type different solutions might vary.

Instead, we present a propositional logic formula and assign meanings to its propositions, and ask to select all correct answers from a list (single/multiple choice). Though this form is more easily guessable, having a list with at least 4 choices (and thus at least 4 possible answers for single choice and 16 for multiple choice) lowers the probability of lucky guessing. If possible, these answer choices should cover regular but also corner cases, highlight important points in definitions etc.

To achieve a parametric problem, we defined 3 complex and 27 easy sub-formulas related to the seating problem, and further 32 statements formalized in natural language as well as in propositional logic. To generate a problem instance, we randomly select 2 complex and 1 easy sub-formulas and build their conjunction. Then we use a SAT solver to assert this formula and classify all 32 statements in tautologies and non-tautologies. Having done this, we randomly select four times one of the two classes and a statement from that class as choices. We assure to select both classes at least once and that there will be at least three statements in both classes. By randomly selecting first the tautology class and then an instance from that class, we assure that each possible answer is equally probable, making the guessing harder (as typically there will be less tautologies than non-tautologies under the statements).

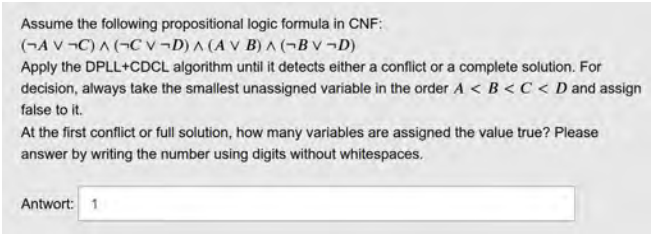
Example: *Watched literals for DPLL*. With the next parametric problem we train the understanding of the watched-literals scheme. We fix four propositions (we use a, b, c, d, but the naming could be also chosen randomly from a given set of options). For each problem instance, we generate a clause with four literals, each literal being one of the propositions with a random sign, yielding $2^4 = 16$ clause variants. Next we generate an assignment of *true*, *false* or *unassigned* to each of the propositions, offering $3^4 = 81$ possible (partial) assignments. The students should decide which literal pairs are

suited to be watched together for the given clause under the given (partial) assignment. Again, to avoid free-text answers, we use the multiple-choice format. To reduce the probability of successful guessing, we include all 7 possible literal pairs (up to ordering) as options. Note the additional option *None of the above*, to exclude the possibility that points will be awarded for not answering the question.



Example: *Boolean constraint propagation*. This task addresses the combined mechanisms of branching and propagation in the DPLL algorithm (excluding backtracking). We again fix how many propositions we want to use (here: 4) and fix their names (here: A, B, C, D), along with static orderings for propositions (here: alphabetic) as well as for the Boolean values (here: $false < true$). A further parameter is the number (here: 4) and maximal size (here: 4) of the clauses. To create a problem instance, we randomly generate the different clauses: for each clause, we randomly decide for each proposition whether it is included, and for each included proposition we select a random sign. We assure that no clause is empty, that the clauses are pairwise different and that each proposition appears at least once. We simulate the DPLL algorithm until either a conflict is detected or until it terminates without reaching any conflict. To be comparable, we process only those instances that reach a full solution after making a number of propagated assignments from a given interval (here: at least two), and dismiss all other instances (i.e. we repeatedly generate new random instances until the above condition is met).

We now need to fix a question to be asked, which provides a good indication whether the students successfully applied the DPLL algorithm, without any complicated input syntax (e.g. providing the final assignment) and without offering good chances to be guessed correctly (e.g. the value of a given proposition in the final assignment). In this example we ask for the number of true propositions in the final assignment, the possible answers being any number between 0 and 4. Using more propositions would further reduce the probability for successful guesses.



Further DPLL-related Topics

Further aspects of automated reasoning for propositional logic that we could not cover in this article but are covered by the ARC material include the conflict-driven clause learning (CDCL) [5] approach (that most state-of-the-art SAT solvers implement), parallel SAT solving techniques and other algorithms not based on the DPLL algorithm. For hands-on exercises, the standard DIMACS [6] input format for SAT solvers is introduced.

Conclusion

In this article we reported on teaching materials developed in the *ARC (Automated Reasoning in the Class)* project. These materials can be used in lectures, seminars and practical courses, as introductory material to prepare for B.Sc., M.Sc. and PhD projects, as well as in the context of training schools. We hope that the community will find these materials useful. We aim to further maintain and extend this collection in the future and are grateful for feedback and external contributions.

Acknowledgements. This work is co-funded by the Erasmus+ Programme of the European Union, project ARC: Automated Reasoning in the Class, 2019-1-RO01-KA203-063943.

References

[1] `https://arc.info.uvt.ro/`

[2] Davis, M., Hilary, P.: A computing procedure for quantification theory. J of ACM **7**(3) (1960) 201–215

[3] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (Jul 1962) 394–397

[4] Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)

[5] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Annual Design Automation Conference, Association for Computing Machinery (2001) 530–535

[6] `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`