Extracting Unsatisfiable Cores for LTL via Temporal Resolution

Viktor Schuppan
Email: Viktor.Schuppan@gmx.de

***Abstract*—Unsatisfiable cores (UCs) are a well established means for debugging in a declarative setting. Still, tools that perform automated extraction of UCs for LTL are scarce. Using resolution graphs to extract UCs is common in many domains. In this paper we construct and optimize resolution graphs for temporal resolution as implemented in the temporal resolution-based solver `TRP++` and we use them to extract UCs for propositional LTL. We implement our method in `TRP++`, and we experimentally evaluate it. Source code of our tool is available.**

***Index Terms*—LTL; unsatisfiable cores; vacuity; temporal resolution;**

I. Introduction

Debugging is an activity that many hardware and software developers spend a fair amount of time on. When faced with some input that induces an undesired behavior it is typically suggested to minimize that failure-inducing input in order to simplify identification of the problem (e.g., [1]). Corresponding research has been performed, e.g., in linear programming (e.g., [2]), constraint satisfaction (e.g., [3]), compilers (e.g., [4]), SAT (e.g., [5]), declarative specifications (e.g., [6]), and LTL satisfiability (e.g., [7]) and realizability (e.g., [8]).

LTL and its relatives are important specification languages for reactive systems (e.g., [9]) and for business processes (e.g., [10]). Experience in verification as well as in synthesis has lead to specifications themselves becoming objects of analysis. Beer et al. report [11] that in their experience "[...] during the first formal verification runs of a new hardware design, typically 20 % of formulas are found to be trivially valid, and that trivial validity always points to a real problem in either the design or its specification or environment.". In a work on LTL synthesis [12] Bloem et al. state that "[...] writing a complete formal specification [...] was not trivial." and "Although this approach removes the need for verification [...] the specification itself still needs to be validated.".

Typically, a specification is expected to be satisfiable. If it turns out to be unsatisfiable, finding a reason for unsatisfiability can help with the ensuing debugging. Frequently, such reason for unsatisfiability is taken to be a part of the unsatisfiable specification that is by itself unsatisfiable (e.g., [7], [3], [2]); this is called an unsatisfiable core (UC) (e.g., [7], [13]). Less simplistic ways to examine an LTL specification ϕ exist [14], and understanding their results also benefits from availability of UCs. First, one can ask whether a certain scenario ϕ', given as an LTL formula, is permitted by ϕ. That is the case iff $\phi \wedge \phi'$ is satisfiable. Second, one can check whether ϕ ensures a certain LTL property ϕ''. ϕ'' holds in ϕ iff $\phi \wedge \neg\phi''$ is unsatisfiable. In the first case, if the scenario turns out not to be permitted by the specification, a UC can help to understand which parts of the specification and the scenario are responsible for that. In the second case a UC can show which parts of the specification imply the property. Moreover, if there are parts of the property that are not part of the UC, then those parts of the property could be strengthened without invalidating the property in the specification; i.e., the property is vacuously satisfied (e.g., [15]). UCs are therefore an important part of design methods for embedded systems (e.g., [14]) as well as for business processes (e.g., [16]). Note that specifications of real world systems may be 100s of pages long (e.g., [17]). Hence, providing automated support for obtaining a UC in case such a specification turns out to be unsatisfiable is crucial. UCs also have applications in avoiding the exploration of parts of a search space that can be known not to contain a solution for reasons "equivalent" to the reasons for previous failures (e.g., [18], [19]) and in certifying the correctness of a result of unsatisfiability (e.g., [20]). While our results also benefit these applications, we focus on debugging below. Despite their relevance interest in UCs for LTL has been somewhat limited (e.g., [19], [7], [21], [22]). In particular, publicly available tools that automatically extract fine-grained UCs for propositional LTL are scarce.

Extracting UCs is often possible using any solver for the logic under consideration by weakening subformulas one by one and using the solver to test whether the weakened formula is still unsatisfiable (e.g., [23]). While that is simple to implement, repeated testing for preservation of unsatisfiability may impose a significant run time burden. Hence, it is interesting to investigate methods to extract UCs from a single run of a solver. Extracting UCs from resolution graphs is common in SAT (e.g., [20]). A resolution method for LTL, temporal resolution (TR), was suggested by Fisher [24], [25] and implemented in `TRP++` [26], [27], [28].

TR lends itself as a basis for extracting UCs for LTL for two reasons. First, the TR-based solver `TRP++` proved to be competitive in a recent evaluation of solvers for LTL satisfiability, in particular on unsatisfiable instances [29]. Second, a TR proof naturally induces a resolution graph, which provides a clean framework for extracting a UC. Note, that while the BDD-based solver `NuSMV` [30] also performed well on unsatisfiable instances in [29], the BDD layer makes extraction of a UC more involved. On the other hand, the tableau-based solvers `LWB` [31] and `pltl` [32] provide access to a proof of unsatisfiability comparable to TR, yet tended to perform

worse on unsatisfiable instances in [29].

In this paper we make the following contributions. We construct resolution graphs for TR for propositional LTL as implemented in TRP++, and we use them to extract UCs. Note that TR is significantly more complex than propositional resolution. Hence, we use the specifics of TR in TRP++ to optimize the construction of resolution graphs. The temporal aspect also allows to extract more fine-grained information from the resolution graph; this is exploited in a companion paper [33], which this paper provides the basis for. We implement our method in TRP++, and we experimentally evaluate it. We make the source code of our solver available.

Conceptually, under the frequently legitimate assumption that a system description can be translated into an LTL formula, our results extend to vacuity for LTL [34]. Due to space constraints we refer to App. D of [35] for details.

In [19] Cimatti et al. perform extraction of UCs for PSL to accelerate a PSL satisfiability solver by performing Boolean abstraction. Their notion of UCs is coarser than ours and their solver is based on BDDs and on SAT. An investigation of notions of UCs for LTL including the relation between UCs and vacuity is performed in [7]. No implementation or experimental results are reported, and TR is not considered. Hantry et al. suggest a method to extract UCs for LTL in a tableau-based solver [21]. No implementation or experiments are reported. Awad et al. [16] use tableaux to extract UCs in the context of synthesizing business process templates. The description of the method is sketchy and incomplete, the notion of UC appears to be one of a subset of a set of formulas, and no detailed experimental evaluation is carried out. In [22] the decision and search problems for minimal UCs for LTL are shown to be PSPACE- and FPSPACE-complete, respectively. In [36] Cimatti et al. show how to prove and explain unfeasibility of message sequence charts for networks of hybrid automata. They consider a different specification language and use an SMT-based algorithm. Some work deals with unrealizable rather than unsatisfiable cores. [8] handles specifications in GR(1), which is a proper subset of LTL. Könighofer et al. present methods to help debugging unrealizable specifications by extracting unrealizable cores and simulating counterstrategies [37] as well as performing error localization using model-based diagnosis [38]. Raman and Kress-Gazit [39] present a tool that points out unrealizable cores in the context of robot control. [7] explores more fine-grained notions of unrealizable cores than [8], [37]. In vacuity Simmonds et al. [15] use SAT-based bounded model checking for vacuity detection. They only consider k-step vacuity, i.e., taking into account bounded model checking runs up to a bound k, and leave the problem of removing the bound k open. For a more extensive discussion on the relation between vacuity and UCs for LTL we refer to App. D of [35] and [7].

Section II starts with preliminaries. TR and its clausal normal form SNF are introduced in Sec. III. In Sec. IV we describe the construction of a resolution graph and its use to obtain a UC. The UCs obtained in Sec. IV are lifted from SNF to LTL in Sec. V and minimized in Sec. VI. In Sec. VII we provide examples that illustrate why these UCs are useful and how to obtain them. We discuss our implementation and experimental evaluation in Sec. VIII. Section IX concludes. Due to space constraints proofs are sketched or omitted. For a full version [35] of this paper including proofs and for implementation, examples, and log files see [40].

