Abstract. When creating a knowledge base of propositional linear temporal logic (PLTL) formulae as a specification, it is easy to introduce inconsistency, particularly if the knowledge base is created by multiple parties. Identifying the exact formulae which lead to inconsistency is a difficult but important problem which allows us to diagnose the source of the error more easily. We describe a method which uses binary decision diagrams to find a minimal PLTL-unsatisfiable subset of a given inconsistent knowledge base, thereby removing any formulae which do not directly contribute to the inconsistency. Our method extends an existing BDD-based method for finding minimal unsatisfiable subsets in classical propositional logic and requires only a single call to a BDD-based theorem prover. We also present a more efficient hybrid method which uses an existing resolution-based method to first find a, not necessarily minimal, PLTL-unsatisfiable subset and then uses our BDD-based method to refine this set to a minimal PLTL-unsatisfiable subset of the knowledge base. We also give experimental results.

1 Introduction

Propositional Linear temporal logic (PLTL) is an extension of classical propositional logic which is well suited to specifying and verifying properties of reactive systems [1]. Traditionally, PLTL has been used to verify properties of digital circuits, but more recently it has been used to reason about business processes [2]. In these applications, a specification is built up out of many smaller formulae describing individual aspects of the digital circuit or business process. While each of these may appear sensible when considered in isolation, it is common for clashes to occur when conjoining the formulae to form a complete specification, particularly if the components are created by multiple parties. The clashes manifest themselves by making the specification inconsistent, meaning that we can deduce any formula we like from this faulty specification. To proceed, we must remove these clashes and restore the consistency of the specification.

Due to the complexity of the interactions between PLTL formulae, and the level of abstraction involved with the PLTL representation, manually debugging such a faulty specification is difficult. As it is likely that only a small subset of the specification actually contributes to the inconsistency, it is desirable to find
this faulty subset by automatically discarding any formulae which do not directly contribute to the problem. The reduction to a minimal faulty core greatly reduces the effort required to debug and fix the issue.

From a semantic viewpoint, we wish to reduce a PLTL-unsatisfiable (inconsistent) set of input formulae to some subset that is as small as possible while still remaining PLTL-unsatisfiable (inconsistent). An obvious naive approach would be to remove formulae one by one, calling a PLTL-satisfiability solver each time to determine whether the remainder is still PLTL-unsatisfiable. However, this approach will quickly become impractical, as PLTL is known to be PSPACE-complete [3].

Here, we describe an alternate method which can be used to minimise an PLTL-unsatisfiable set of PLTL formulae. We use Binary Decision Diagrams (BDDs) as a data structure which provides an efficient representation of the set of all models of the input set, together with a BDD-based theorem prover for PLTL. We are then able to adapt an existing method for finding minimal unsatisfiable subsets of classical propositional logic [4] to PLTL, allowing a minimal PLTL-unsatisfiable subset to be extracted with only a single call to the theorem prover.

Schuppan [5] shows that a method based upon temporal resolution can be used to reduce a PLTL-unsatisfiable set of PLTL formulae into a smaller, but not necessarily minimal, set of PLTL-unsatisfiable formulae. He uses the term “unsatisfiable cores” for the resulting set.

Awad et al [2] have given an alternative method for computing minimal PLTL-unsatisfiable subsets using a theorem prover for PLTL based upon the tableau method.

When we evaluated our method against these other methods, we found that it generally out-performed the method based upon tableaux, but it did poorly against the resolution-based method. The last comparison is not quite fair since the resolution-method does not compute minimal subsets, but it opens the door to a hybrid approach: first use the resolution-based method to find a PLTL-unsatisfiable core, then use our BDD-based approach to minimise this core.

We show that our hybrid approach gives significant reductions in a large number of cases since the resolution-based method rarely finds minimal PLTL-unsatisfiable cores. Our hybrid method also outperforms the tableau-based method.

