xMaple* Technical Documentation

Jonathan Chung

September 12, 2023

1 Overview

The overall design of the Extended Resolution (ER) solver framework is described in the related thesis chapter [2]. The `xMaple*` family of solvers are based on refactored versions of the `MapleSAT` and `MapleLCM` solvers, where functionality has been moved out of the `Solver.h` and `Solver.cc` files into dedicated classes. The most up-to-date version of the code is implemented on `MapleLCM`, with an older implementation on `MapleSAT`. The overall file structure is similar for both the `xMapleSAT` and `xMapleLCM` solvers. The ER-based solver is implemented as a subclass of the solver class, and the core CDCL solver (source code in the `core` subdirectory), which is not based on ER, can be compiled without the ER code base (`er` subdirectory). The ER code requires the `ExtDefMap` data structure, which is defined in the MiniSAT template library (`mtl` subdirectory). Different versions of the solvers can be compiled by providing different arguments to the `Makefile` in the `xMapleLCM` directory.

Solver	Compilation command
`MapleLCM`	`make maplelcm`
`xMapleLCM_random`	`make xmaplelcm_rnd_rng`
`xMapleLCM_common`	`make xmaplelcm_sub_lbd`
`xMapleLCM_ler`	`make xmaplelcm_glucoser`

Table 1: Makefile options to compile each of the different solvers.

The overall design of `xMaple*` uses user-defined callback functions as heuristics to control clause selection, variable definition, variable substitution, and variable deletion. The desired callback functions are selected at compile time and bound to function objects which are used throughout the ER-based solver. Each of the different ER-based solvers in Table 1 is implemented simply by providing different heuristic functions.

2 Solver Files/Classes

2.1 Refactored MapleLCM

The `MapleLCM` solver has been refactored, separating related functionality into different classes to make it easier to understand and maintain. Although the classes still interact with one another, this grouping of functionality makes it easier to understand the overall solving algorithm, which is particularly useful for developing an ER-based solver. Notably, the code for maintaining scores and data structures has been moved to "event handlers" within each of the different classes (search for `::handleEvent` in the code base), which separates the implementation of the heuristics from the main solver loop. The core classes (i.e., the ones in the `core` subdirectory) are described in this section in alphabetical order.

`AssignmentTrail` This class keeps a record of the variable assignment order, the decision levels, and the reasons for each of the assignments.

`BranchingHeuristicManager` This class selects decision variables for branching and manages data structures for branching heuristics.

`ClauseDatabase` This class manages clauses and clause deletion.

`ConflictAnalyzer` This class is responsible for analyzing conflict graphs to generate and simplify learnt clauses.

`ProofLogger` This class handles the implementation details associated with printing DRUP proofs. It contains methods for both binary and ASCII output formats.

`PropagationQueue` This class represents the BCP propagation queue. It supports three different propagation modes (see Priority BCP work [2] for descriptions): Immediate BCP, Delayed BCP, and Out-of-Order BCP.

`RandomNumberGenerator` This class contains a small number of utility functions for generating random numbers using a given seed value.

`RestartHeuristicManager` This class determines when the solver should restart and manages data structures for restart heuristics.

`Solver` This is the main solver class. It provides the interface for adding clauses, solving the given problem instance, and outputting the solution. It has ownership of all the other solver components.

`UnitPropagator` This class is responsible for actually performing BCP using the `PropagationQueue`. It is also responsible for maintaining the watchers.

2.2 xMapleLCM

The additional components for xMapleLCM are implemented as two classes in the `er` subdirectory, with an additional header file for data types and a separate source file for user-defined heuristics. The `ExtDefMap` data structure is implemented as a header file in the `mtl` subdirectory.

`ERManager` This class contains the major variables and data structures for ER, including the data structures for extension variable definitions, buffers to store the clauses and definitions for each of the ER components, ER heuristics and their configuration variables, and infrastructure for measuring the amount of time spent on each of the different components. Many of the functions in the public API of this class are wrapper functions for calling the user-defined heuristic functions, measuring the amount of time spent on each section of computation, and piping data from one function to another.

`ERSolver` This class augments the base solver with extended resolution capabilities. In particular, it extends the `Solver` base class and overrides the `search` and `solve_` methods, adding ER components to the main search loop. This class has ownership of the `ERManager` object.

`ERTypes.h` This header file contains the `ExtDef` struct (which is an intermediate representation for variable definitions), definitions of preprocessor macros for configuring the ER solver, and aliases of function objects for each of the different user-specifiable ER heuristics. The `ExtDef` struct contains a positive literal x representing the extension variable, two basis literals a and b such that $x \leftrightarrow a \vee b$, and a list of clauses to learn in addition to the clauses encoding the definition $x \leftrightarrow a \vee b$.

`ERUserHeuristics.cc` This `.cc` file contains all the implementations of the user-specifiable ER heuristics (search for `ERManager::user_ext`). These functions are declared as methods for the `ERManager` class to give them access to the class's member variables. These functions are implemented in a separate file to more clearly mark that they are heuristic functions, not core parts of the ER solver framework.

`ExtDefMap` This class is a data structure for storing extension variables and their definitions. It is designed to provide fast lookups in both directions: from extension variables to their corresponding definitions, and from pairs of literals to corresponding extension variables. It is also used to quickly access the set of extension variables that can be deleted (i.e., the set of extension variables which are not used to define another extension variable).