II. Preliminaries

We use a standard version of LTL, see, e.g., [41]. Let $\mathbb{B}$ be the set of Booleans, and let AP be a finite set of atomic propositions. The set of *LTL formulas* is constructed inductively as follows. The Boolean constants 0 (false), 1 (true) $\in \mathbb{B}$ and any atomic proposition $p \in AP$ are LTL formulas. If ψ, ψ' are LTL formulas, so are $\neg\psi$ (not), $\psi \vee \psi'$ (or), $\psi \wedge \psi'$ (and), $\mathbf{X}\psi$ (next time), $\psi\mathbf{U}\psi'$ (until), $\psi\mathbf{R}\psi'$ (releases), $\mathbf{F}\psi$ (finally), and $\mathbf{G}\psi$ (globally). We use $\psi \rightarrow \psi'$ (implies) as an abbreviation for $\neg\psi \vee \psi'$.

III. Temporal Resolution (TR)

TR works on formulas in a clausal normal form called separated normal form (SNF) [25]. For any atomic proposition $p \in AP$ p and $\neg p$ are *literals*. Let $p_1, \ldots, p_n$, $q_1, \ldots, q_{n'}$, l with $0 \leq n, n'$ be literals such that $p_1, \ldots, p_n$ and $q_1, \ldots, q_{n'}$ are pairwise different. Then (i) $(p_1 \vee \ldots \vee p_n)$ is an *initial clause*; (ii) $(\mathbf{G}((p_1 \vee \ldots \vee p_n) \vee (\mathbf{X}(q_1 \vee \ldots \vee q_{n'}))))$ is a *global clause*; and (iii) $(\mathbf{G}((p_1 \vee \ldots \vee p_n) \vee (\mathbf{F}(l))))$ is an *eventuality clause*. l is called an *eventuality literal*. As usual an empty disjunction (resp. conjunction) stands for 0 (resp. 1). $()$ or $(\mathbf{G}())$, denoted $\Box$, stand for 0 or $\mathbf{G}(0)$ and are called *empty clause*. The set of all SNF clauses is denoted $\mathbb{C}$. Let $c_1, \ldots, c_n$ with $0 \leq n$ be SNF clauses. Then $\bigwedge_{1 \leq i \leq n} c_i$ is an LTL formula in *SNF*. Every LTL formula ϕ can be transformed into an equisatisfiable formula ϕ' in SNF [25].

The production rules of TRP++ are shown in Tab. I. The second column assigns a name to a production rule. The third and fifth columns list the premises. The seventh column gives the conclusion. Columns 4, 6, and 8 are described below. Columns 9–11 become relevant only in later sections.

The algorithm in Fig. 1 provides a high level view of TR in TRP++ [26]. The algorithm takes a set of starting clauses C in SNF as input. It returns *unsat* if C is found to be unsatisfiable (by deriving $\Box$) and *sat* otherwise. Resolution between two initial or two global clauses or between an initial and a global clause is performed by a straightforward extension of propositional resolution. The corresponding production rules are listed next to *saturation* in Tab. I. Given a set of SNF clauses C we say that one *saturates* C if one applies these production rules to clauses in C until no new clauses are generated. Resolution between a set of initial and global clauses and an eventuality clause with eventuality literal l requires finding a set of global clauses that allows to infer conditions under which $\mathbf{XG}\neg l$ holds. Such a set of clauses is called a *loop* in $\neg l$. Loop search involves all production rules in Tab. I except init-ii, init-in, step-nn, and step-nx.

In line 1 the algorithm in Fig. 1 initializes M with the set of starting clauses and terminates iff one of these is the empty clause. Then, in line 2, it saturates M (terminating iff

TABLE I

PRODUCTION RULES USED IN TRP++. LET $P \equiv p_1 \vee \ldots \vee p_n$, $Q \equiv q_1 \vee \ldots \vee q_{n'}$, $R \equiv r_1 \vee \ldots \vee r_{n''}$, AND $S \equiv s_1 \vee \ldots \vee s_{n'''}$.

	rule	premise 1	part.	premise 2	part.	conclusion	part.	p.1 – c	p.2 – c	vt. c
saturation	init-ii	$(P \vee l)$	M	$(\neg l \vee Q)$	M	$(P \vee Q)$	M	✔	✔	✔
	init-in	$(P \vee l)$	M	$(\mathbf{G}(\neg l \vee Q))$	M	$(P \vee Q)$	M	✔	✔	✔
	step-nn	$(\mathbf{G}(P \vee l))$	M	$(\mathbf{G}(\neg l \vee Q))$	M	$(\mathbf{G}(P \vee Q))$	M	✔	✔	✔
	step-nx	$(\mathbf{G}(P \vee l))$	M	$(\mathbf{G}((Q) \vee (\mathbf{X}(\neg l \vee R))))$	M	$(\mathbf{G}((Q) \vee (\mathbf{X}(P \vee R))))$	M	✔	✔	✔
	step-xx	$(\mathbf{G}((P) \vee (\mathbf{X}(Q \vee l))))$	ML	$(\mathbf{G}((R) \vee (\mathbf{X}(\neg l \vee S))))$	ML	$(\mathbf{G}((P \vee R) \vee (\mathbf{X}(Q \vee S))))$	ML	✔	✔	✔
augmentation	aug1	$(\mathbf{G}((P) \vee (\mathbf{F}(l))))$			M	$(\mathbf{G}(P \vee l \vee wl))$	M	✔	—	✔
	aug2	$(\mathbf{G}((P) \vee (\mathbf{F}(l))))$			M	$(\mathbf{G}((\neg wl) \vee (\mathbf{X}(l \vee wl))))$	M	✖	—	✔
BFS loop search	BFS-loop-it-init-x	$c \equiv (\mathbf{G}((P) \vee (\mathbf{X}(Q))))$ with $\lvert Q \rvert > 0$			M	c	L	✔	—	✔
	BFS-loop-it-init-n	$(\mathbf{G}(P))$			M	$(\mathbf{G}((0) \vee (\mathbf{X}(P))))$	L	✔	—	✔
	BFS-loop-it-init-c	$(\mathbf{G}(P))$	L'	$(\mathbf{G}((Q) \vee (\mathbf{F}(l))))$	M	$(\mathbf{G}((0) \vee (\mathbf{X}(P \vee l))))$	L	✖	✖	✔
	BFS-loop-it-sub	$c \equiv (\mathbf{G}(P))$ with $c \rightarrow (\mathbf{G}(Q))$			L	$(\mathbf{G}((0) \vee (\mathbf{X}(Q \vee l))))$ generated by BFS-loop-it-init-c	L	✔	—	✖
	BFS-loop-conclusion1	$(\mathbf{G}(P))$	L	$(\mathbf{G}((Q) \vee (\mathbf{F}(l))))$	M	$(\mathbf{G}(P \vee Q \vee l))$	M	✔	✔	✔
	BFS-loop-conclusion2	$(\mathbf{G}(P))$	L	$(\mathbf{G}((Q) \vee (\mathbf{F}(l))))$	M	$(\mathbf{G}((\neg wl) \vee (\mathbf{X}(P \vee l))))$	M	✔	✖	✔

Input: A set of SNF clauses C.
Output: *Unsat* if C is unsatisfiable; *sat* otherwise.

```
1  M ← C; if □ ∈ M then return unsat;
2  saturate(M); if □ ∈ M then return unsat;
3  augment(M);
4  saturate(M); if □ ∈ M then return unsat;
5  M' ← ∅;
6  while M' ≠ M do
7      M' ← M;
8      for c ∈ C . c is an eventuality clause do
9          C' ← {□};
10         repeat
11             initialize-BFS-loop-search-iteration(M, c, C', L);
12             saturate-step-xx(L);
13             C' ← {c' ∈ L | c' has empty X part};
14             C'' ← {(G(Q)) | (G((0) ∨ (X(Q ∨ l)))) ∈
                L generated by [BFS-loop-it-init-c]};
15             found ← subsumes(C', C'');
16         until found or C' = ∅ ;
17         if found then
18             derive-BFS-loop-search-conclusions(c, C', M);
19             saturate(M); if □ ∈ M then return unsat;
20 return sat;
```

Fig. 1. LTL satisfiability checking via TR in TRP++.

the empty clause is generated). In line 3 it *augments* M by applying production rule [aug1] to each eventuality clause in M and [aug2] once per eventuality literal in M, where wl is a fresh proposition. This is followed by another round of saturation in line 4. From now on the algorithm in Fig. 1 alternates between searching for a loop for some eventuality clause c (lines 9–18) and saturating M if loop search has generated new clauses (line 19). It terminates, if either the empty clause was derived (line 19) or if no new clauses were generated (line 20).