The remainder of the paper is organised as follows. First, we introduce the syntax and semantics of PLTL in Section 2. In Section 3, we provide a precise definition of minimal PLTL-unsatisfiability, and then describe our algorithm for reducing an input set of PLTL formulae to a minimal PLTL-unsatisfiable subset in Section 4. Section 5 then experimentally evaluates this approach. Finally, we describe other related work in Section 6 and possible improvements in Section 7 and before concluding in Section 8.

2 Syntax and Semantics of PLTL

Let $Atm$ be a non-empty set of propositional variables. The set of PLTL formulae is defined using the following BNF grammar, where $p \in Atm$:
\( \varphi ::= p | T | \bot | \neg \varphi \land \psi \land \psi \land \psi \rightarrow \psi | X \varphi | F \varphi | G \varphi | \psi | B \varphi \)

PLTL formulae are interpreted over an infinite sequence of states \( S = (s_i)_{i \in \mathbb{N}} \), where each state is assigned a subset of \( Atm \), representing the propositions which are true at that moment in time. Let \( \vartheta \) be the interpretation function from states to subsets of \( Atm \).

We can then define a satisfaction relation \( \models \) between a pair \( I = (S, \vartheta) \), a state \( s_i \) and a formula \( \varphi \) as:

\[
\begin{align*}
I, s_i \models & T \quad \text{iff } p \in \vartheta(s_i) \\
I, s_i \not\models & \bot \\
I, s_i \models & p \quad \text{iff } p \in \vartheta(s_i) \\
I, s_i \not\models & \neg \varphi \quad \text{iff } I, s_i \not\models \varphi \\
I, s_i \models & \varphi \lor \psi \quad \text{iff } I, s_i \models \varphi \text{ or } I, s_i \models \psi \\
I, s_i \models & \varphi \land \psi \quad \text{iff } I, s_i \models \varphi \text{ and } I, s_i \models \psi \\
I, s_i \models & \varphi \rightarrow \psi \quad \text{iff } I, s_i \not\models \varphi \text{ or } I, s_i \models \psi \\
I, s_i \models & X \varphi \quad \text{iff } I, s_{i+1} \models \varphi \\
I, s_i \models & G \varphi \quad \text{iff } \text{for all } j \in \mathbb{N}, j \geq i \text{ implies } I, s_j \models \varphi \\
I, s_i \models & F \varphi \quad \text{iff } \text{there exists } j \in \mathbb{N} \text{ such that } j \geq i \text{ and } I, s_j \models \varphi \\
I, s_i \models & \psi \quad \text{iff } \text{there exists } j \in \mathbb{N} \text{ such that } j \geq i, I, s_j \models \psi, \text{ and for all } k \in \mathbb{N}, j > k \geq i \text{ implies } I, s_k \models \varphi \\
I, s_i \models & B \psi \quad \text{iff } I, s_i \models \varphi \text{ and } I, s_i \models G \varphi
\end{align*}
\]

An interpretation \( I \) PLTL-satisfies a formula \( \varphi \) if \( I, s_0 \models \varphi \). A formula \( \varphi \) is PLTL-satisfiable if there exists an interpretation \( I \) which PLTL-satisfies it, and is PLTL-unsatisfiable otherwise. A set of formulae \( \Gamma \) is PLTL-satisfiable (PLTL-unsatisfiable) if \( \bigwedge_{\varphi \in \Gamma} \varphi \) is PLTL-satisfiable (PLTL-unsatisfiable). Finally, note that \( F \varphi \equiv \top \lor \psi \) and \( G \varphi \equiv \bot \land \neg \varphi \).

Next, we briefly consider the notion of (local) logical consequence. Given a set of PLTL formulae \( \Gamma \) and a single PLTL formula \( \varphi \), we say that \( \varphi \) is a logical consequence of \( \Gamma \), written \( \Gamma \models \varphi \), if for all interpretations \( I \), if \( I, s_0 \models \Gamma \) then \( I, s_0 \models \varphi \). By definition, \( \Gamma \models \varphi \) exactly when \( \Gamma \cup \{ \neg \varphi \} \) is PLTL-unsatisfiable.

