A PLTL-Prover Based on Labelled Superposition with Partial Model Guidance

Martin Suda[1,2,3,⋆] and Christoph Weidenbach[1,⋆⋆]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany
[3] Charles University, Prague, Czech Republic

Abstract. Labelled superposition (LPSup) is a new calculus for PLTL. One of its distinguishing features, in comparison to other resolution-based approaches, is its ability to construct partial models on the fly. We use this observation to design a new decision procedure for the logic, where the models are effectively used to guide the search. On a representative set of benchmarks, our implementation is then shown to considerably advance the state of the art.

1 Introduction

Labelled superposition (LPSup) is a new calculus for Propositional Linear Temporal Logic (PLTL). In previous work [7] we have shown a saturation based approach to deciding PLTL with LPSup. Here we instead rely on the ability of LPSup to generate partial models on the fly and use a SAT solver to drive the search and select inferences. This typically leads to a fast discovery of models, but also drastically reduces the number of inferences that need to be performed before an instance can be shown unsatisfiable.

Our method doesn't work with PLTL formulas directly, but instead relies on a certain normal form, which we review in Sect. 2. Algorithms for deciding PLTL formulas are inherently complicated, because one needs to show the existence of an infinite path through the world structure. Our algorithm is described in two steps in Sect. 3. First we show how a certain modification of bounded model checking can be turned into a complete method for the reachability tasks. This is then used as a subroutine to decide whole PLTL. Although the ideas underlying our algorithm rely on the theory of [7] that cannot be repeated here in full due to lack of space, we provide the most important ideas to understand our approach. In the final section 4, we compare LS4, an implementation of our algorithm, to other existing PLTL-provers on a representative set of benchmarks. The results clearly indicate that partial model guidance considerably improves the state of the art of symbolic based approaches to PLTL satisfiability checking.

⋆ Supported by Microsoft Research through its PhD Scholarship Programme.
⋆⋆ Supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

B. Gramlich, D. Miller, and U. Sattler (Eds.): IJCAR 2012, LNAI 7364, pp. 537–543, 2012.

2 Preliminaries

The language of propositional formulas and clauses over a given signature $\Sigma = \{p, q, \ldots\}$ of propositional variables is defined in the usual way. By propositional *valuation*, or simply a *world*, we mean a mapping $W : \Sigma \to \{0, 1\}$. We write $W \models P$ if a propositional formula P is satisfied by W. The syntax of PLTL is an extension of the propositional one by temporal operators $\Box, \Diamond, \mathsf{U}, \ldots$ We do not detail the syntax here, due to lack of space, but will instead directly rely on so called *Separated Normal Form (SNF)* to which any PLTL formula can be translated by a satisfiability preserving transformation with at most linear increase in size [6]. The semantics of PLTL is based on a discrete linear model of time, where the structure of possible time points is isomorphic to $\mathbb{N}$: An *interpretation* is an infinite sequence $(W_i)_{i\in\mathbb{N}}$ of worlds. In order to talk about two neighboring worlds at once we introduce a primed copy of the basic signature: $\Sigma' = \{p', q', \ldots\}$. Primes can also be applied to formulas and valuation with the obvious meaning. Formulas over $\Sigma \cup \Sigma'$ can be evaluated over the respective joined valuation: When both W_1 and W_2 are valuations over Σ, we write $[W_1, W_2]$ as a shorthand for the mapping $W_1 \cup (W_2)' : (\Sigma \cup \Sigma') \to \{0, 1\}$.

The input of our method is a refinement of SNF based on the results of [3]:

Definition 1. *A* PLTL-specification S *is a quadruple* (Σ, I, T, G) *such that*

- Σ *is a finite propositional signature,*
- I *is a set of* initial *clauses* C_i *(over the signature* Σ*),*
- T *is a set of* step *clauses* $C_t \vee D'_t$ *(over joined signature* $\Sigma \cup \Sigma'$*),*
- G *is a set of* goal *clauses* C_g *(over the signature* Σ*).*[1]

An interpretation $(W_i)_{i\in\mathbb{N}}$ *is a* model *of* $S = (\Sigma, I, T, G)$ *if*

1. *for every* $C_i \in I$, $W_0 \models C_i$,
2. *for every* $i \in \mathbb{N}$ *and every* $C_t \vee D'_t \in T$, $[W_i, W_{i+1}] \models C_t \vee D'_t$,
3. *there is infinitely many indices* j *such that for every* $C_g \in G$, $W_j \models C_g$.