Loop search for some eventuality clause c may take several *iterations* (lines 11–15). Each loop search iteration uses saturation restricted to [step-xx] as a subroutine (line 12). Therefore, each loop search iteration has its own set of clauses L in which it works. We call M and L *partitions*. Columns 4, 6, and 8 in Tab. I indicate whether a premise (resp. conclusion) of a production rule is taken from (resp. put into) the main partition (M), the loop partition of the current loop search iteration (L), the loop partition of the previous loop search iteration (L'), or either of M or L as long as premises and conclusion are in the same partition (ML). In line 11 partition L of a loop search iteration is initialized by applying production rule [BFS-loop-it-init-x] once to each global clause with non-empty $\mathbf{X}$ part in M, rule [BFS-loop-it-init-n] once to each global clause with empty $\mathbf{X}$ part in M, and rule [BFS-loop-it-init-c] once to each global clause with empty $\mathbf{X}$ part in the partition of the previous loop search iteration L'. Notice that by construction at this point L contains only global clauses with non-empty $\mathbf{X}$ part. Then L is saturated using only rule [step-xx] (line 12). A loop has been found iff each global clause with empty $\mathbf{X}$ part that was derived in the previous loop search iteration is subsumed by at least one global clause with empty $\mathbf{X}$ part that was derived in the current loop search iteration (lines 13–15). Subsumption between a pair of clauses corresponds to an instance of production rule [BFS-loop-it-sub]; note, though, that this rule does not produce a new clause but records a relation between two clauses to be used later for extraction of a UC. Loop search for c terminates, if either a loop has been found or no clauses with empty $\mathbf{X}$ part were derived (line 16). If a loop has been found, rules [BFS-loop-conclusion1] and [BFS-loop-conclusion2] are applied once to each global clause with empty $\mathbf{X}$ part that was derived in the current loop search iteration (line 18) to obtain the loop search conclusions for the main partition.

IV. UC EXTRACTION

In this section we describe, given an unsatisfiable set of SNF clauses C, how to obtain a subset of C, C^{uc}, that is by itself unsatisfiable from an execution of the algorithm in Fig. 1. The general idea of the construction is unsurprising in that during the execution of the algorithm in Fig. 1 a resolution graph is built that records which clauses were used to generate other clauses (Def. 1). Then the resolution graph is traversed backwards from the empty clause to find the subset of C that was actually used to prove unsatisfiability (Def. 2). The main concern of Def. 1, 2, and their proof of correctness in Thm. 1 (see App. A of [35]) is therefore that/why certain parts of the TR proof do not need to be taken into account when determining C^{uc}. Remark 1 complements this by showing for other parts of the TR proof that they are indeed required to obtain C^{uc}. Finally, in Remark 2, the specifics of TR in the algorithm in Fig. 1 and of Def. 1, 2 are used to optimize construction of the resolution graph.

Definition 1: A *resolution graph* G is a directed graph consisting of (i) a set of vertices V, (ii) a set of directed edges $E \subseteq V \times V$, (iii) a labeling of vertices with SNF clauses $L_V : V \rightarrow \mathbb{C}$, and (iv) a partitioning $\mathcal{Q}^V$ of the set of vertices V into one main partition M^V and one partition L_i^V for each BFS loop search iteration: $\mathcal{Q}^V : V = M^V \uplus L_0^V \uplus \ldots \uplus L_n^V$. Let C be a set of SNF clauses. During an execution of the algorithm in Fig. 1 with input C a resolution graph G is constructed as follows. In line 1 G is initialized: (i) V contains one vertex v per clause c in C: $V = \{v_c \mid c \in C\}$, (ii) E is empty: $E = \emptyset$, (iii) each vertex is labeled with the

corresponding clause: $L_V : V \to C, L_V(v_c) = c$, and (iv) the partitioning $\mathcal{Q}^V$ contains only the main partition M^V, which contains all vertices: $\mathcal{Q}^V : M^V = V$. Whenever a new BFS loop search iteration is entered (line 11), a new partition L_i^V is created and added to $\mathcal{Q}^V$. For each application of a production rule from Tab. I that either generates a new clause in partition M or L or is the first application of rule BFS-loop-it-sub to clause c'' in C'' in line 15: (i) if column 11 (*vt. c*) of Tab. I contains ✔, then a new vertex v is created for the conclusion c (which is a new clause), labeled with c, and put into partition M^V or L_i^V; (ii) if column 9 (*p.1 – c*) (resp. column 10 (*p.2 – c*)) contains ✔, then an edge is created from the vertex labeled with premise 1 (resp. premise 2) in partition M^V or L_i^V to the vertex labeled with the conclusion in partition M^V or L_i^V.

Definition 2: Let C be a set of SNF clauses to which the algorithm in Fig. 1 has been applied and shown unsatisfiability, let G be the resolution graph, and let $v_\square$ be the (unique) vertex in the main partition M^V of the resolution graph G labeled with the empty clause $\square$. Let G' be the smallest subgraph of G that contains $v_\square$ and all vertices in G (and the corresponding edges) that are backward reachable from $v_\square$. The *UC of C in SNF*, C^{uc}, is the subset of C such that there exists a vertex v in the subgraph G', labeled with $c \in C$, and contained in the main partition M^V of G: $C^{uc} = \{c \in C \mid \exists v \in V_{G'} \,.\, L_V(v) = c \wedge v \in M^V\}$.

Theorem 1: Let C be a set of SNF clauses to which the algorithm in Fig. 1 has been applied and shown unsatisfiability, and let C^{uc} be the UC of C in SNF. Then C^{uc} is unsatisfiable.

Assume for a moment that in columns 9 (*p.1 – c*) and 10 (*p.2 – c*) of Tab. I all ✖ are replaced with ✔, i.e., that each conclusion in the resolution graph is connected by an edge to each of its premises rather than only to a subset of them. In that case the UC in SNF according to Def. 2 would contain all clauses of the set of starting clauses C that contributed to deriving the empty clause and, hence, to establishing unsatisfiability of C. In that case it would follow directly from the correctness of TR that C^{uc} is unsatisfiable. It remains to show that not including an edge (i) from premise 1 to the conclusion for rule aug2, (ii) from premise 2 to the conclusion for rule BFS-loop-conclusion2, (iii) from premise 2 to the conclusion for rule BFS-loop-it-init-c, and (iv) from premise 1 to the conclusion for rule BFS-loop-it-init-c in the resolution graph G maintains the fact that the resulting C^{uc} is unsatisfiable. To see the intuition behind (i) note that for a vertex v_c labeled with a conclusion c of rule aug2 in the main partition M^V to be backward reachable from the (unique) vertex in the main partition M^V of the resolution graph G labeled with the empty clause $\square$, $v_\square$, the occurrence of $\neg wl$ in c must be "resolved away" at some point on the path from v_c to $v_\square$. It turns out that this can only happen by resolution with a clause that is derived from the conclusion of rule aug1 applied to an eventuality clause c' with eventuality literal l. By construction of the resolution graph G $v_{c'}$ must be backward reachable from $v_\square$ and, therefore, c' must be included in the UC in SNF. Hence, an execution of the algorithm in Fig. 1 with input C^{uc} will produce c from c'. A similar reasoning as for (i) applies to (ii). For (iii) note that a conclusion of rule BFS-loop-it-init-c can only be backward reachable from $v_\square$ if the corresponding BFS loop search iteration is successful and a vertex labeled with one of the resulting conclusions of rules BFS-loop-conclusion1 or BFS-loop-conclusion2 is backward reachable from $v_\square$. The latter fact implies that an eventuality clause with the same eventuality literal as in premise 2 of rule BFS-loop-it-init-c is present in the UC in SNF. Hence, an execution of the algorithm in Fig. 1 with input C^{uc} will produce premise 2 of BFS-loop-it-init-c as required. Finally, (iv) is obtained by understanding that in a BFS loop search iteration the premises 1 of rule BFS-loop-it-init-c essentially constitute a hypothetical fixed point; if the BFS loop search iteration is successful, then the hypothetical fixed point is proven to be an actual fixed point. For the correctness of a proof of unsatisfiability of C it is only relevant that this hypothetical fixed point is shown to be an actual fixed point but not how the hypothesis is obtained. For a formalization of the above reasoning see App. A of [35].

