Clause-Learning for Modular Systems

David Mitchell and Eugenia Ternovska
{mitchell,ter}@cs.sfu.ca

Computational Logic Laboratory, Simon Fraser University

Abstract. We present an algorithm, CDCL-AMS, for solving Modular Systems consisting of a set of modules where, for each module, we have a simple "black-box" solver. The algorithm is based on the Conflict-Directed Clause Learning algorithm for SAT, and communicates asynchronously with the black-box solvers to accommodate high variability in response latencies.

1 Introduction

In many modern contexts, finding a solution to a problem amounts to solving a combinatorial search problem where the constraints are implicit in a collection of more-or-less independent modules, each of which is a knowledge base or problem-solving system in its own right, and typically presented via network connections. The conflict-directed clause learning (CDCL) algorithm [5] is the basis of SAT solvers with impressive performance on many constraint problems. However, in the multi-module context we consider here, it is often undesirable or even impossible to transform each module into a set of explicit constraints. In this context, each module is a black box which answers queries, for example of the form "do you have a solution consistent with partial solution X?" In many settings, the response latencies for modules will be substantial, highly variable, and largely un-correlated. The main purpose of this paper is to present a CDCL-based algorithm suited to this context.

Modular Systems A general logic-based formalization of problem solving in a multi-module context is provided by the notion of Modular Systems, as defined in [8]. This generalizes the formalization of a decision problem as a class of structures, or of a search problem as model expansion [6]. Formally, a *module* is a class of structures for a fixed vocabulary. Modular systems are defined by combining primitive modules with an algebra of modular systems. The algebra is similar to Codd's relational algebra, but defined on *classes of structures* rather than on relational tables. The operations are Sequential Composition, Union, Complementation, Projection and Feedback.

In this short paper, we restrict our attention to systems which are conjunctions of "primitive" modules. In the context of solving a particular problem instance. The algorithm is based on the idea of querying modules with a partial structures, which for a particular problem instance are on a fixed universe. In this setting, queries are essentially propositional, so for simplicity we present our algorithms in purely propositional form. Our setting is as follows.

1. Each module M_i is a set of truth assignments for propositional vocabulary σ_i. We say assignment α satisfies M_i if $\alpha \in M_i$, and we say module M_i implies clause C iff every assignment in M_i satisfies C.
2. Modular System $\mathcal{M}$ is a conjunction of modules. Its vocabulary is the union of those of its modules. $\mathcal{M}$ is satisfiable if there is a truth assignment for σ which satisfies every module. $\mathcal{M}$ implies clause C if every satisfying assignment for $\mathcal{M}$ satisfies C.
3. For each module M_i, we have a solver S_i that answers queries in the form of partial truth assignments for σ_i. We consider only solvers which are propagators, that is, when queried with partial assignment α for σ_i, in finite time return one of:
   (a) $\langle \text{Reject}, \text{Reason} \rangle$, where Reason is a set of clauses which are implied by M_i and false in α.
   (b) $\langle \text{Accept}, \text{Advice} \rangle$, where Advice is a set of clauses implied by M_i, with exactly one literal not defined by α and all other literals false in α.

Remark 1. It is possible for Advice or Reason to be empty. In particular, a solver S_i for which Advice and Reason are *always* empty is simply verifier for M_i. It follows that any verifier can be wrapped as a (not very helpful) propagator.

The CDCL Algorithm We assume the reader is familiar with CDCL, but review some aspects here, and also fix some notation. Given CNF formula Γ, the algorithm incrementally constructs a sequence of literals α (the "assignment stack") which defines a partial truth assignment for Γ. (We henceforth gloss over the distinction between assignments and sequences of literals.) The algorithm alternately adds an unassigned literal l to α (called a "decision"), and then extends α with any literals newly determined by unit propagation. The sub-sequence of α consisting of only decision literals is the "decision sequence" corresponding to α, and will be denoted δ. Each literal l of α that is not a decision is labelled with the clause that the unit propagation engine used to set it true, called the "reason for l". This guess-propagate process continues until either $\alpha \models \Gamma$, in which case the algorithm halts, or the unit propagation engine determines that a literal l must be true, but $\neg l$ is already in α, called a "conflict".