A PLTL-specification S is *satisfiable* if it has a model.

Our algorithm for deciding satisfiability of PLTL-specifications to be described next relies on incremental SAT solver technology as described in [5]. There each call to the SAT solver is parameterized by a set of unit assumptions. It either returns a model of all the clauses inserted to the solver that also satisfies the given assumptions, or the UNSAT result along with a subset of the given assumptions that were needed in the proof. Negation of literals from the returned subset can be seen as a new conflict clause that has just been shown semantically entailed by the clauses stored in the solver.

[1] The specification stands for the PLTL formula $(\bigwedge C_i) \wedge \Box\,(\bigwedge (C_t \vee \bigcirc D_t)) \wedge \Box\Diamond\,(\bigwedge C_g)$ and can be understood as a symbolic representation of a Büchi automaton recognizing the set of all models of the original input formula.

3 The Algorithm

In order to explain the basic mechanics of our algorithm we first focus on a simpler problem of reaching a goal world only once. That is, given a specification $S = (\Sigma, I, T, G)$, we first try to establish whether there is a finite sequence of worlds $W_0, W_1, \ldots, W_k$ such that $W_0 \models I$, $W_k \models G$ and $[W_i, W_{i+1}] \models T$ for every two neighboring worlds W_i and W_{i+1}. An algorithm for this problem known from verification is called Bounded Model Checking (BMC) [2], where one looks for such a sequence by successively trying increasing values of k (starting with $k = 0$) and employs a SAT solver to answer the respective satisfiability queries[2] until a model for the sequence is found. We modify this approach in order to gain more information from the individual runs with answer UNSAT and to be able to ensure termination in the case of an overall unsatisfiable input.

The idea is to use multiple instances of the solver, as many as there is worlds in the current sequence, and build the sequence progressively, from the beginning towards the end. Each individual *solver instance* contains variables of the joined signature ($\Sigma \cup \Sigma'$) and thus represents two neighboring worlds. However, only the primed part is actually used for SAT solving. As the search proceeds forward, the world model constructed over Σ'-variables in the solver instance i is transformed to a set of assumptions over the Σ-variables for the instance $(i + 1)$. If a world model cannot be completed in the instance $(i + 1)$ due to inconsistency (the current world sequence cannot be extended by one more step) the instance $(i+1)$ returns a *conflict clause* over its assumptions on Σ-variables, which is *propagated* back and added to the solver instance i as a clause over Σ'. Thus the instance i will now produce a different world model, a model which additionally satisfies the added conflict clause. The whole situation is depicted in Fig. 1. We can also see from there how the individual solver instances are initialized. The first contains only the clauses from I (and doesn't depend on assumptions), all the other instances contain clauses from T, and the last instance, additionally, the clauses from G.

One *round* of the algorithm (for a specific value of k) ends either by building an overall sequence of worlds of length $k + 1$, which is a reason for termination with result SAT, or by deriving an empty clause in the first solver instance. Standard BMC would then simply increase k and continue searching for longer sequences. We can do better than that. By analyzing the overall proof[3] of the empty clause, we may discover it doesn't depend on (has not been derived with the help of) I or G in which case we terminate and report overall UNSAT: the same proof will also work for larger values of k. Even if the proof depends on both I, G (and of course T), we can still perform the following check: Define *layer* j as the set of all clauses that depend on G and have been propagated to the solver instance that lies j steps before the last one. Formally, we set layer 0 to be

[2] For every fixed k the question becomes whether there exist a model over $\bigcup_{i=0}^{k} \Sigma^{(k)}$ of the formula $I^{(0)} \wedge \bigwedge_{i=0}^{k-1} T^{(i)} \wedge G^{(k)}$, where $T^{(i)}$ stands for T primed i times, etc.

[3] Proof recording is not needed on the SAT solver side. The described analysis can be implemented with the help of so called *marker literals* as explained, e.g., in [1].

