On SAT Technologies for dependency management and beyond

Daniel Le Berre   Anne Parrain*
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
{leberre,parrain}@cril.univ-artois.fr

Abstract

SAT solvers technology is now mature enough to be part of the engineer toolbox side by side with Mixed Integer Programming and Constraint Programming tools. As of June 2008, two great pieces of software are using SAT technology to manage dependency like problems: the open source Linux distribution OpenSuse 11.0, released on June 19, 2008, integrates a custom SAT solver in their dependency manager Zypp. The new update manager of Eclipse 3.4, called Equinox p2, also uses SAT technology to resolve dependencies in their OSGi platform. The use of SAT technology in Software Product Lines has already been pointed out by several authors. We believe that the current interest for solving dependency management problems in SAT technologies opens quite interesting challenges to the SAT community. First, since those problems are met in software with interactive use, SAT engines need to solve them within seconds. Second, providing one solution is usually not sufficient: finding the best solution is usually what users want. Finally, fully supported open source or commercial SAT engines are needed for a broader adoption in the software engineering community.

1 Introduction

The success of SAT technology in Electronic Design Automation (EDA), in particular in the area of Bounded Model Checking [6] has pushed forward the development of SAT solvers. They are now being used routinely as lightweight constraint programming solvers. The performance breakthrough due to the Chaff SAT solver[19], and the design some years later of the minimalistic Minisat [13] allowed the availability of numerous very efficient SAT engines in various languages. One of the main advantage of SAT solvers against more traditional constraint programming solvers is its simple unified input format (Dimacs [15]) that allows SAT solvers to be used just as black boxes, or as SAT components in a more software engineering vocabulary.

If many users of SAT solvers are currently coming from EDA, the use of SAT technology is currently growing in software engineering. Recent works in software design analysis clearly focus on efficient translation into SAT [23] and take advantages of new features developed in SAT solvers (e.g. Unsat cores [22]) to improve both user experience and scalability. In Software Product Lines, SAT solvers can be used to decide whether a product configuration is safe or not and how to implement the product configuration [3, 21]. Related work includes the use of CSP and BDD approaches to tackle the same problems [4, 5]. The EDOS project [16, 11] found in SAT technology a good mean to validate the package dependencies for Linux distributions, i.e to answer the question:“Are all the packages of that distribution installable?”. In the same spirit, OPIUM [24] is a dependency manager for the Linspire linux distribution based on pseudo boolean solvers, one of the numerous extensions to SAT.

As a consequence of such research work on dependency management, two great pieces of software are using SAT technology to manage dependency like problems: the open source Linux distribution OpenSuse 11.0 [1], released on June 19, 2008, integrates a custom SAT solver in their dependency manager ZYpp. The new update manager of Eclipse 3.4, called Equinox p2, also uses SAT technology to resolve dependencies in its OSGi platform [2]. Finally, another famous framework for Java users, Maven, decided recently[3] to use SAT technologies to resolve their package dependencies.

*This work is supported in part by the European research project number 214898 Mancoosi (www.mancoosi.org).

[1]http://en.opensuse.org/OpenSUSE_11.0
[2]http://wiki.eclipse.org/Equinox_P2_Resolution
[3]http://jira.codehaus.org/browse/MARTIFACT-20

2 Dependency decision problem

Dependencies between packages can be easily modeled using propositional logic. For instance, package A depends on package B and package C can be expressed by the logical formula $A \rightarrow B \wedge C$ which in turn can be expressed by the two clauses $A \rightarrow B$ and $A \rightarrow C$ (or $\neg A \vee B$ and $\neg A \vee C$). If all the dependencies are requirements of a conjunctive form, even if incompatibilities between packages are expressed, the resulting SAT problem is made of Horn clauses, i.e. clauses containing at most one positive literal, thus is solvable in linear time [10].

The dependency problem becomes interesting when a given feature can be provided by several artifacts: this is the case for instance of several versions of the same OSGi bundle[4] in Eclipse. Sometimes, the same feature is provided by different packages, depending on their origin: to install latex in a linux distribution, one can use for instance texlive-latex or tetex-latex. In that case, we would express such dependency by something like $latex \rightarrow texlive\_latex \vee tetex\_latex$ which is no longer a Horn clause. Thus the dependency problem becomes NP-complete [8].