If \( \varphi \) encapsulates a desirable property that we expect to hold whenever \( \Gamma \) holds then we will ask whether \( \Gamma \models \varphi \) holds by testing whether \( \Gamma \cup \{ \neg \varphi \} \) is PLTL-unsatisfiable. If it is PLTL-satisfiable then there is something wrong with the knowledge base. Alternatively, if we want to check whether a knowledge base \( \Gamma \) permits a certain formula \( \varphi \) to be true, we test whether \( \Gamma \cup \{ \varphi \} \) is PLTL-satisfiable - a PLTL-unsatisfiable result indicates that something is wrong with the specification. In both cases, a problem occurs if \( \Gamma \) itself is PLTL-unsatisfiable since \( \Gamma \models \varphi \) will hold for all \( \varphi \), and \( \Gamma \cup \{ \varphi \} \) will be PLTL-unsatisfiable for all \( \varphi \). If \( \Gamma \) is PLTL-unsatisfiable then a minimal PLTL-unsatisfiable subset \( X \) of \( \Gamma \) would allow us to localise the problem to \( X \).

As a running example, consider the specification given in Example 1 below, adapted from [5]. Each formula within the specification is annotated with an English description of the meaning it encodes.
Example 1.

\[ \Gamma^a = \{ G (req \rightarrow ((X gnt) \land (X X gnt))) \} \] (1)
Whenever a request is made, it must be granted in the following two timesteps
\[ G (gnt \rightarrow X \neg gnt) \] (2)
Requests cannot be granted at two consecutive timesteps
\[ G (pause \rightarrow X (\neg gnt \lor resume)) \} \] (3)
If processing is paused, from the next timestep, no requests may be granted until processing is resumed

This specification is clearly faulty since the first formula requires \( gnt \) to be true in two consecutive states after \( req \) becomes true while the second formula forbids \( gnt \) from being true in any two consecutive states. Although each formula can be understood on its own, when conjoined to form a knowledge base, equations 1 and 2 conflict as soon as \( req \) becomes true. However, this knowledge base by itself is still PLTL-satisfiable - for example, consider an interpretation where no requests or grants are made at all. However, querying the knowledge base allows us to check whether certain situations are allowed, such as in example 2, where we test whether a request can be made at any point.

Example 2.

\[ \Gamma^b = \{ G (req \rightarrow ((X gnt) \land (X X gnt))), G (gnt \rightarrow X \neg gnt), G (pause \rightarrow X (\neg gnt \lor resume), F req) \} \]

The set \( \Gamma^b \) is PLTL-unsatisfiable since every interpretation for it will contain some state \( s_i \) that makes \( req \) true, which will fire formulae 1 and require \( gnt \) to be true at \( s_{i+1} \) and \( s_{i+2} \), which is forbidden by formula 2.

3 Minimal PLTL-Unsatisfiability

We first consider the definition of minimal unsatisfiability used in classical propositional logic (CPL). For \( a \in Atm \) both \( a \) and \( \neg a \) are literals. A clause is a disjunction of literals. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. It is well-known that every formula of classical propositional logic can be put into CNF while preserving logical equivalence. Minimal unsatisfiability is then defined in terms of the CNF treated as a set of clauses \( \{c_1, c_2, \cdots, c_n\} \).

**Definition 1.** Let \( \Delta \) be a CNF formula of CPL. Then \( \Delta \) is minimal unsatisfiable if \( \Delta \) is unsatisfiable and \( \forall c \in \Delta. \Delta \setminus c_i \) is satisfiable [4].
This is extended to PLTL by considering a set of PLTL formulae instead of a set of CNF clauses.