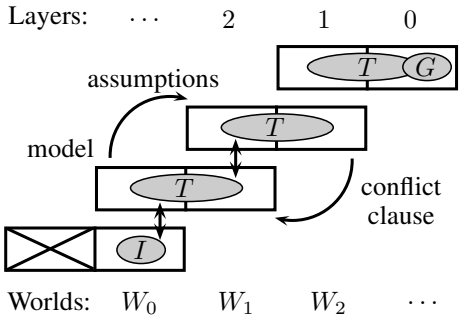

Fig. 1. The information exchanged between the individual solver instances. Completed world model is passed forward as a set of variable assumptions for the next instance. Failed run delivers a conflict clause over the variable assumptions so that the previous instance can be updated and SAT checking may find a different model. The very first solver instance doesn't depend on assumptions. When the last instance reports SAT, we have an overall model for the reachability task.

equal to G. Now, if there are indices $j_1 \neq j_2$ such that layers j_1 and j_2 are equal, we also terminate the algorithm with result UNSAT: we have just discovered a *layer repetition* in the proof, which means we know how we would derive empty clauses also in any of the rounds to come.[4] Note that the case of repeating layers is bound to occur, if not preceded by a different reason for termination, as there are only finitely many different sets of clauses over Σ. This shows the overall termination, and thus completeness, of our modification of BMC.

We now move to providing an algorithm for the general case, where a goal world is required to be shown reachable infinitely many times. In that algorithm we use the above described procedure as a basic building block. In fact, we call the configuration of solver instances as the one in Fig. 1 a *block*. The algorithm starts by building the first block exactly as described above. If this first step doesn't succeed in providing a sequence of worlds, leading from a world satisfying I to a one satisfying G, we terminate with result UNSAT. Otherwise we continue adding new blocks, but now the first solver instance of each new block no longer contains the clauses from I and is instead connected via the model/assumptions link described before to the world represented by the last solver instance of the previous block. This way we continue producing a sequence of blocks, each block being itself a sequence of solver instances (see Fig. 2), the whole thing representing a partial (unfinished) overall model of the given specification. As in the above procedure, each block grows from its initial length 1, and is only extended when necessary and just by one solver instance at a time.

For termination, we perform the following *model repetition check* to recognize a satisfiable specification. Each time a particular run of the SAT solver constructs

[4] Intuitively, the proof can be "cut" at the index j_1, and the part between j_1 and j_2 inserted arbitrarily many times, thus giving rise to proofs of arbitrary length.

Fig. 2. The layout of blocks as the search proceeds forward. The copies of clauses form T that occupy the positions of every pair of neighboring worlds are not depicted.

a new world model, i.e. a new world in the sequence, we scan all the worlds of previous blocks, and if one of them is equal to the new one, we terminate and report SAT. Note that only the worlds of previous blocks are eligible for the test, because we need to ensure that at least one world satisfying the clauses from G lies between the two repeating ones. The particular infinite sequence formally required as a model of the specification in now easily seen to be represented by the world sequence constructed so far, where the segment of worlds between the repetition will be traversed infinitely often.

Recognizing unsatisfiable specifications is again based on proof analysis. Note that now more than one block (or more precisely the set of goal clauses thereof) may be involved in the derivation of the empty clause. Each time an empty clause is derived, we *extend* the latest block involved in the proof by one additional solver instance and discard any blocks further to the right of it (in the sense of Fig. 2). Then we resume the search. As before each block maintains a sequence of *layers* of clauses. This time layer j contains the clauses that depend on *the block's own copy of* G and have been propagated to a solver instance that lies j steps before its last one. Detecting layer repetition for the first block incurs termination with the result UNSAT as before. If we detect repetition in a block which is not the first one, we perform a so called *leap* inference: A particular repeating layer is selected (see [7] for the details) and its clauses are globally added to the set G. Then the current block is discarded and the search continues from the last solver instance of the previous block. By construction,[5] this last instance currently doesn't provide a model for the strengthened set G, which is a key observation for proving overall termination of the algorithm, because it implies that the leap inference can only be applied finitely many times.

4 Experimental Evaluation

We implemented our algorithm in C++ with Minisat [5] version 2.2 as the backend solver. Although a more efficient implementation with just one solver instance (over an extended signature and special decision heuristic) seems possible, we really use multiple instances of the solver as described before, because it allows us to use the solver in a blackbox manner. An additional abstraction

[5] The intuition behind the leap inference is the following: We have just discovered that none of the successor worlds of the lastly visited G-world is itself a G-world. Thus the lastly visited G-world doesn't have the vital property of lying on a loop in the state space and may be safely discarded from consideration.

layer over the solver has been developed that allows us to mark any clause by a set of block indices it depends on in a form of *marker literals* [1]. That is how we perform the proof analysis described in the previous section without an actual need of true proof recording on the SAT solver side. The standard `set` container is used to represent layers and a simple linear pass implements both the model repetition check and the layer repetition check. We found out that most of the overall running time is typically spent inside individual calls to Minisat and therefore didn't attempt any further optimizations of the checks. As an additional trick we adapted the variable and clause elimination preprocessing of Minisat [4] to be also used on our inputs. This is done only once, before the actual algorithm starts. Special care needs to be taken, because of the dependency between the variables in Σ and Σ'. Moreover, we still need to be able to separate the clauses after elimination into sets I, T and G, which can be achieved by a clever use of marker literals.