By taking the fact that each vertex in the resolution graph has at most 2 incoming edges into account, the first part of the following Prop. 1 is immediate from Def. 1 and 2. The second part is obtained by bounding the number of (i) different clauses in each partition, (ii) iterations in each loop search by the length of the longest monotonically increasing sequence of Boolean formulas over AP, and (iii) loop searches by the number of different loop search conclusions.

Proposition 1: Let C be a set of SNF clauses to which the algorithm in Fig. 1 is applied and shows unsatisfiability. Construction and backward traversal of the resolution graph and, hence, construction of C^{uc} according to Def. 2 can be performed in time $\mathcal{O}(|V|)$ in addition to the time required to run the algorithm in Fig. 1. $|V|$ is at most exponential in $|AP| + log(|C|)$.

Remark 1: Theorem 1 shows that not including premises for production rules marked by ✖ in columns 9 (*p.1 – c*) and 10 (*p.2 – c*) of Tab. I during the construction of the resolution graph still leads to a UC. It does not discuss whether the remaining premises, marked by ✔ in columns 9 (*p.1 – c*) and 10 (*p.2 – c*) of Tab. I, actually need to be included to guarantee a UC. For all premises of all production rules marked by ✔ in columns 9 (*p.1 – c*) and 10 (*p.2 – c*) of Tab. I it turns out that they are indeed required to obtain a UC. The proof in App. A of [35] is essentially obtained by providing suitable examples.

Remark 2: The specifics of TR in the algorithm in Fig. 1 and the fact that not all premises need to be included during the construction of the resolution graph allow to optimize extraction of UCs by pruning the resolution graph during the execution of the algorithm in Fig. 1 extended with the construction in Def. 1, 2 as follows. (i) Notice that after the completion of a (successful or unsuccessful) loop search for some eventuality clause c in lines 9–19 of the algorithm in Fig. 1 no new edges between the main partition and one of the partitions used during the just completed loop search for c will be created. Hence, after completion of an execution of lines 9–19 of the algorithm in Fig. 1 vertices not backward reachable

TABLE II
TRANSLATION FROM LTL TO SNF.

Subf.	Prop.	SNF Clauses ($+$ polarity occurrences)	SNF Clauses ($-$ polarity occurrences)
$1/0/p$	$1/0/p$	—	—
$\neg\psi$	$x_{\neg\psi}$	$(\mathbf{G}(x_{\neg\psi} \rightarrow (\neg \boxed{x_{\psi}})))$	$(\mathbf{G}((\neg x_{\neg\psi}) \rightarrow \boxed{x_{\psi}}))$
$\psi \wedge \psi'$	$x_{\psi\wedge\psi'}$	$(\mathbf{G}(x_{\psi\wedge\psi'} \rightarrow \boxed{x_{\psi}}))$, $(\mathbf{G}(x_{\psi\wedge\psi'} \rightarrow \boxed{x_{\psi'}}))$	$(\mathbf{G}((\neg x_{\psi\wedge\psi'}) \rightarrow ((\neg \boxed{x_{\psi}}) \vee (\neg \boxed{x_{\psi'}}))))$
$\psi \vee \psi'$	$x_{\psi\vee\psi'}$	$(\mathbf{G}(x_{\psi\vee\psi'} \rightarrow (\boxed{x_{\psi}} \vee \boxed{x_{\psi'}})))$	$(\mathbf{G}((\neg x_{\psi\vee\psi'}) \rightarrow (\neg \boxed{x_{\psi}})))$, $(\mathbf{G}((\neg x_{\psi\vee\psi'}) \rightarrow (\neg \boxed{x_{\psi'}})))$
$\mathbf{X}\psi$	$x_{\mathbf{X}\psi}$	$(\mathbf{G}(x_{\mathbf{X}\psi} \rightarrow (\mathbf{X} \boxed{x_{\psi}})))$	$(\mathbf{G}((\neg x_{\mathbf{X}\psi}) \rightarrow (\mathbf{X} \neg \boxed{x_{\psi}})))$
$\mathbf{G}\psi$	$x_{\mathbf{G}\psi}$	$(\mathbf{G}(x_{\mathbf{G}\psi} \rightarrow (\mathbf{X} x_{\mathbf{G}\psi})))$, $(\mathbf{G}(x_{\mathbf{G}\psi} \rightarrow \boxed{x_{\psi}}))$	$(\mathbf{G}((\neg x_{\mathbf{G}\psi}) \rightarrow (\mathbf{F} \neg \boxed{x_{\psi}})))$
$\mathbf{F}\psi$	$x_{\mathbf{F}\psi}$	$(\mathbf{G}(x_{\mathbf{F}\psi} \rightarrow (\mathbf{F} \boxed{x_{\psi}})))$	$(\mathbf{G}((\neg x_{\mathbf{F}\psi}) \rightarrow (\mathbf{X} \neg x_{\mathbf{F}\psi})))$, $(\mathbf{G}((\neg x_{\mathbf{F}\psi}) \rightarrow (\neg \boxed{x_{\psi}})))$
$\psi \mathbf{U} \psi'$	$x_{\psi\mathbf{U}\psi'}$	$(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow (\boxed{x_{\psi'}} \vee \boxed{x_{\psi}})))$, $(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow (\boxed{x_{\psi'}} \vee (\mathbf{X} x_{\psi\mathbf{U}\psi'}))))$, $(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow (\mathbf{F} \boxed{x_{\psi'}})))$	$(\mathbf{G}((\neg x_{\psi\mathbf{U}\psi'}) \rightarrow (\neg \boxed{x_{\psi'}})))$, $(\mathbf{G}((\neg x_{\psi\mathbf{U}\psi'}) \rightarrow ((\neg \boxed{x_{\psi}}) \vee (\mathbf{X} \neg x_{\psi\mathbf{U}\psi'}))))$
$\psi \mathbf{R} \psi'$	$x_{\psi\mathbf{R}\psi'}$	$(\mathbf{G}(x_{\psi\mathbf{R}\psi'} \rightarrow \boxed{x_{\psi'}}))$, $(\mathbf{G}(x_{\psi\mathbf{R}\psi'} \rightarrow (\boxed{x_{\psi}} \vee (\mathbf{X} x_{\psi\mathbf{R}\psi'}))))$	$(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow ((\neg \boxed{x_{\psi'}}) \vee (\neg \boxed{x_{\psi}}))))$, $(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow ((\neg \boxed{x_{\psi'}}) \vee (\mathbf{X} \neg x_{\psi\mathbf{R}\psi'}))))$, $(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow (\mathbf{F} \neg \boxed{x_{\psi'}})))$

from the main partition can be pruned from the resolution graph. (ii) Moreover, note that, because there is no edge from instances of premise 1 to the conclusion induced by production rule BFS-loop-it-init-c, there are no outgoing edges from a failed loop search iteration (lines 11–15 of the algorithm in Fig. 1). Therefore, if a loop search iteration fails, all vertices and edges in the partition of that loop search iteration can be pruned from the resolution graph right away.

