Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. This note serves as system description for our SAT solvers that entered the SAT Race 2010 affiliated to the SAT conference 2010.

Overview

To the main track of the SAT Race 2010 we submitted a new SAT solver called Lingeling, and also new versions of our already existing SAT solvers PicoSAT and PrecoSAT. Lingeling was also submitted in its parallel version Plingeling to the multi-threaded track.

PicoSAT 935

PicoSAT version 935 is not much different from already described versions [3, 4], except that it puts more effort into failed literal detection. Failed literal probing is interleaved with search and is the only pre-processing technique used in this version of PicoSAT. We have recently improved trace, core and proof generation, which however should not have any influence on pure solving speed. PicoSAT remains our platform for proof and core generation, and is also the only solver submitted, that supports incremental SAT solving.

PrecoSAT 570

PrecoSAT version 236 was the winner of the application track of the SAT competition in 2009. The new version 570 submitted to the SAT race 2010 differs in the following aspects. First, the hash table for storing binary clauses was removed. Even though, it allows constant time detection of equivalent literals, it presents a non negligible space overhead and even more is subsumed by equivalence reasoning through decomposition into strongly connected components anyhow.

We switched to a Glucose style reduce strategy [2] for garbage collecting less important learned clauses. As Glucose we never remove binary learned clauses. The "glue" value, i.e. the number of levels of variables that ever made a specific learned clause conflicting or forcing, is represented with a 4-bit value and all glues above 15 are assigned glue value 15. Another technique we added is on-the-fly self-subsuming resolution [7] during conflict analysis and pre-processing.

Technical Report 10/1, August 2010, FMV Reports Series
Institute for Formal Models and Verification, Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria

Our new results on Blocked Clause Elimination (BCE) [8] are also incorporated. We have a new separate BCE pre-processing phase, which is scheduled before variable elimination (VE) [5]. In practice BCE turns out to be much cheaper than VE. Initially we actually run BCE until no more blocked clauses can be removed before switching to VE.

We have also implemented inverse arcs [1], dynamic subsumption and strengthening of learned clauses with learned clauses, as well as some simple form of autarky pruning. However, these last three techniques had a negative impact on the number of solved instances from the SAT competition 2009, and are therefore disabled in the version submitted to the SAT race 2010.

Lingeling 271

PrecoSAT was developed as a prototype to explore interleaving search and preprocessing. The success in the SAT competition 2009 showed, that this approach can be useful. Lingeling is a new solver that builds on the same principle. Lingeling consists of roughly 10 KLOC of C (PrecoSAT 6 KLOC C++).

The data structures used in Lingeling are engineered to use much less space than those in PrecoSAT. While PrecoSAT is much faster compiled in 32-bit mode, the opposite is true for Lingeling. The space reduction is achieved by three techniques. First, binary and ternary clauses are represented implicitly through *occurrence lists* [3] as in Siege [11]. Large clauses with four or more literals are stored separately on *literal stacks*.

Second, the literals of these large clauses with four or more literals are referenced in the occurrence lists not with pointers but through their stack position. On 64-bit machines this technique saves half the space required to store references and only needs 4 instead of 8 bytes.

Third, occurrence lists are implemented as integer stacks, which reside in one large integer stack, called *occurrences stack*, and are again referenced through stack position and their size, i.e. two integers, instead of the usual three pointers used in a typical stack implementation. On 64-bit machines, this gives a size reduction from 24 to 8 bytes, on 32-bit machines a size reduction from 12 to 8 bytes. The occurrences stack uses a fast specialized memory allocator and is defragmented periodically with a moving garbage collector.

The occurrence list of a literal contains one integer for an occurrence in a binary clause and two integers for an occurrence in a ternary clause. For larger clauses there are also two integers: the first integer is a blocking literal [12], the second integer is the stack offset of the literals of the clause on the literal stack. Through bit-stuffing the first integer of an occurrence also encodes its type (binary, ternary, large) and also contains an additional bit for separating redundant and irredundant clauses. The second integer for large redundant clauses has 4 bits to encode the glue of the referenced clause as the literals of redundant clauses are kept in separate stacks: for each glue value there is one stack of literals. Redundant clauses with the largest glue value of 15 are not stored permanently. They are removed during backtracking.