For clause set Γ and set δ of literals, we write $l \in \text{UP}(\Gamma, \delta)$ if setting the literals of δ true and running unit propagation on Γ results in l being set true, and $C \in \text{UP}(\Gamma, \delta)$ if C is obtained from a clause of Γ when, for every literal $l \in \text{UP}(\Gamma, \delta)$ we delete from δ any clause containing l, and delete from each clause of δ any occurrence of $\neg l$. Notice that, if α is an assignment stack, and δ the corresponding decision sequence then $\alpha = \text{UP}(\Gamma, \delta)$. Also, we denote the empty clause by $\Box$, so $\Box \in \text{UP}(\Gamma, \delta)$ that executing unit propagation from Γ and δ produces a conflict.

Upon a conflict, CDCL derives a clause C by resolution, using the reasons labelling α. The clause C is used to determine a proper prefix α' of α, which replaces α as the current assignment stack (called "back-jumping"), and C is added to Γ (it is called a "learned" clause). The clause C and assignment stack α bear a particular relationship: C is an "asserting clause" α' and Γ, as defined next.

A clause C is a conflict clause for decision sequence δ and clause set Γ iff: 1) $\square \in \mathrm{UP}(\Gamma, \overline{C})$, and 2) For each literal $l \in C$, $\bar{l} \in \mathrm{UP}(\Gamma, \delta)$. C is an asserting clause for Γ and δ if it is a conflict clause and also satisfies: 3) For exactly one literal $l \in C$, $\bar{l} \notin UP(\Gamma, \delta^-)$, where δ^- is δ with its last element removed.

Observe that, when C is a conflict clause for α and Γ, $\mathrm{UP}(\Gamma \cup \{C\}, \delta)$ contains at least one literal not in α, so immediately after back-jumping, the new assignment stack is extended by unit propagation, which could in turn lead to a new conflict. If a state is reached where this does not happen, the algorithm returns to extending α with decisions.

Synchronous CDCL for Sets of Modules We can make a version of CDCL for sets of modules very simply, provided we query them synchronously and the response times are sufficiently fast. To do this, we simply query each solver every time we extend the current assignment stack, adding all reasons and advice returned by the solvers to Γ. However, we are interested in the case when these properties do not hold, in which case it is unreasonable to make so many queries, and unreasonable to have the algorithm wait for all solvers to respond to a given query before proceeding.

2 Asynchronous CDCL for Modular Systems

Our asynchronous clause-learning algorithm, which we call CDCL-AMS, has three processes (not including the solvers): The CDCL Engine, the Query Handler and the Response Handler. These communicate via four data objects: clause set Γ; set QUERIES of available solver queries; set HOLD decision sequences corresponding to pending queries; and set CONTINUE of query responses waiting to be handled by the Engine. The Engine is a CDCL solver that tries to decide satisfiability of $\mathcal{M}$. It generates models of Γ, which become queries to solvers. Each query is extended until either it is rejected by a solver, at which time at least one clause is added to Γ, or is accepted by all solvers. The other processes handle the communication between the Engine and the solvers.

Γ is initially empty, and is extended by clauses obtained from solvers in response to queries, and by the standard clause learning mechanism. The algorithm proceeds as in standard CDCL, extending its assignment stack until either a conflict or a satisfying assignment for Γ is obtained. Upon conflict, learned clause derivation and back-jumping are carried out. This process is identical to standard CDCL, except for the role of HOLD, which is discussed below. If $\alpha \models \Gamma$, α becomes a query to send to a solver, and the Engine:

1. Adds α to QUERIES;
2. Adds δ, the decision sequence corresponding to α, to HOLD;

3. Replaces δ with some proper subsequence, and modifies α accordingly. (This is back-jumping in the absence of a true conflict. Correctness requires only that the back-jump is proper, i.e., removes at least one decision from α.)