3 How to Add a New Heuristic

To add a new heuristic for the ER components of the solver, declare the heuristic function in `ERManager.h` and implement it in `ERUserHeuristics.cc`. Then, in the `ERManager` constructor implemented in `ERManager.cc`, bind the heuristic function to the corresponding function object using `std::bind`. The solver framework currently supports 6 different types of heuristics:

- Clause filtering predicate (search for `user_extFilPredicate_`): this is the first step of clause selection, used to quickly discard clauses from further consideration. This predicate takes a clause reference as input and outputs a Boolean value indicating whether the clause should be kept or discarded (true to keep, false to discard).

- Clause selection heuristic (search for `user_extSelHeuristic_`): this is the second and final step of clause selection. It takes the list of clauses selected by the clause filtering predicate and selects a subset which it deems to be interesting. This heuristic is parameterized by the maximum number of clauses that should be selected. Rather than allocating a new list for output, the clause selection heuristic appends its clauses to a vector object which is allocated by the caller.

- New variable definition heuristic (search for `user_extDefHeuristic_`): this heuristic is responsible for taking the set of clauses identified by clause selection and generating a set of extension variable definitions from it. The variable definitions generated by this heuristic are stored as `ExtDef` structs, which are appended to a vector object which is allocated by the caller. To avoid duplicating variable IDs, the ID of each new variable should be computed as the sum of the current number of variables in the solver and the number of definitions that have been queued to be added: `Var x = assignmentTrail.nVars() + extVarDefBuffer.size();`. This is necessary in case extension variable definitions have been queued previously and have not yet been introduced into the solver.

- New variable substitution predicate (search for `user_extSubPredicate_`): this predicate is responsible for determining whether it would be useful to try replacing pairs of literals in a clause with a corresponding extension variable. It takes a clause as input and outputs a Boolean value indicating whether the solver should try substituting pairs of literals in the clause with extension variables (true to perform substitution, false to skip substitution).

- New variable deletion setup (search for `user_extDelPredicateSetup_`): this function is responsible for setting up data structures to decide whether an extension variable should be deleted. It is called at the beginning of the extension variable deletion routine, before any calls to the extension variable deletion predicate.

- New variable deletion predicate (search for `user_extDelPredicate_`): this predicate is responsible for determining whether it would be useful to delete an extension variable. It takes an extension variable as input and outputs a Boolean value indicating whether the extension variable should be deleted (true to delete the variable, false to keep it).

4 Existing Heuristics

The heuristics described in the thesis [2] (as well as some other baseline heuristics) have already been implemented and are provided in the source code. Some of these heuristics have associated parameter values which can be set via command-line options. These options and their default values are defined in `ERManager.cc`. There are some general options that control how often extension variables are added into the solver (`opt_ext_freq` and `opt_ext_num`) and how frequently extension variables should be deleted (`opt_ext_del_freq`). Other options and brief descriptions of each heuristic are described below:

- `user_extFilPredicate_width`: decide whether to consider a clause based on its width. Clauses are selected if their width lies within the user-specified range (see the `opt_ext_min_width` and `opt_ext_max_width` options).
- `user_extFilPredicate_lbd`: decide whether to consider a clause based on its LBD. Clauses are selected if their LBD lies within the user-specified range (see the `opt_ext_min_lbd` and `opt_ext_max_lbd` options).
- `user_extFilPredicate_ler`: only consider the last two learnt clauses as described in the LER paper [1].
- `user_extSelHeuristic_all`: dummy heuristic – this simply accepts all the clauses filtered by the clause filtering predicate. This is currently used for implementing the LER method.
- `user_extSelHeuristic_activity`: select k clauses with the highest activities (see the `opt_ext_wndw` option).
- `user_extDefHeuristic_random`: define extension variables by selecting two literals at random from the set of selected clauses.
- `user_extDefHeuristic_subexpression`: define extension variables by selecting the most frequently-appearing pairs of literals. This has two different implementations for counting pairs of subexpressions: one which naively iterates over every pair of literals in every pair of clauses, and another which uses set intersections. Switching between these implementations can be done by defining the `ER_USER_ADD_SUBEXPR_SET_INTERSECTION` preprocessor macro to either true or false. This also has two different implementations for selecting the most frequent pairs of literals: one using

the quickselect algorithm, and another based on insertion sort. The implementations can be selected by defining the `USE_QUICKSELECT_SUBEXPRS` preprocessor macro.

- `user_extDefHeuristic_ler`: define extension variables according to the LER method [1].
- `user_extSubPredicate_size_lbd`: decide whether to substitute into a clause based on its width and LBD (see the `opt_ext_sub_min_width` and `opt_ext_sub_max_width` options).
- `user_extDelPredicateSetup_none`: don't perform any setup for deleting extension variables.
- `user_extDelPredicate_none`: don't delete extension variables.
- `user_extDelPredicate_all`: delete all extension variables
- `user_extDelPredicateSetup_activity`: setup function for deleting low-activity extension variables. Sorts extension variables by activity.
- `user_extDelPredicate_activity`: deletes extension variables with low activity (see the `opt_ext_act_thresh` option).

References

[1] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[2] Chung, Jonathan. Prioritized unit propagation and extended resolution techniques for sat solvers. Master's thesis, University of Waterloo, 2023.