LINGELING and Friends Entering the SAT Challenge 2012

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

This note describes our SAT solvers submitted to the SAT Challenge 2012, all based on the same LINGELING backend.

I. LINGELING

Compared to the version submitted to the SAT competition 2011 and described in [1], we removed complicated algorithms and features, which did not really have any observable impact on the run-time for those benchmarks we tried. In particular, various versions of distillation inprocessors were removed.

Regarding inprocessing [4], there are two new probing variants. One is called *simple probing* and tries to learn hyper binary resolutions eagerly. The other variant is based on *tree-based look-ahead*, which is a simplified version of the implementation in March [2]. These two techniques are complemented by *gaussian elimination* and a new *congruence closure algorithm*, which both use extracted gates to generate and propagate equivalences.

We also switched to one merged inprocessing phase, called *simplification*, where all inprocessors run one after each other, instead of allowing each inprocessor to be scheduled and interleaved with search individually.

Furthermore, for most inprocessors we have now a way to save the part of the formula on which the inprocessor did not run until completion (actually currently only "untried variables"). In the next simplification phase, the algorithm can be resumed on that part, such that eventually we achieve the same effect as really running the various algorithms until completion. Previously we used randomization to achieve a similar effect. This technique also allowed us to remove certain limits, such as the maximum number of occurrences or the maximum resolvent size in variable elimination.

We moved to an inner-outer scheme for the size of kept learned clauses, also called *reduce schedule*. The inner scheme follows the previously implemented LBD resp. glue based scheme as in Glucose. As in the previous version the solver might switch to activities dynamically, if the glue distribution is skewed. The outer schedule is Luby controlled and resets the learned clause data based limit to its initial size. This idea is particularly useful for crafted instances.

Another new feature is to occasionally use the opposite of the saved phase for picking the value for the decision variable. These *flipping* intervals start at the top-level and while flipping is enabled the phases of assigned variables are not saved (as in probing) in order not to counteract the effect of the phase saving mechanism.

The exponential VSIDS scheme of MiniSAT has been replaced by a new variant of a *variable-move-to-front* strategy with multiple queues ordered by priority. This seems to be at least as effective as the previous scheme, but updating and querying the queue turns out to be substantially faster.

In general we simplified internal data structures with the hope to make the code a little bit more accessible.

II. PLINGELING

The parallel version of LINGELING has not changed much. Still only units and equivalences are shared among multiple solver instances. Actually, we even removed most options previously set differently for each instance of the LINGELING core library, except of course for the seed of the random number generator and in addition kept the different choices for the default phase. Last but not least we only use one instance during parsing. This first solver instance is cloned after preprocessing. This reduces the memory usage on certain instances considerably, particularly, since we can stop cloning as soon too much memory is already in use.

III. CLINGELING

CLINGELING is based on our new *Concurrent Cube and Conquer* (CCC) approach [5]. This is an extension of our previous *Cube and Conquer* (CC) technique [3]. The new idea is to run a CDCL solver and a look-ahead solver *concurrently*.

CLINGELING uses the new `lglfork` API call, provided by the LINGELING library for copying a solver instance. It additionally uses a global LINGELING instance with assumptions to partially simulate what ILINGELING does in the original CC approach (but in an interleaving fashion and with only one worker thread). The look-ahead literal is computed by the tree-based probing algorithm discussed already above.

In the current version of CLINGELING, the CDCL part and the look-ahead are interleaved, and thus not really run in parallel. Which also means that there is no benefit from multi-core machines as in the original CC (and CCC) approach. But compared to CC we can find better cut-off limits for switching from look-ahead to pure CDCL with inprocessing this way, which is one of the motivations behind CCC.

Another drawback of this online approach is that the global solver, can not determine up-front the set of variables that can be eliminated in pre- and inprocessing.

IV. FLEGEL

FLEGEL can be seen as a poor man's version of CLINGELING. It uses the `fork` system call for backtracking and in principle such a front-end should be easy to build for any SAT library, which can produce a look-ahead decision. It just runs preprocessing and a limited amount of search of the CDCL solver (with inprocessing) at each node before calculating the next look-ahead literal. Then the process is forked. The child process adds the look-ahead literal as unit and continues the same procedure recursively. Currently parents wait for their child to terminate, before adding the negation of the original look-ahead literal as unit. So even FLEGEL uses as many processes as active search nodes, i.e. the height of the search tree, no parallelism is used.