Table 1. Number of problems (SAT/UNSAT) solved by each prover – timelimit 60s

problem set	# problems	LS4	trp++	'satisfiable'	'model'
TRP-suite	22	2/20	2/19	2/13	2/13
HW-reach	465	38/55	3/30	0/0	0/0
HW-live	118	38/15	7/7	3/4	0/1

We compared our implementation[6], which we call LS4, with clausal temporal resolution prover trp++[7] version 2.1 and two tableaux-based decision procedures implemented in the PLTL module of the Logics Workbench[8] Version 1.1, namely the 'satisfiable' function and the 'model' function. All the tests were performed on our servers with 3.16 GHz Xeon CPU, 16 GB RAM, running Debian 5.0. We collected several benchmark sets from different sources. The TRP-suite consists of 22 problems available on the web page of trp++[7] in the TOY and FO subdirectories. Further, we translated into PLTL the benchmarks from Hardware Model Checking Competition (HWMCC) 2011[9]. We obtained a set of 465 problems, here denoted HW-reach, from the safety checking track, and 118 problems from the liveness track, HW-live. Note that the competition examples are natively described as circuits in the form of And-Inverter Graphs; these were translated into clause form by standard techniques. The results from these benchmarks are summarized in Table 1. For each prover we report the number of satisfiable and unsatisfiable problems solved in 60 seconds. For a second test we generated formulas from several scalable families[6] and in Table 2 we report for each family the maximal size a prover was able to solve in 60 seconds.

We can see that LS4 is the only system to solve all the problems in the TRP-suite in the given time limit. It also by far outperforms the other systems on

[6] `http://www.mpi-inf.mpg.de/~suda/ls4.html`
[7] `http://www.csc.liv.ac.uk/~konev/software/trp++/`
[8] `http://www.lwb.unibe.ch/`
[9] `http://fmv.jku.at/hwmcc11/`

Table 2. Maximal formula size (from the range) solved in 60s by the provers.

formula family	size range	LS4	trp++	'satisfiable'	'model'
C1	1-100	100	100	3	100
C2	2-20	19	20	3	2
bincnt_u	1-16	10	16	11	6
bincnt_s	1-16	10	11	11	7
binflip_u	2-10	10	5	6	3
binflip_a	2-10	10	5	6	4

the problems coming from verification.[10] The formula families let us see that guidance by a partial model is not always an advantage. For example, on the bincnt_u family, LS4 has to construct an exponentially long path before it starts deriving conflict clauses. Moreover, these need to be propagated back trough all the worlds of the path before the final contradiction is reached. On the other side, the binflip families are already more difficult for the saturation based prover trp++. For example, on binflip_u of size 5, trp++ generates 1494299 resolvents before deriving the empty clause (in 3.67s), while LS4 needs only 1891 calls to Minisat and derives 936 non-empty conflict clauses before reaching the same conclusion (and spends 0.01s on that). To sum up, our test results demonstrate that LS4 considerably advances the state of the art in PLTL satisfiability checking.

References

[1] Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Efficient Generation of Unsatisfiability Proofs and Cores in SAT. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 16–30. Springer, Heidelberg (2008)

[2] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[3] Degtyarev, A., Fisher, M., Konev, B.: A Simplified Clausal Resolution Procedure for Propositional Linear-Time Temporal Logic. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 85–99. Springer, Heidelberg (2002)

[4] Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)

[5] Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

[6] Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Trans. Comput. Logic 2, 12–56 (2001)

[7] Suda, M., Weidenbach, C.: Labelled Superposition for PLTL. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 391–405. Springer, Heidelberg (2012)

[10] Note that a dedicated tool `suprove`, the winner of the safety checking track of HWMCC 2011, solved 395 problems in 900s. The winner of the liveness track, the tool `tip`, solved 77 problems within the same timelimit.