V. FROM LTL TO SNF AND BACK

We use a structure-preserving translation to translate an LTL formula into a set of SNF clauses, which slightly differs from the translation suggested in [25]. It is well known that ϕ and $SNF(\phi)$ according to Def. 3 are equisatisfiable and that a satisfying assignment for ϕ (resp. $SNF(\phi)$) can be extended (resp. restricted) to a satisfying assignment of $SNF(\phi)$ (resp. ϕ).

Definition 3: Let ϕ be an LTL formula over atomic propositions AP, and let $X = \{x, x', \ldots\}$ be a set of fresh atomic propositions not in AP. Assign each occurrence of a subformula ψ in ϕ a Boolean value or a proposition according to column 2 of Tab. II, which is used to reference ψ in the SNF clauses for its superformula. Moreover, assign each occurrence of ψ a set of SNF clauses according to column 3 or 4 of Tab. II. Let $SNF_{aux}(\phi)$ be the set of all SNF clauses obtained from ϕ that way. Then the *SNF of* ϕ is defined as $SNF(\phi) \equiv x_\phi \wedge \bigwedge_{c \in SNF_{aux}(\phi)} c$.

In the following Def. 4 we describe how to map a UC in SNF back to a UC in LTL.

Definition 4: Let ϕ be an unsatisfiable LTL formula, let $SNF(\phi)$ be its SNF, and let C^{uc} be the UC of $SNF(\phi)$ in SNF. Then the *UC of ϕ in LTL*, ϕ^{uc}, is obtained as follows. For each positive (resp. negative) polarity occurrence of a proper subformula ψ of ϕ with proposition x_ψ according to Tab. II, replace ψ in ϕ with 1 (resp. 0) iff C^{uc} contains no clause with an occurrence of proposition x_ψ that is marked blue boxed in Tab. II. (We are sloppy in that we "replace" subformulas of replaced subformulas, while in effect they simply vanish.)

Theorem 2: Let ϕ be an unsatisfiable LTL formula, and let ϕ^{uc} be the UC of ϕ in LTL. Then ϕ^{uc} is unsatisfiable.

Remark 3: In Def. 10 of [7] a UC of an unsatisfiable formula in LTL is obtained by replacing some occurrences of positive polarity subformulas with 1 and some occurrences of negative polarity subformulas with 0 while maintaining unsatisfiability. By construction in Def. 4 and with Thm. 2 it is immediate that a UC in LTL according to Def. 4 above is a UC according to Def. 10 of [7].

VI. MINIMAL UCS

In this section we introduce notions of and algorithms to obtain minimal UCs. The results are either straightforward (Remark 4) or well known (Def. 5, Remark 5). Still, the material is needed in the experimental evaluation and within the flow of the paper this seems to be the appropriate place.

Definition 5: (See, e.g., [7]: irreducible UC) A UC C^{uc} in SNF is *minimal* iff $\forall c \in C^{uc} \,.\, C^{uc} \setminus \{c\}$ is satisfiable. A UC ϕ^{uc} in LTL is *minimal* iff there is no positive polarity occurrence of a subformula that can be replaced with 1 and no negative polarity occurrence of a subformula that can be replaced with 0 without making ϕ^{uc} satisfiable.

Remark 4: Let ϕ be an unsatisfiable LTL formula, C its translation to SNF, C^{uc} a minimal UC of C in SNF, and ϕ^{uc} the UC of ϕ in LTL obtained by mapping C^{uc} back to LTL via Def. 4. Then ϕ^{uc} is not necessarily minimal.

Remark 5: A common way to obtain minimal UCs works by repeatedly attempting to remove parts of a UC (e.g., [23]). If the modified formula is still unsatisfiable, then the removal is made permanent; otherwise it is undone. The procedure continues until all parts of the UC have been considered for removal. This is called *deletion-based extraction of minimal UCs* (e.g., [23]). In the case of LTL the algorithm attempts to replace positive polarity occurrences of subformulas with 1 and negative polarity ones with 0. It terminates, if no more replacements can be performed without making the resulting formula satisfiable. This method may be expensive due to the required number of satisfiability tests, so it is often used to minimize a UC that has been obtained by other means such as those described in Sec. IV, V (e.g., [2], [3]).

VII. EXAMPLES

In this section we first present examples of using UCs for LTL to help understanding a specification given in LTL. Then we show an example of TR with the corresponding resolution graph and UC extraction in SNF.

The first example (1a)–(1c) is based on [42]. We would like to see whether a req (request) can be issued (1d). This is impossible, as (1a) requires a req to be followed by 3 gnts (grant), whereas (1b) forbids two subsequent gnts. The UC in (2) clearly shows this.

$$
\begin{aligned}
&(\mathbf{G}(req \rightarrow ((\mathbf{X}gnt) \wedge (\mathbf{XX}gnt) \wedge (\mathbf{XXX}gnt)))) && \text{(1a)}\\
\wedge\ &(\mathbf{G}(gnt \rightarrow \mathbf{X}\neg gnt)) && \text{(1b)}\\
\wedge\ &(\mathbf{G}(cancel \rightarrow \mathbf{X}((\neg gnt)\mathbf{U}go))) && \text{(1c)}\\
\wedge\ &(\mathbf{F}req) && \text{(1d)}
\end{aligned}
$$

$$(\mathbf{G}(req \rightarrow ((\mathbf{X}gnt) \wedge (\mathbf{XX}gnt)))) \wedge (\mathbf{G}(gnt \rightarrow \mathbf{X}\neg gnt)) \wedge (\mathbf{F}req) \quad (2)$$

The second example (3) in Fig. 2 is adapted from a lift specification in [43] (we used a somewhat similar example in [7]). The lift has two floors, indicated by f_0 and f_1. On each

$$
\begin{aligned}
&(\neg u) \wedge (f_0) \wedge (\neg b_0) \wedge (\neg b_1) \wedge (\neg up) && \text{(3a)}\\
&\wedge (\mathbf{G}((u \rightarrow \neg \mathbf{X} u) \wedge ((\neg \mathbf{X} u) \rightarrow u))) && \text{(3b)}\\
&\wedge (\mathbf{G}(f_0 \rightarrow \neg f_1)) && \text{(3c)}\\
&\wedge (\mathbf{G}((f_0 \rightarrow \mathbf{X}(f_0 \vee f_1)) \wedge (f_1 \rightarrow \mathbf{X}(f_0 \vee f_1)))) && \text{(3d)}\\
&\wedge (\mathbf{G}(u \rightarrow ((f_0 \rightarrow \mathbf{X} f_0) \wedge ((\mathbf{X} f_0) \rightarrow f_0)))) && \text{(3e)}\\
&\wedge (\mathbf{G}(u \rightarrow ((f_1 \rightarrow \mathbf{X} f_1) \wedge ((\mathbf{X} f_1) \rightarrow f_1)))) && \text{(3f)}\\
&\wedge (\mathbf{G}(((\neg u) \rightarrow ((b_0 \rightarrow \mathbf{X} b_0) \wedge ((\mathbf{X} b_0) \rightarrow b_0))))) && \text{(3g)}\\
&\wedge (\mathbf{G}(((\neg u) \rightarrow ((b_1 \rightarrow \mathbf{X} b_1) \wedge ((\mathbf{X} b_1) \rightarrow b_1))))) && \text{(3h)}\\
&\wedge (\mathbf{G}(((b_0 \wedge \neg f_0) \rightarrow \mathbf{X} b_0) \wedge ((b_1 \wedge \neg f_1) \rightarrow \mathbf{X} b_1))) && \text{(3i)}\\
&\wedge (\mathbf{G}((f_0 \wedge \mathbf{X} f_0) \rightarrow ((up \rightarrow \mathbf{X} up) \wedge ((\mathbf{X} up) \rightarrow up)))) && \text{(3j)}\\
&\wedge (\mathbf{G}((f_1 \wedge \mathbf{X} f_1) \rightarrow ((up \rightarrow \mathbf{X} up) \wedge ((\mathbf{X} up) \rightarrow up)))) && \text{(3k)}\\
&\wedge (\mathbf{G}(((f_0 \wedge \mathbf{X} f_1) \rightarrow up) \wedge ((f_1 \wedge \mathbf{X} f_0) \rightarrow \neg up))) && \text{(3l)}\\
&\wedge (\mathbf{G}((sb \rightarrow (b_0 \vee b_1)) \wedge ((b_0 \vee b_1) \rightarrow sb))) && \text{(3m)}\\
&\wedge (\mathbf{G}(((f_0 \wedge \neg sb) \rightarrow (f_0 \mathbf{U} (sb \mathbf{R} ((\mathbf{F} f_0) \wedge (\neg up)))))) && \text{(3n)}\\
&\wedge (\mathbf{G}(((f_1 \wedge \neg sb) \rightarrow (f_1 \mathbf{U} (sb \mathbf{R} ((\mathbf{F} f_0) \wedge (\neg up)))))) && \text{(3o)}\\
&\wedge (\mathbf{G}((b_0 \rightarrow \mathbf{F} f_0) \wedge (b_1 \rightarrow \mathbf{F} f_1))) && \text{(3p)}
\end{aligned}
$$