However, SAT solvers can nowadays solve some instances of the SAT problem with as many as millions of variables and clauses while they were still unable to solve a custom crafted 117 variables only SAT instance during the SAT 2007 Solver competition [5]. So, while there is no warranty that a SAT solver can solve efficiently SAT instances resulting from the translation of a "real" problem into SAT, it is often the case that such solvers perform well on those instances. From our own experience, the dependency decision problem can be easily solved using modern SAT solvers.

3 From satisfaction to optimization

However, the dependency problem is often *under-constrained*, which means that there is usually a lot of possible solutions. And not all those solutions are usually equally good. For Linux distributions for instance, one could take into account the number of packages to install, or their size, etc in order to propose an installation with the minimum number of packages or an installation with the smallest footprint on the hard drive. Those criteria can be easily modeled in an optimization framework by an objective function to minimize. Such objective function is of the form $\sum_{i=1}^{n} a_i x_i$ where a_i is an integral coefficient and x_i is a propositional variable where true is denoted by value 1 and false is denoted by value 0.

[4]OSGi is the component based model component chosen by Eclipse for its plug-in architecture. See `http://www.osgi.org/` for details.

[5]`http://www.satcompetition.org/2007/`

If one wants to minimize the number of installed packages, all a_i will be 1 and all x_i will correspond to the propositional variables encoding packages to install. If one wants to minimize the size of the installed packages, then the a_i will encode the size of each packages in bytes for instance.

In the case of SPL, the objective function could encode to install as many features as possible, to look for the cheapest or the most expensive product, etc.

Adding such objective function to a set of clauses creates an instance of the Binate Covering problem [9]. Such problem has already been studied in EDA for logic synthesis (to minimize the number of components needed to perform a given operation). From a complexity theory, that problem is NP-hard, which means that is is at least as hard as SAT.

The binate covering problem can be seen as a very specific case of an Integer Linear Program in which case efficient ILP frameworks exist (e.g. CPLEX). However, it is currently not clear if ILP solvers are the right approach to tackle those problems.

Indeed, ILP restricted to boolean variables has been also a recent area of research in the SAT community, under the generic name "Pseudo Boolean problems", inherited from the very first work on that subject [1, 2]. A competition of Pseudo Boolean solvers has been organized to assess the efficiency of existing solutions[18]. Among the current best ones, one can note the pure SAT approach of Minisat+[12], the hybrid approach Pueblo [20] (used in the tool OPIUM) or the modern branch-and-bound Bsolo [17]. Those solvers have been compared to the Gnu Linear Programming toolkit, and in many cases performed better. No comparison with commercial ILP solvers has been done yet.

In Artificial Intelligence, the binate covering problem is sometimes presented as a so called "Weighted partial MAX-SAT problem": the clauses of the original binate covering problem are called hard clauses, i.e. they must be satisfied. The objective function is encoded by weighted unit clauses, whose weight is a_i and whose literal is $\neg x_i$. If the weight is the same for all the soft clauses, the problem is called "partial MAXSAT". Since 2006, there is an annual competition of MAXSAT solvers. The interest on MAXSAT solvers is currently growing so MAXSAT solvers will be yet another way to tackle the binate covering problem in the future.

SAT solvers are currently efficient enough to solve SAT instances mapping real useful problems. Regarding solvers for SAT extensions like Pseudo Boolean or MAXSAT, the picture is less clear. Furthermore, we have seen that the so-called binate covering problem can be solved in many-ways: the best solution is yet to be determined.

The evaluation of the solvers is currently done on the assumption that the solvers are used in batch mode, i.e. that they can take several minutes, if not hours, to answer. Indeed, it is often the case in the area of model checking that the engineer writes its model during the day and let the SAT solver to check it during the night. In the SAT competition for instance, the timeout used in the industrial category is set to 1200 seconds in the first stage, and 10000 seconds in the second stage. As a consequence, the use of SAT solvers in interactive tools such as Eclipse or Linux update managers, or a Feature Model Editor such as FeatureIDE or FAMA requires specifically tailored solvers (see e.g. custom SAT engines in EDOS (debian) or OpenSuse.

Finally, the notion of quality of the solution is also a hot topic in that case: it is unlikely that a solver can efficiently solve all those NP-hard problems within a second, so non optimal solutions need to be returned after that time. We enter here in the area of anytime algorithms, for which local search algorithms are usually pretty good [14].

4 When modeling matters

The expected results mentioned in the previous section were simple, and global. However, in real applications, the user preferences are usually manifold and expressed in a local manner. Take for instance the case of the use of SAT technology in Eclipse p2: one expected result of the objective function is to make sure that most recent versions are installed preferably to older ones. If there is only one package, this is not a problem. However, as soon as there is more than one package, we enter the area of multi-objective/criteria optimization.