**Definition 2.** Let $\Gamma$ be a set of PLTL formulae. $\Gamma$ is a minimal PLTL-unsatisfiable set if $\Gamma$ is PLTL-unsatisfiable, and $\forall \varphi \in \Gamma. \Gamma \setminus \{\varphi\}$ is PLTL-satisfiable.

Thus, a set of formulae might have several minimal PLTL-unsatisfiable subsets, which may be different sizes. In this case, we consider a minimum PLTL-unsatisfiable subset to be a subset with the least cardinality.

**Definition 3.** A minimum PLTL-unsatisfiable subset of $\Gamma$ is a minimal PLTL-unsatisfiable subset $S$ of $\Gamma$ such that for every PLTL-unsatisfiable subset $S'$ of $\Gamma$, $|S'| \geq |S|$.

Note that minimum PLTL-unsatisfiable subsets are not necessarily unique. In this paper, we are only concerned with the extraction of minimal PLTL-unsatisfiable subsets from a set of input formulae – we do not attempt to find minimum PLTL-unsatisfiable subsets.

### 4 Finding Minimal PLTL-unsatisfiable Subsets

We will introduce the method used to reduce a set of PLTL formulae into a minimal PLTL-unsatisfiable subset in two parts. First, we will describe the general strategy, using a version of the method presented in [4] adapted for PLTL. Then, we will discuss specific details relating to implementing this method with BDDs.

#### 4.1 General strategy

Let $\Gamma = \{\varphi_1, \ldots, \varphi_m\}$ be the set of PLTL-formulae we want to reduce to minimal PLTL-unsatisfiability, and let $X$ be the set of propositional variables in $\Gamma$. Further, let $\Gamma_i = \Gamma \setminus \{\varphi_i\}$.

The first step in the process is to label each $\varphi_i$ with a set of propositional variables. We introduce a set $Y = \{y_1, \ldots, y_k\}$ of $k = \lceil \log(m+1) \rceil$ new variables. By conjoining these $k$ variables in different combinations of positive and negative literals, we can produce a total of $2^k$ “tag” formulae. For example, with $k = 2$, the resulting tags are $\{G(y_1 \land y_2), G(y_1 \land \neg y_2), G(\neg y_1 \land y_2), G(\neg y_1 \land \neg y_2)\}$. Let $T_i$ be the $i$-th such tag formula. Since $2^k \geq m$, we can thus produce a new set of PLTL-formulae $T\Gamma$ by tagging each PLTL-formula $\varphi_i$ in $\Gamma$ with $T_i$ such that $T\Gamma$ is also a set of PLTL-formulae:

$$T\Gamma = \{T_1 \lor \varphi_1, \ldots, T_m \lor \varphi_m\}$$

**Example 3.** We now augment the $\Gamma^b$ of Example 2 with tag variables, giving:

$$T\Gamma^b = \{G(y_1 \land y_2) \lor (G(req \rightarrow ((X\ gnt) \land (X\ X\ gnt)))),$$

$$G(y_1 \land \neg y_2) \lor (G(gnt \rightarrow X\ \neg gnt)),$$

$$G(\neg y_1 \land y_2) \lor (G(pause \rightarrow X\ (\neg gnt\ U\ resume))),$$

$$G(\neg y_1 \land \neg y_2) \lor (F\ req)\}$$
Consider an interpretation $I$ where all states use the same instantiation of the $Y$ variables such that exactly one $T_i \lor \varphi_i$ of $T \Gamma$ also evaluates to true under this interpretation, regardless of the contents of $\varphi_i$. Such interpretations allow us to remove one $\varphi_i$ at a time from $\Gamma$ by instantiating the $Y$ variables uniformly in all states.

If $\alpha$ is an instantiation of the $Y$ variables, we will write $\alpha(T \Gamma)$ to denote the subset of $T \Gamma$ obtained by removing the element $T_i \lor \varphi_i$ that evaluates to true as described above.