Fig. 2. A lift specification.

floor there is a button to call the lift (b_0, b_1). sb is 1 if some button is pressed. If the lift moves up, then up must be 1; if it moves down, then up must be 0. u switches turns between actions by users of the lift (u is 1) and actions by the lift (u is 0). For more details we refer to [43]. We first assume that an engineer is interested in seeing whether it is possible that b_1 is continuously pressed (4). As the UC (5) shows, this is impossible as b_1 must be 0 at the beginning.

$$\mathbf{G} b_1 \qquad (4) \qquad\qquad (\neg b_1) \wedge \mathbf{G} b_1 \qquad (5)$$

Now the engineer modifies her query such that b_1 is pressed only from time point 1 on (6). As shown by the UC in (7) that turns out to be impossible, too.

$$\mathbf{X}\mathbf{G} b_1 \qquad (6)$$

$$(\neg u) \wedge ((\neg b_1) \wedge ((\mathbf{G}((\neg u) \rightarrow ((\mathbf{X} b_1) \rightarrow b_1))) \wedge (\mathbf{X}\mathbf{G} b_1))) \qquad (7)$$

The engineer now tries to have b_1 pressed from time point 2 on and, again, obtains a UC. She becomes suspicious and checks whether b_1 can be pressed at all (8). She sees that it cannot and, therefore, this specification of a lift must contain a bug. She can now use the UC in (9a)–(9f) to track down the problem. This example illustrates the use of UCs for debugging, as (9a)–(9f) is significantly smaller than (3).

$$\mathbf{F} b_1 \qquad (8)$$

$$
\begin{aligned}
&(f_0) \wedge (\neg b_1) \wedge (\neg up) && \text{(9a)}\\
&\wedge (\mathbf{G}(f_0 \rightarrow \neg f_1)) && \text{(9b)}\\
&\wedge (\mathbf{G}(f_0 \rightarrow \mathbf{X}(f_0 \vee f_1))) && \text{(9c)}\\
&\wedge (\mathbf{G}((f_0 \wedge \mathbf{X} f_0) \rightarrow ((\mathbf{X} up) \rightarrow up))) && \text{(9d)}\\
&\wedge (\mathbf{G}((f_0 \wedge \mathbf{X} f_1) \rightarrow up)) && \text{(9e)}\\
&\wedge (\mathbf{G}(b_1 \rightarrow \mathbf{F} f_1)) && \text{(9f)}\\
&\wedge (\mathbf{F}(b_1)) && \text{(9g)}
\end{aligned}
$$

In Fig. 3 we show an example of an execution of the TR algorithm with the corresponding resolution graph and UC extraction in SNF. The set of starting clauses C to be solved is $\mathbf{G}(a \vee \neg b)$, $\mathbf{G}(a \vee b \vee \mathbf{X}(a \vee b))$, $\mathbf{G}((\neg a) \vee \mathbf{X} a)$, $\mathbf{G}((\neg a) \vee \mathbf{F} \neg a)$, shown in the first row from the bottom in the rectangle shaded in light red. In Fig. 3 TR generally proceeds from bottom to top; in the top right corner the empty clause $\square$ is generated, indicating unsatisfiability. Clauses are connected with directed edges from premises to conclusions according to columns 9, 10 in Tab. I. Edges are labeled with production rules, where "BFS-loop" is abbreviated to "loop", "init" to "i", and "conclusion" to "conc". Saturation in line 2 of the algorithm in Fig. 1 produces $\mathbf{G}(a \vee b \vee \mathbf{X} a)$ in the second row from the bottom.[1] The other 2 clauses in that row are generated by augmentation (line 3). The following saturation (line 4) produces no new clauses. The dark green shaded rectangle is the loop partition for the first loop search iteration. Row 3 contains the clauses obtained by initialization of the BFS loop search iteration (line 11). Note that clause $\mathbf{G}(\mathbf{X} \neg a)$, generated by BFS-loop-it-init-c, has no edge coming in from the main partition. Row 4 then contains the clauses generated from those in row 3 by saturation restricted to step-xx (line 12). The subsumption test fails in this iteration, as none of the clauses in row 4 subsumes the empty clause (lines 13–15). The light green shaded rectangle is the loop partition for the second loop search iteration. Row 5 contains the clauses obtained by initialization and row 6 those obtained from them by restricted saturation. This time the subsumption test succeeds, and the loop search conclusions are shown in row 7 (line 18). Finally, row 8 contains the derivation of the empty clause $\square$ via saturation (line 19). The thick, dotted, blue clauses and edges show the part of the resolution graph that is backward reachable from $\square$. As all starting clauses in C are backward reachable from $\square$, the UC of C in SNF is C (note that this example serves to illustrate the mechanism rather than the benefit of UC extraction).

For a complete example that includes translation between LTL and SNF and leads to a proper UC see App. E of [35].

TABLE III
OVERVIEW OF BENCHMARK FAMILIES.

category	family	source	# s. no UC	# s. UC	# s. minimal UC	|largest solved|
application	**alaska_lift**	[43], [44]	75	72	72	4605
	anzu_genbuf	[12]	16	16	16	1924
	forobots	[45]	25	25	25	635
crafted	**schuppan_O1formula**	[29]	27	27	27	4006
	schuppan_O2formula	[29]	8	8	8	91
	schuppan_phltl	[29]	4	4	4	125
random	**rozier_random**	[46]	62	62	62	157
	trp	[47]	397	397	330	1421

VIII. Experimental Evaluation

We implemented extraction of UCs as described in Sec. IV, V in TRP++. We also implemented deletion-based minimization of UCs obtained with the previous method (Sec. VI).

Our examples are based on [29]. For all benchmark families that consist of a sequence of instances of increasing difficulty we stopped after two instances that could not be solved due to time or memory out. Some instances were simplified to 0 during the translation from LTL to SNF; these instances were discarded. In Tab. III we give an overview of the benchmark families. Columns 1–3 give the category, name, and the source of the family. Columns 4–6 list the numbers of instances that were solved by our implementation without UC extraction, with UC extraction, and with minimal UC extraction. Column 7 indicates the size (number of nodes in the syntax tree) of the largest instance solved without UC extraction.