As in PrecoSAT we interleave various pre-processing algorithms with a standard CDCL algorithm [10]. The pre-processing techniques that are both implemented in PrecoSAT and Lingeling include failed literal probing, lazy hyper binary resolution, decomposition into strongly connected components (SCCs) for equivalence reasoning, BCE and VE.

In failed literal probing we do not merge equivalent literals with a union-find data structure as in PrecoSAT. Lingeling only relies on SCC decomposition for extracting equivalences. Equivalent literals are substituted in subsequent garbage collection phases. The solver maps external variable indices to internal variable indices. Fixed, eliminated and substituted internal variable indices are recycled. The map from external to internal variable indices is updated accordingly during garbage collection.

VE in Lingeling does not extract gates as in PrecoSAT, but generates an irredundant prime cover of the clauses generated in one variable elimination step, if the number of distinct variables that occur in clauses of the eliminated variable x is small. Currently an irredundant prime cover is generated if there are not more than 11 other variables beside x. Of course a variable is only eliminated if the number of new clauses is smaller or equal to the number of old clauses.

Additionally, we have implemented clause distillation [9], transitive reduction of the binary implication graph, and a new unpublished technique that removes hidden tautologies. All three techniques try to remove or shrink clauses and subsume a basic form of failed literal probing.

In failed literal probing, we randomly probe literals and switch back to search if the number of visited clauses reaches a limit that is computed based on the number of clauses visited in the CDCL search so far. The same technique is used for all the other pre-processing techniques, except for BCE and VE which schedule elimination attempts based on variable occurrences. In BCE and VE the number of attempted resolutions is limited instead of the number of visited clauses.

Plingeling 271

Plingeling is the multi-threaded version of Lingeling using Pthreads. The number of worker threads is specified on the command line. For each worker a separate SAT solver instance is generated by the boss thread, which also reads the DIMACS file sequentially. The solver instances of worker threads differ in the choice for the random number seed, the effort allowed in different pre-processing algorithms, and the default ordering for variable phases and indices, which are used for initialization and tie-breaking in decision heuristics. The solver instances do not share clauses, except that they send generated unit clauses to the boss thread, as well as receive unit clauses from the boss thread in regular intervals. They also check for early termination of other workers. Thus Plingeling is a portfolio-based parallel SAT solver such as ManySAT [6], but only exchanges unit lemmas.

This functionality is implemented through three call-back functions *Produce*, *Consume*, and *Terminate*, which can be registered during initialization of a SAT solver instance. The core library does not depend on Pthreads and the same object code is used, both for the sequential front-end Lingeling as well as for the multi-threaded front-end Plingeling.

The boss maintains a global unit table, which is lazily synchronized among all workers. Since *Produce* is called for every top-level assignment, the call-back function saves produced units in a thread local buffer, which does not need any synchronization, unless the buffer capacity is exhausted, in which case the thread local buffer is flushed to the global unit table. This also happens before *Consume* returns the interval of new units flushed to the global unit table by other threads.

Both *Consume* and *Terminate* are called from inner loops of most pre-processing algorithms in regular intervals and of course also in the main CDCL loop. The intervals are measured in the number of solver steps, which includes the number of resolutions in pre-processing and propagations during search.

References

1. G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In H. K. Büning and X. Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.
2. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *IJCAI*, pages 399–404, 2009.
3. A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
4. A. Biere. P{re,i}coSAT@SC'09. In *SAT 2009 Competitive Event Booklet*, 2009. `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`.
5. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
6. Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6:245–262, 2009.
7. H. Han and F. Somenzi. On-the-fly clause improvement. In O. Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 2009.
8. M. Järvisalo, A. Biere, and M. Heule. Blocked clause elimination. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.
9. H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electr. Notes Theor. Comput. Sci.*, 119(2):51–65, 2005.
10. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability.* IOS Press, 2009.
11. L. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.
12. N. Sörensson. MS 2.1 and MS++ 1.0 SAT Race 2008 editions.