Example 4. Consider the $T \Gamma^b$ of Example 3, and an instantiation $\alpha$ that sets both $y_1$ and $y_2$ to true. Then, $T_1$, which is $G (y_1 \land y_2)$, evaluates to true under any interpretation that uses $\alpha$ in all states, while all other tag formulae evaluate to false under the same interpretation. The set $T \Gamma^b$ then simplifies to the following set of PLTL-formula:

$$\alpha(T \Gamma^b) = \{G (gnt \rightarrow X \neg gnt),$$
$$G (pause \rightarrow X (\neg gnt \lor \text{resume}),$$
$$F \text{req}\}$$

We now formalise the intuition given above that an instantiation of the $Y$ variables allows us to disable a single formula from $\Gamma$ at a time. Let $f : 2^Y \rightarrow \mathbb{N}$ be a function mapping an instantiation $\alpha$ of $Y$ to the index $f(\alpha)$ such that the tag formula $T_{f(\alpha)}$ evaluates to true, while all other tag formulae $T_j \neq f(\alpha)$ evaluate to false under any interpretation that uses $\alpha$ in all states. This has the effect of removing $\varphi_{f(\alpha)}$ from $\Gamma$ while retaining all other formulae, giving $\Gamma_{f(\alpha)}$. We record this as:

Lemma 1.

$$\alpha(\Gamma) = \begin{cases} 
\Gamma_{f(\alpha)} & \text{if } 1 \leq f(\alpha) \leq m \\
\Gamma & \text{otherwise}
\end{cases}$$

Our main theorem requires us to use a notion of variable forgetting adapted to PLTL. Specifically, given an interpretation $I(X, Y)$ over propositional variables $X \cup Y$, we write $\exists X. I(X, Y)$ to denote a new interpretation, over variables $Y$ only, that is identical to $I(X, Y)$ as far as the instantiations of variables $Y$ are concerned.

We are now ready to present our main theorem. Note that we sometimes slightly abuse notation and take a set of formulae $\Gamma$ to be the conjunction of all members of $\Gamma$.

Theorem 1. Let $X$ contain all the propositional variables from a finite set $\Gamma$ of PLTL-formulae of cardinality $m$. Then $\Gamma$ is minimal PLTL-unsatisfiable iff there are exactly $m$ interpretations $\exists X. I(X, Y)$ where $I(X, Y)$ is a model of $T \Gamma$.

The following proof adapts the proof given in [4] to PLTL.
Proof. Let $\alpha$ be one of the $2^k$ possible instantiations for the $k = \lceil \log (m+1) \rceil$ new propositional variables $Y = \{y_1, \ldots, y_k\}$ and let $X$ contain all the propositional variables from the formulae in $\Gamma$. Further, let $I_\alpha$ be an interpretation where all states instantiate the $Y$ variables according to $\alpha$.

First, assume that $\Gamma$ is minimal PLTL-unsatisfiable. That is, $\Gamma$ is PLTL-unsatisfiable, and for all $i$ such that $1 \leq i \leq m$, the subset $\Gamma_i$ is PLTL-satisfiable. Thus, according to Lemma 1, $\alpha(\Gamma)$ is satisfiable iff $1 \leq f(\alpha) \leq m$. In other words, there are exactly $m$ ways to choose an $\alpha$ such that $I_\alpha$ can satisfy $\Gamma$.

Next, we assume that there are exactly $m$ ways to choose an $\alpha$ such that $I_\alpha$ can satisfy $\Gamma$. It follows that $\Gamma$ is PLTL-unsatisfiable, as otherwise, we would have $2^k > m$ ways to choose an $\alpha$ such that $I_\alpha$ can satisfy $\Gamma$. Now, as $\Gamma$ is PLTL-unsatisfiable, $\alpha(\Gamma)$ is PLTL-unsatisfiable for all $f(\alpha) > m$. Thus, in order for there to be $m$ ways to choose an $\alpha$ such that $I_\alpha$ can possibly satisfy $\Gamma$, each $\Gamma_i$ must be PLTL-satisfiable (recall that $\alpha(\Gamma) = \Gamma_i$ for $1 \leq f(\alpha) \leq m$). Hence, $\Gamma$ is PLTL-unsatisfiable, but every $\Gamma_i$ is PLTL-satisfiable, and therefore $\Gamma$ is minimal PLTL-unsatisfiable. \( \square \)