The experiments were performed on a laptop with Intel Core i7 M 620 processor at 2 GHz running Ubuntu 12.04. The time

[1] While it may seem that some clauses are not considered for loop initialization or saturation, this is due to either subsumption (e.g., $\mathbf{G}(a \vee b \vee \mathbf{X}(a \vee b))$ by $\mathbf{G}(a \vee b \vee \mathbf{X} a)$) or the fact that TRP++ uses *ordered* resolution (e.g., $\mathbf{G}(a \vee b \vee \mathbf{X} a)$ with $\mathbf{G}(\neg w \overline{a} \vee \mathbf{X}((\neg a) \vee w \overline{a}))$; [27]). Both are issues of completeness of TR and not discussed in this paper.

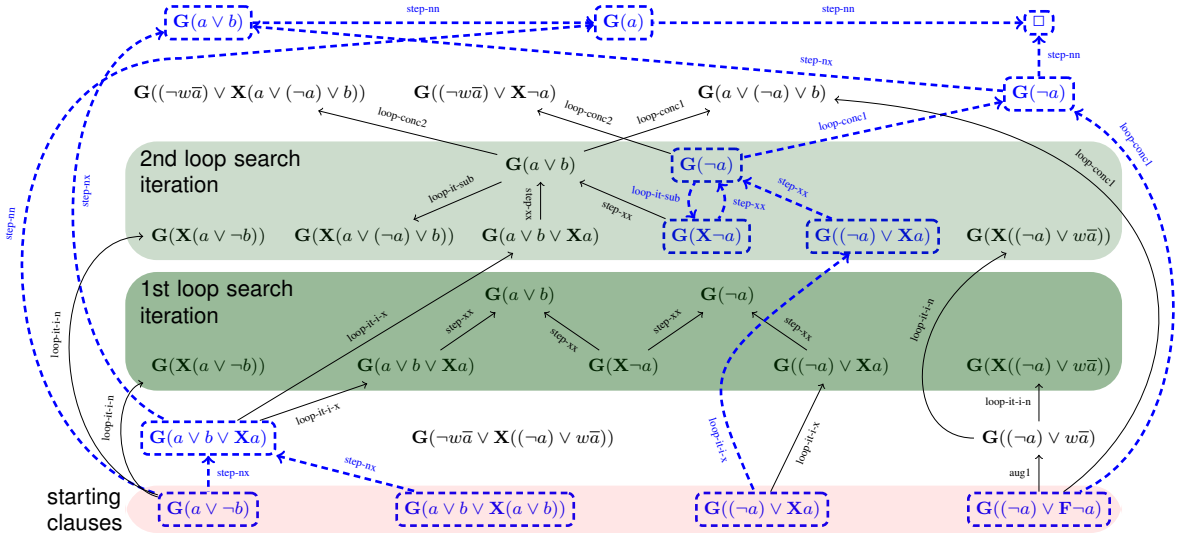

Fig. 3. Example of an execution of the TR algorithm with corresponding resolution graph and UC extraction in SNF.

and memory limits were 600 seconds and 6 GB.

In Fig. 4 (a), (b) we show the overhead that is incurred by extracting (non-minimal) UCs as described in Sec. IV, V over not extracting UCs. In Fig. 4 (c) we compare the sizes of the input formulas with the sizes of their (non-minimal) UCs. Our data show that extraction of UCs is possible with quite acceptable overhead in run time and memory usage. The resulting UCs are often significantly smaller than the input formula.

Figure 4 (d)–(f) show the costs and benefits of applying deletion-based minimization (Sec. VI) to (non-minimal) UCs obtained as described in Sec. IV, V. Costs and benefits are somewhat varied. Minimal UCs can be computed for all instances for which (non-minimal) UCs were obtained except for all 67 instances in family **trp_N12y**.

In Fig. 4 (g)–(l) we show the benefit of the optimizations described in Sec. IV when extracting (non-minimal) UCs. We show the impact on the peak size of the resolution graph rather than on run time or memory, as the former is implementation independent. The impact of including premise 1 of BFS-loop-it-init-c during construction of the resolution graph and disabling immediate pruning of vertices and edges in partitions of failed loop search iterations from the resolution graph in Fig. 4 (h) (the former implies the latter) and of disabling pruning non-reachable vertices from the resolution graph between loop searches in Fig. 4 (k) is quite significant. The impact in the remaining cases (Fig. 4 (g), (i), (j)) is negligible. However, in cases (i) and (j) there is an instance where disabling the optimization leads to a larger (non-minimal) UC. This occurs more often also in case (h).

IX. Conclusions

In this paper we showed how to obtain UCs for LTL via temporal resolution, and we demonstrated with an implementation in TRP++ that UC extraction can be performed efficiently. The resulting UCs are significantly smaller than the corresponding input formulas. In parallel work [33] this paper has been used as a basis to suggest enhancing UCs for LTL with information on when subformulas of a UC are relevant for unsatisfiability. The similarity of temporal resolution and some BDD-based algorithms at a high level and work on resolution with BDDs ([48]) suggests to explore whether computation of UCs is feasible for BDD-based algorithms. Another direction for transfer of our results is resolution-based computation of unrealizable cores [49].

Acknowledgments

I thank B. Konev and M. Ludwig for making TRP++ and TSPASS available. I also thank A. Cimatti for bringing up the subject of temporal resolution. Initial parts of the work were performed while working under a grant by the Provincia Autonoma di Trento (project EMTELOS).

References

[1] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, 2002.
[2] J. Chinneck and E. Dravnieks, "Locating Minimal Infeasible Constraint Sets in Linear Programs," *ORSA J. on Computing*, vol. 3, no. 2, 1991.
[3] R. Bakker *et al.*, "Diagnosing and Solving Over-Determined Constraint Satisfaction Problems," in *IJCAI*, 1993.
[4] D. Whalley, "Automatic Isolation of Compiler Errors," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, 1994.
[5] R. Bruni and A. Sassano, "Restoring Satisfiability or Maintaining Unsatisfiability by finding small Unsatisfiable Subformulae," in *SAT*, ser. Electronic Notes in Discrete Mathematics, vol. 9. Elsevier, 2001.
[6] I. Shlyakhter *et al.*, "Debugging Overconstrained Declarative Models Using Unsatisfiable Cores," in *ASE*, 2003.
[7] V. Schuppan, "Towards a notion of unsatisfiable and unrealizable cores for LTL," *Sci. Comput. Program.*, vol. 77, no. 7-8, 2012.
[8] A. Cimatti *et al.*, "Diagnostic Information for Realizability," in *VMCAI*, ser. LNCS, vol. 4905. Springer, 2008.
[9] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. Springer'06.
[10] M. Pesic and W. van der Aalst, "A Declarative Approach for Flexible Business Processes Management," in *Business Process Management Workshops*, ser. LNCS, vol. 4103. Springer, 2006.
[11] I. Beer *et al.*, "Efficient Detection of Vacuity in Temporal Model Checking," *FMSD*, vol. 18, no. 2, 2001.
[12] R. Bloem *et al.*, "Specify, Compile, Run: Hardware from PSL," in *COCV*, ser. ENTCS, vol. 190(4). Elsevier, 2007.