When a solver S becomes available, the Query Handler selects and removes an appropriate query from QUERIES and submits it to S. (A query is appropriate for S if S could reject it. Slightly more precisely: the vocabulary of α, less any prefix of α that has previously been accepted by S, has non-empty intersection with the vocabulary of S.) When S responds to query α, the Response Handler:

1. Adds all clauses in the returned Reasons or Advice to Γ. If the response is Reject, but Reasons is empty, it adds to Γ the clause consisting of the disjunction of the negations of the literals in δ;
2. Marks α to indicate that S has accepted it;
3. Removes the δ corresponding to α from HOLD.

A solution to $\mathcal{M}$ is an assignment that has been accepted by every solver. In the algorithm, a query is sent only to one solver before being returned to the main engine (via CONTINUE). This keeps all reasoning in one algorithm, and also ensures that information obtained by the main engine is exploited as soon as possible, avoiding, for example, submitting a query to a solver when clauses returned by other solvers already imply its rejection. To keep track of which solvers have accepted a query, each assignment stack has a mark for each solver indicating its largest prefix which has been accepted by that solver. When a query is returned to the main engine, via CONTINUE, it may be extended, both by unit propagation involving new clauses and by new decisions, and these marks are maintained.

The purpose of the HOLD set is to ensure that, while a response to a query based on decision sequence δ is pending (in QUERIES, or being handled by a solver), the engine does not generate any query which is "no better than" δ (i.e., is a superset of δ). To this end, each decision sequence δ in HOLD is treated by the unit propagation engine as the clause $\overline{\delta}$, the disjunction of complements of literals in δ.

The clauses in HOLD cannot be used as "reasons" for the purpose of asserting clause derivation because they are guesses and might not be implied by $\mathcal{M}$. In the derivation, any literal set by unit propagation by using a clause of HOLD must be treated as a decision literal. The standard methods for asserting clause derivation ensure every learned clauses is unique. This now fails, and it is possible that an execution of the body of the main loop of the Engine fails to generate a new query or add a new clause. Loop iterations which are essentially "wheel-spinning" may result. While undesirable, it seems that this cannot entirely be avoided: If solvers take sufficiently long to respond to queries, the main engine will generate all possible resolvents and all possible queries based on Γ, and can make no further progress until some further query response arrives.

The CDCL-AMS Engine is given by Algorithm 1.

Algorithm 1: CDCL-AMS Engine

```
Input: Vocabulary $\sigma$
Output: SAT or UNSAT
1  $\delta \leftarrow$ any non-empty decision sequence
2  repeat
3      if CONTINUE $\neq \emptyset$ then
4          Remove one query $\gamma$ from CONTINUE
5          if $\gamma$ satisfied all modules then
6              return SAT
7          end
8          Remove $\gamma$ from HOLD
9          $\delta \leftarrow \gamma$
10     end
11     $\delta \leftarrow$ ExtendAndLearn-AMS($\delta, \Gamma$)
12     if $\delta = \langle\rangle$ then
13         return UNSAT
14     end
15     if $UP(\Gamma, \delta) \models \Gamma$ then
16         Add $\delta$ to HOLD
17         Add $\delta$ to QUERIES
18         $\delta \leftarrow$ a proper sub-sequence of $\delta$
19     end
20 end
```

Algorithm 1 is described in terms of decision sequences (δ and γ), but the corresponding assignment stack, labelled with both reasons and marks from accepting solvers, is implicitly maintained, and is in fact what is being operated on. (For example, in adding δ to QUERIES, it is clear that the labelled assignment stack must be intended. The exception is line 16, where it is indeed just the decision sequence δ that is added to HOLD.)

In line 11, **ExtendAndLearn-AMS(δ,Γ)** carries out the process of extending decision sequence δ until either it satisfies Γ, or generates a conflict. In the latter case, it may generate multiple conflicts (and learned clauses), eventually returning a decision sequence δ such that either $UP(\Gamma, \delta) \models \Gamma$ or computing UP(Γ, δ) does *not* generate a conflict. (This may correspond to many iterations of the main loop of CDCL as it is usually presented.) At line 12, if $\delta = \langle\rangle$, the last learned clause was $\square$.