V. TREENGELING

TREENGELING is our latest SAT solver with LINGELING backend and tries to capture the positive aspects of PLINGELING, FLEGEL and CLINGELING and actually to some extend also ILINGELING [3]. To simulate *forking* in FLEGEL we implemented a *clone* function `lglclone` as part of LINGELING. This function in contrast to `lglfork`, which is used in CLINGELING, generates an identical behaving solver instance, instead of just copying clauses and assumptions. In the context of TREENGELING this allows to additionally copy saved phases, variable queue, etc., so all the state, from the original solver instance to the clone.

The clone has all information for reconstructing a solution. So there is no need to propagate the solution back by merging a forked copy with the `lgljoin` API call, as it is necessary if the copy was generated by `lglfork`, e.g., as in CLINGELING.

Up to this point TREENGELING is very similar to FLEGEL. However, since we have all the cloned solver instances in one address space (as in CLINGELING) we can easily use multiple threads to run the updated clones in parallel. TREENGELING is a parallel solver and uses the infra-structure for parallel execution also used in PLINGELING and ILINGELING.

Solver instances are stored in nodes and we start with one single solver instance with the original formula, which is then first simplified in a *simplification* phase. If there are less open nodes than a predefined limit, a decision literal is selected by tree-based look-ahead from the smallest solver instance in a *look-ahead* phase. This instance is cloned and saved in a new node during the *splitting* phase. The decision is added as unit to the clone and negated to the original solver instance.

Lookahead and then splitting existing solvers this way is actually performed in parallel. After splitting, the solvers of open nodes are run for a certain conflict limit in a *search* phase. If a solver instance finds a solution it is printed and the whole search terminates. If the solver instance of one node proves unsatisfiability, it is closed. After all solver instances terminated their limited search, closed nodes are flushed. If no more nodes remain, the search terminates with proving unsatisfiability.

Then the conflict limit is updated in an *update* phase. If a node was closed in this round the limit is decreased and otherwise increased, both in a geometrical way. Due to the potential exponential increase of the conflict limit over multiple rounds, TREENGELING with one worker behaves very similar to the base LINGELING solver. It does not behave identically though, as it is the case for PLINGELING with one worker. The pseudo-code of this procedure looks as follows:

```
search(lim);
while (!flush) {
  simp; lookahead; split; upd(lim); search;
}
```

Sub-procedures work in parallel, e.g. simplification (`simp`) is run in parallel for the minimum of still open nodes and number of cores. The default is to use both the number of cores as maximum limit on the number of open nodes and worker (threads). For multiple workers units are added and tree-based lookahead has to be performed, but otherwise, since the workers run in parallel independently, the (wall-clock time) performance is not expected to be much worse than for plain LINGELING. Preliminary experiments justify this claim. This does not seem to hold for CLINGELING nor FLEGEL.

TREENGELING is deterministic, e.g. always traverses the same search space and produces the same number of conflicts (actually only for unsatisfiable instances) etc., as long the maximum number of active nodes stays the same and the same memory limit is used. The number of threads available to work in parallel during simplification, search or lookahead does not influence the search. With more available cores, more threads can be run in parallel, without run-time penalty.

However, in order to use more threads, more active nodes have to exist in parallel. In preliminary experiments we unfortunately saw a negative effect on wall-clock time, if more open nodes are used than available cores (except when hyper-threading is available). Thus the effectiveness of this approach w.r.t. speed-up is not really understood yet. On processors with four cores and no hyper-threading, TREENGELING with a maximum of four open nodes, is expected to perform slightly better than PLINGELING, e.g., substantially, but not dramatically better than plain LINGELING.

VI. ACKNOWLEDGEMENTS

This work heavily depends on research results obtained with my collaborators Marijn, Matti, Oliver, Siert, and Peter, and of course the whole SAT community. A full list of proper references can be found in the following papers.

REFERENCES

[1] Armin Biere. Lingeling and friends at the SAT Competition 2011. FMV Report Series Technical Report 11/1, Johannes Kepler University, Linz, Austria, 2011.

[2] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In *SAT 2004 Selected Papers*, volume 3542 of *LNCS*, pages 345–359. Springer, 2005.

[3] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT solvers by lookaheads. In *Proc. HVC 2011*, 2012. To appear.

[4] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *Proc. IJCAR'12*. To appear.

[5] Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent Cube-and-Conquer. Submitted.