Indeed, suppose that package A is available in three versions: A_1, A_2, A_3. Suppose that package B is available in two versions: B_1, B_2. Suppose that package X depends on A and B. The best solution would be to pick the latest versions, B_2 and A_3. However, they are incompatible. Is it better to take A_2 and B_2 or B_1 and A_3? There is maybe no way to answer that question. In that case, an automated solution to solve that dependency problem will answer one of them, without any assumption that can be made on the solution. If one can add a preference between A and B, then we can discriminate between the two solutions and favor the first solution for instance if B is more important than A or the second one if A is more important than B.

Artificial Intelligence and Operational Research have solutions to deal with such kind of problems. In the specific case of Eclipse p2, the preferences on the different plugin versions could be modeled for instance using the Qualitative Choice Logic [7].

5 Research toys or real tools?

There are numerous SAT/Pseudo Boolean/MAXSAT solvers out there, in various languages, with various features. However, very few are really supported in the sense that there is no clear way to get help, enter bug reports, etc. And very few are really open source: many solvers have a license restricting their use for research purpose only. Most of them are not full featured in the sense that if they all allow to solve the SAT decision problem, very few support also proof logging, minimal unsat core, model enumeration, model counting, optimization support, support for multi-core processors, etc. In that sense, users of SAT technology must first define all the features they really need in order to choose the right solver (or the right solvers, since it might be just impossible to find a solver with all needed features). In that sense, SAT solvers are still research toys.

On the other hand, various products (from both academia and private companies) are now based on currently available SAT solvers. So the same solvers can be considered as real tools too.

The main current issue is certainly the input format used in the SAT community: the Dimacs format created for the Second Dimacs Challenge[15]. It is a simple integer based input format very convenient for SAT solver developers. It's simplicity is one key element of the incredible evolution of SAT solvers: everybody can easily read or produce SAT instances using that format, in any language. As a consequence, SAT solvers became quickly "black-boxes" fed using Dimacs formatted SAT instances. It was thus easy to compare the behavior of several solvers on the same benchmarks, organize sat competitions, etc. However, it is not a nice input format for people willing to try SAT technology: one needs to design an abstraction layer between its problem and the Dimacs format or the SAT solver first. The audience of SAT technology growing, a more user friendly input format is now necessary.

6 Conclusion

We have seen that SAT technology receives currently a lot of attention in the software engineering community, especially in the area of dependency management, that is also a concern inherent to Software Product lines. We have seen that the basic decision problem related to dependency management is in practice easily solvable with current state-of-the-art SAT solvers. However, dependency management in real software is likely to be better modelled as an optimization problem called binate covering problem for which numerous solutions exist. We currently do not know the best approach for solving binate covering instances encoding the dependency problem. Another issue with dependency management lies in the fact that the problem is likely to be under-constrained, i.e. it is likely that the problem admit several equivalent solutions for the solver. As a consequence, it will be very difficult to control the answers provided by the solver, and to offer some warranties about those solutions. It must be clear for users of such technology that the most important part of the work is to properly express their problem in terms of preferences among possible solutions, then to deduce from those preferences the objective function of their binate covering problem. Finally, the use of solvers in interactive tools is likely to change the solvers landscape since most of them are currently designed to be used in batch mode.

References

[1] P. Barth. Linear 0-1 inequalities and extended clauses. Technical Report MPI-I-94-216, Max-Plank-Institut fr Informatik, Saarbrücken, Germany, 1994.

[2] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Max-Planck Institut fr Informatik, Saarbrücken, 1995.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[4] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[5] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In R. Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[7] G. Brewka, S. Benferhat, and D. L. Berre. Qualitative choice logic. *Artif. Intell.*, 157(1-2):203–237, 2004.

[8] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[9] O. Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.

[10] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[11] The edos project. http://www.edos-project.org.

[12] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:1–26, 2006.

[13] N. E. en and N. Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[14] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[15] D. Johnson and M. Trick, editors. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[16] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.

[17] V. Manquinho and J. Marques-Silva. On using cutting planes in pseudo-boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:209–219, 2006. Research Note.

[18] V. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (pb'05). *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:103–143, 2006.

[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[20] H. M. Sheini and K. A. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:165–182, 2006.

[21] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[22] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[23] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[24] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.