Remark 2. If at any point Γ becomes unsatisfiable, the algorithm essentially becomes a standard CDCL solver. In particular, it generates no new queries, and once existing contents of QUERIES have been exhausted, and their responses all handled, the algorithm becomes CDCL proving unsatisfiability of Γ.

In Line 5, "γ satisfied all modules" means that every solver S accepted (some previous version of) γ at a time when it was (already) total for the vocabulary of S.

Correctness If the algorithm returns SAT, then some assignment is total for, and accepted by, every solver, so $\mathcal{M}$ is satisfiable. If the algorithm returns UNSAT, it is because the empty clause has been derived from Γ. Since $\mathcal{M} \models \Gamma$, $\mathcal{M}$ is unsatisfiable. It remains to establish termination.

Observe that the main engine is simply a CDCL engine which generates all satisfying assignments for Γ. (Γ grows monotonically, so some generated satisfying assignments later are not satisfying, but this does not affect the argument.) To see that the algorithm makes progress, we observe that every assignment α generated by the engine is eventually extended until it either becomes a satisfying assignment for $\mathcal{M}$, or is "killed" by generation of a clause which is implied by $\mathcal{M}$, but is false in α. We need to see that, in each iteration of the Engine main loop, progress is made either by δ being killed by a new clause added to Γ, or by "getting closer" to satisfying every module. For the moment, set aside the question of the "wheel-spinning" iterations mentioned above. Consider line 11, where δ is re-assigned by the call to **ExtendAndLearn-AMS(δ,Γ)**. To avoid ambiguity, let's write β for the new value of δ. If at line 3 CONTINUE was empty, then the standard CDCL process will ensure that either β is a proper extension of δ, or that a clause that "kills" δ was added to Γ, and β is new. So we have progress. If CONTINUE was not empty, δ could be killed either because the response for δ was Reject (in which case a killing clause was added to Γ), or because unit propagation from δ produced a conflict based on other clauses that have been added to Γ since δ was generated as a query. If δ does not get killed, and is not total, then β is set to a proper extension of δ. If δ is not killed and is total, then there is some solver S which has accepted a larger prefix of α than the previous time this query was in QUERIES, and α is marked with this information. In each case, we have made progress. It remains to verify that every query is eventually responded to, and that "wheel-spinning" iterations do not prevent eventual progress. There are finitely many possible queries, and each solver responds to each query in finite time, so each query is eventually responded to by a solver. Wheel-spinning iterations are only possible if HOLD is not empty, and since every query is eventually responded to, every element of HOLD is eventually deleted, at which point progress is ensured.

3 Discussion

Related Work Many related algorithms have been presented in the literature. These include propagation via lazy clause generation [7], algorithms used in used in "lazy" SMT solvers, methods for supporting external constraints in SMT and ASP solvers [3, 2, 1], and distributed and parallel CSP and SAT algorithms [4]. An abstract algorithmic scheme for Modular System solvers was given in [8]. We will give detailed comparisons in a longer paper.

Future Work A number of details of this algorithm warrant more careful discussion, and there are many refinements and heuristics to consider when contemplating implementation, even without taking into account the practical complexity of interacting with real on-line solvers. We will discuss some of these in a longer paper. In future work, we intend to examine the extension of these algorithms to the full algebra of modular systems; develop versions for use with solvers which are more than just propagators; study the relationship between problem structure and algorithm complexity; develop versions which make use of FO vocabulary of modules as classes of structures; attend more closely to issues that need to be addressed for implementability; and develop versions for use in distributed and many-core computational environments.

References

1. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
2. Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, 2012.
3. Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2009.
4. Norbert Manthey. *Towards Next Generation Sequential and Parallel SAT Solvers.* PhD thesis, TU Dresden, 2014.
5. João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
6. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 430–435. AAAI Press / The MIT Press, 2005.
7. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.
8. Shahab Tasharrofi and Eugenia Ternovska. A semantic account for modularity in multi-language modelling of search problems. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2011.