System Description of Candy Kingdom – A Sweet Family of SAT Solvers

Markus Iser* and Felix Kutzner†
Institute for Theoretical Computer Science, Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: *markus.iser@kit.edu, †felix.kutzner@qpr-technologies.de

***Abstract*—Candy is a branch of the Glucose 3 SAT solver and started as a refactoring effort towards modern C++. We replaced most of its custom lowest-level data structures and algorithms by their C++ standard library equivalents and improved or reimplemented several of its components. New functionality in Candy is based on gate structure analysis and random simulation.**

I. Introduction

The development of our open-source SAT solver **Candy**[1] started as a branch of the well-known **Glucose** [?], [?] CDCL SAT solver (version 3.0). With Candy, we aim to facilitate the solver's development by refactoring the Glucose source code towards modern C++ and by reducing dependencies within the source code. This involved replacing most custom lowest-level data structures and algorithms by their C++ standard library equivalents. The refactoring effort enabled high-level optimizations of the solver such as inprocessing and cache-efficient clause memory management. We also increased the extensibility of Candy via static polymorphism, e.g. allowing the solver's decision heuristic to be customized without incurring the overhead of dynamic polymorphism. This enabled us to efficiently implement variants of the Candy solver. Furthermore, we modularized the source code of Candy to make its subsystems reusable. The quality of Candy is assured by automated testing, with the functionality of Candy tested on different compilers (Clang, GCC, Microsoft C/C++) and operating systems (Linux, Apple macOS, Microsoft Windows) using continuous integration systems. In what follows, we present the optimizations we implemented in Candy and describe two variants of the solver.

II. Clause Memory Management

Unlike Glucose, we use regular pointers to reference clauses in Candy. To reduce the memory access overhead, we introduced a dedicated cache-optimized clause storage system. To this end, we reduced the memory footprint of clauses by shrinking the clause header, in which only the clause's size, activity and LBD values as well as a minimal amount of flags are stored. For clauses containing 500 literals or less, our new clause allocator preallocates clauses in buckets of same-sized clauses. Clauses larger than 500 literals are individually allocated on the heap. Buckets containing small clauses are regularly sorted by their activity in descending order to group frequently-accessed clauses, thereby concentrating memory accesses to smaller memory regions. Moreover, the watchers are regularly sorted by clause size and activity.

[1] https://github.com/udopia/candy-kingdom

III. Improved Incremental Mode

We enabled several clause simplifications in Candy's incremental mode that had been deactivated in Glucose's incremental mode. Also, certificates for unsatisfiability can be generated in incremental mode for sub-formulas not containing assumption literals. This is achieved by suppressing the emission of learnt clauses containing assumption literals as well as the output of the empty clause until no assumptions are used in the resolution steps by which unsatisfiability is deduced.

IV. Inprocessing

We improved the architecture of clause simplification such that Candy can now perform simplification based on clause subsumption and self-subsuming resolution during search. The original problem's clauses are included as well as learnt clauses that are persistent in the learnt clause database, i.e. clauses of size 2 and clauses with an LBD value no larger than 2.

V. Gate Structure Analysis and Applications

Candy provides modules for gate extraction [?] and random simulation [?]. Detemerminized random simulation is used on gate structure extracted from SAT problems to generate conjectures about literal equivalences and backbone variables. The Candy solver variant Candy/RSAR uses these conjectures to compute and iteratively refine under-approximations of the SAT problem instance [?]. The Candy solver variant Candy/RSIL uses branching heuristics based on implicit learning [?], [?] to stimulate clause learning by violating extracted conjectures about variable equivalencies where possible, otherwise using the VSIDS branching heuristic. Candy/RSIL includes the sub-variants Candy/RSILv, with which the probability of implicit learning being used is successively halved after a fixed amount of decisions, and Candy/RSILi, with which usage budgets are assigned to each of the two implications represented by a conjecture. If budgets are assigned, such an implication can be used for implicit learning only if its budget is nonzero, and an implication's budget is decreased every time it is used for implicit learning.

Employing implicit learning has proven particularly efficient for solving miter problems with general-purpose SAT solvers [**?**]. Candy includes fast miter problem detection heuristics enabling implicit learning to be enabled only for SAT problem instances detected to be miter encodings within a given time limit. For a general description of miter problems, see e.g. [**?**].