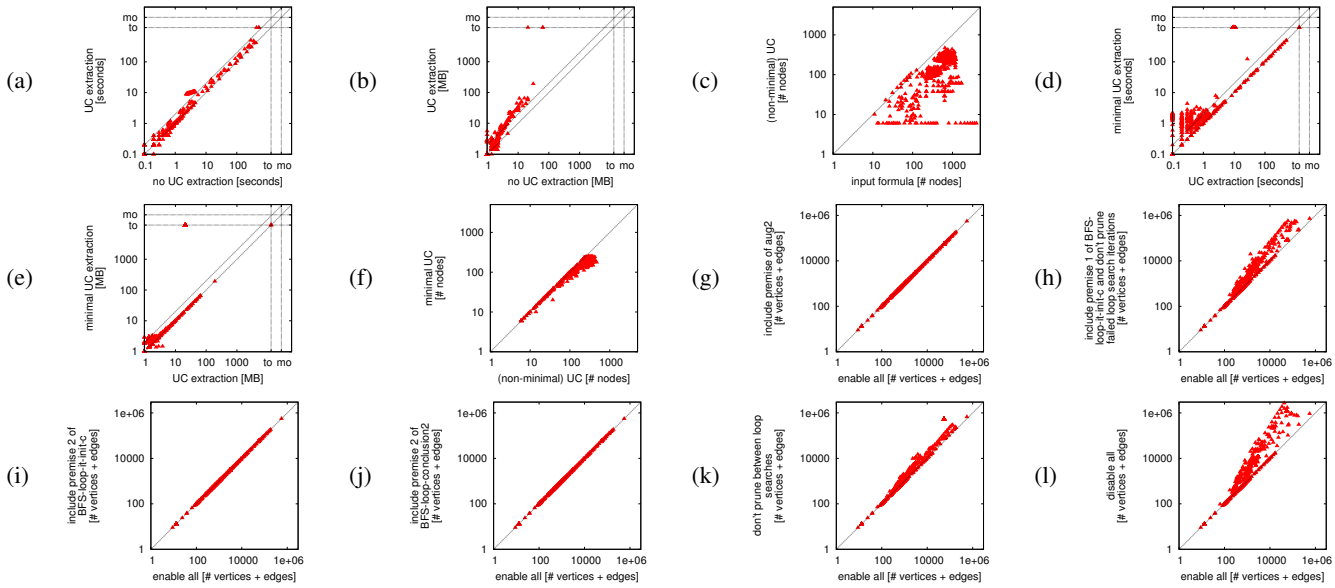

Fig. 4. (a)–(c) compare UC extraction (y-axis) with no UC extraction (x-axis). (a) and (b) show the overhead incurred in terms of run time (in seconds) and memory (in MB). (c) shows the size reduction obtained, where size is measured as the number of nodes in the syntax trees. (d)–(f) compare minimal UC extraction (y-axis) with UC extraction (x-axis). (d) and (e) show overhead incurred in terms of run time (in seconds) and memory (in MB). (f) shows the size reduction obtained, where size is measured as the number of nodes in the syntax trees. The off-center diagonal in (a), (b), (d), and (e) shows where $y = 2x$. (g)–(l) show the benefit of optimizations as reduction in peak size of resolution graph (number of vertices + number of edges). The x-axis shows all optimizations enabled. The y-axis of (g)–(k) shows one optimization disabled: (g) include premise of aug2, (h) include premise 1 of BFS-loop-it-init-c and disable immediate pruning of failed loop search iterations, (i) include premise 2 of BFS-loop-it-init-c, (j) include premise 2 of BFS-loop-conclusion2, (k) disable pruning of the resolution graph between loop searches. The y-axis of (l) shows all optimizations disabled.

[13] H. Hoos, "Heavy-Tailed Behaviour in Randomised Systematic Search Algorithms for SAT?" Univ. of British Columbia, Dept. of Computer Science, Tech. Rep. TR-99-16, 1999.

[14] I. Pill *et al.*, "Formal analysis of hardware requirements," in *DAC*, 2006.

[15] J. Simmonds *et al.*, "Exploiting resolution proofs to speed up LTL vacuity detection for BMC," *STTT*, vol. 12, no. 5, 2010.

[16] A. Awad *et al.*, "An iterative approach to synthesize business process templates from compliance rules," *Inf. Syst.*, vol. 37, no. 8, 2012.

[17] A. Chiappini *et al.*, "Formalization and validation of a subset of the European Train Control System," in *ICSE (2)*, 2010.

[18] E. Clarke *et al.*, "SAT Based Predicate Abstraction for Hardware Verification," in *SAT*, ser. LNCS, vol. 2919. Springer, 2003.

[19] A. Cimatti *et al.*, "Boolean Abstraction for Temporal Logic Satisfiability," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007.

[20] A. Van Gelder, "Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs both Preorder and Postorder Resolution," in *AMAI*, 2002.

[21] F. Hantry and M. Hacid, "Handling Conflicts in Depth-First Search for LTL Tableau to Debug Compliance Based Languages," in *FLACOS*, ser. EPTCS, vol. 68, 2011.

[22] F. Hantry, L. Saïs, and M. Hacid, "On the complexity of computing minimal unsatisfiable LTL formulas," *ECCC*, vol. 19, no. 69, 2012.

[23] J. Marques Silva, "Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper)," in *ISMVL*, 2010.

[24] M. Fisher, "A Resolution Method for Temporal Logic," in *IJCAI*, 1991.

[25] M. Fisher, C. Dixon, and M. Peim, "Clausal temporal resolution," *ACM Trans. Comput. Log.*, vol. 2, no. 1, 2001.

[26] U. Hustadt and B. Konev, "TRP++: A temporal resolution prover," in *Collegium Logicum*. Kurt Gödel Society, 2004, vol. 8.

[27] ——, "TRP++ 2.0: A Temporal Resolution Prover," in *CADE*, ser. LNCS, vol. 2741. Springer, 2003.

[28] http://www.csc.liv.ac.uk/~konev/software/trp++/.

[29] V. Schuppan and L. Darmawan, "Evaluating LTL Satisfiability Solvers," in *ATVA*, ser. LNCS, vol. 6996. Springer, 2011.

[30] A. Cimatti *et al.*, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002.

[31] A. Heuerding *et al.*, "Propositional Logics on the Computer," in *TABLEAUX*, ser. LNCS, vol. 918. Springer, 1995.

[32] http://users.cecs.anu.edu.au/~rpg/PLTLProvers/.

[33] V. Schuppan, "Enhancing Unsatisfiable Cores for LTL with Information on Temporal Relevance," in *QAPL*, ser. EPTCS, vol. 117, 2013.

[34] A. Gurfinkel and M. Chechik, "How Vacuous Is Vacuous?" in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004.

[35] V. Schuppan, "Extracting unsatisfiable cores for LTL via temporal resolution (full version)," 2013, http://www.schuppan.de/viktor/time13/VSchuppan-TIME-2013-full.pdf.

[36] A. Cimatti, S. Mover, and S. Tonetta, "Proving and explaining the unfeasibility of message sequence charts for hybrid systems," in *FMCAD*. FMCAD Inc., 2011.

[37] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications using simple counterstrategies," in *FMCAD*, 2009.

[38] ——, "Debugging Unrealizable Specifications with Model-Based Diagnosis," in *HVC*, ser. LNCS, vol. 6504. Springer, 2010.

[39] V. Raman and H. Kress-Gazit, "Analyzing Unsynthesizable Specifications for High-Level Robot Behavior Using LTLMoP," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011.

[40] http://www.schuppan.de/viktor/time13/.

[41] E. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. Elsevier and MIT Press, 1990.

[42] B. Jobstmann and R. Bloem, "Optimizations for LTL Synthesis," in *FMCAD*, 2006.

[43] A. Harding, "Symbolic Strategy Synthesis For Games With LTL Winning Conditions," Ph.D. dissertation, University of Birmingham, 2005.

[44] M. De Wulf *et al.*, "Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008.

[45] A. Behdenna, C. Dixon, and M. Fisher, "Deductive Verification of Simple Foraging Robotic Behaviours," *Int. J. of Intelligent Comput. and Cybernetics*, vol. 2, no. 4, 2009.

[46] K. Rozier and M. Vardi, "LTL satisfiability checking," *STTT*, vol. 12, no. 2, 2010.

[47] U. Hustadt and R. A. Schmidt, "Scientific Benchmarking with Temporal Logic Decision Procedures," in *KR*. Morgan Kaufmann, 2002.

[48] T. Jussila, C. Sinz, and A. Biere, "Extended Resolution Proofs for Symbolic SAT Solving with Quantification," in *SAT*, ser. LNCS, vol. 4121. Springer, 2006.

[49] P. Noël, "A Transformation-Based Synthesis of Temporal Specifications," *Formal Asp. Comput.*, vol. 7, no. 6, 1995.