Our Theorem 1 is a direct translation to handle PLTL from Huang’s version for CPL [4]. If we can count the models $\exists X.I(X, Y)$ of $TT\Gamma$ while forgetting variables $X$, Theorem 1 enables us to determine whether a given $\Gamma$ is minimal PLTL-unsatisfiable.

Example 5. Consider the $TT\Gamma$ given in Example 3. We try each of the 4 possible valuations for the 2 new variables in $Y = \{y_1, y_2\}$ in turn:

- $\alpha_1(\Gamma) = \Gamma_{y_1}^b$ is PLTL-satisfiable (put $req$ true at $s_0$)
- $\alpha_2(\Gamma) = \Gamma_{y_2}^b$ is PLTL-satisfiable (put $req$ true at $s_0$)
- $\alpha_3(\Gamma) = \Gamma_3$ is PLTL-unsatisfiable (since formulae 1 and 2 clash after $req$ eventually becomes true as demanded by formula 4)
- $\alpha_4(\Gamma) = \Gamma_4$ is PLTL-satisfiable (put $req$ false everywhere).

Thus, there are three CPL-valuations $\alpha_j$ with $j \in \{1, 2, 3\}$ of $Y$ such that $\alpha_j(\Gamma)$ is satisfiable, so $TT\Gamma$ has 3 models with variables $X$ forgotten. Since $|\Gamma| = 4 \neq 3$, Theorem 1 tells us that $\Gamma$ is not minimal PLTL-unsatisfiable. However, removing the third formula from $\Gamma$ to obtain $\Gamma_3^b$ would also result in a total of 3 models for $TT\Gamma_3^b$ with variables $X$ forgotten. As $|\Gamma_3^b| = 3$, Theorem 1 tells us that $\Gamma_3^b$ is a minimal PLTL-unsatisfiable set.

We are also able to use the tag formulae to determine which $\varphi_i$ can be removed without affecting PLTL-unsatisfiability. If $TT\Gamma$ remains PLTL-unsatisfiable after removing some $\varphi_i$ by assigning the appropriate values to the tag variables, then $\varphi_i$ can be safely removed without affecting PLTL-unsatisfiability.

By combining these two ideas, we are able to produce a minimal PLTL-unsatisfiable subset of $\Gamma$ by repeatedly checking for PLTL-unsatisfiability and then finding a formula to remove. We describe this in full in Algorithm 1 below. This algorithm will currently require multiple calls to the theorem prover for PLTL-unsatisfiability, however, the BDD-based method described in Section 4.2 allows this to be reduced to a single call.
Algorithm 1 Minimise($\Gamma$)

1. Let $\Gamma$ be $\{\varphi_1, \ldots, \varphi_m\}$ and let $X$ be the propositional variables in $\Gamma$
2. Let $Y = \{y_1, \ldots, y_k\}$ be $k = \lceil \log(m+1) \rceil$ new propositional variables
3. Let $T_i$ be the $i$-th tag and let $\Gamma_T = \{T_1 \lor \varphi_1, \ldots, T_m \lor \varphi_m\}$

while Number of CPL-valuations over $Y$ for $\exists X. \Gamma_T \neq |\Gamma|$ do
   for each $i \in [1, |\Gamma|]$ do
      Let $\alpha_i$ be a CPL-valuation over $Y$ such that $\alpha_i(T_i) = \top$ if $\alpha_i(\Gamma_T)$ is PLTL-unsatisfiable then
         $\Gamma \leftarrow \Gamma \setminus \varphi_i$
      end if
   end for
end while

4.2 Using BDDs To Compute Minimal PLTL-Unsatisfiable Subsets

We now describe how we implemented a slight variant of this algorithm using BDDs and our BDD-based theorem prover for PLTL pltlbdd based on Marrero’s fixpoint method [6]. Since we used pltlbdd as a black box, we give a very high level description which omits the inner workings of pltlbdd. In particular, our description elides some complications to simplify the description of our pltlbdd works.

A BDD over a set $X = \{x_1, \ldots, x_n\}$ of propositional variables represents a function mapping each of the $2^n$ Boolean valuations on these variables to one of $\{\top, \bot\}$. If we view the valuations which the BDD maps to $\top$ as being “selected”, then a BDD represents a subset of the $2^n$ classical valuations over $X$. Thus a BDD is a function $f : 2^X \mapsto \{\top, \bot\}$ that selects some member from the powerset $2^X$ of $X$.

Given a set $\Delta$ of PLTL-formulæ as input pltlbdd first computes the set $sf(\Delta)$ of all subformulæ of $\Delta$. The cardinality $d$ of $sf(\Delta)$ is a polynomial in the number of symbols in $\Delta$. Using the finite model property of PLTL, it can be shown that to determine the PLTL-satisfiability of any subset of $\Delta$, we need to consider the interpretations that give a truth value to the elements of $sf(\Delta)$ only. That is, each subset of $sf(\Delta)$ can be thought of as a state that maps the members of this subset to true and we need to consider interpretations over only these $2^d$ states. The canonical way to construct an interpretation is to make a state $s' \subseteq sf(\Delta)$ a successor of a state $s \subseteq sf(\Delta)$ exactly when $\{\psi | X \psi \in s\} \subseteq s'$; that is, when every $X \psi$ in $s$ appears in $s'$ without the $X$.

Not all of these $2^d$ states actually appear in some PLTL-interpretation, so pltlbdd weeds out all the states which cannot possibly appear in any PLTL-interpretation built out of $sf(\Delta)$, and returns only the “good” ones as follows.

Given a set $\Delta$ of PLTL-formulæ as input, pltlbdd assigns a new propositional variable for each member of $sf(\Delta)$, so it creates $X = \{x_1, \ldots, x_d\}$. It then constructs all possible PLTL-interpretations over these variables and outputs a BDD $W$ which “selects” all CPL-valuations over $X$ that lead to a PLTL-interpretation built out of $sf(\Delta)$. Intuitively, each selected CPL-valuation corresponds to a state which appears in some PLTL-interpretation built out of $sf(\Delta)$.
More specifically, a formula $\psi \in sf(\Delta)$ is true in some PLTL-interpretation exactly when the variable $x_\psi$ for $\psi$ is true in any of these selected CPL-valuations over $X$. For each $\psi \in sf(\Delta)$, let $[\psi]$ be the set of all selected states from $W$ that make $\psi$ true in some PLTL-model.

Given some $S \subseteq \Delta$, we can construct the BDD $[S] = \wedge_{\psi \in S}[[\psi]]$ where $[[\psi]]$ is the BDD-representation of $\psi$ given by pltlbdd. If $W \wedge [S] = \emptyset$ then $S$ is PLTL-unsatisfiable, else $W \wedge [S]$ is a BDD representing all states over $sf(\Delta)$ which PLTL-satisfy $S$. For details see [6].

Given $\Gamma = \{\varphi_1, \ldots, \varphi_m\}$, we call pltlbdd with input $\Gamma$ and obtain the output BDD $W$ representing all the good states that can be built out of $sf(\Gamma)$. We use standard BDD operations to count the number of such CPL-valuations in terms of the tag variables $Y$ in $O(m)$ time [4], and to restrict the BDD $\exists X. [TT]$ to each valuation $\alpha$ of the tag variables, allowing us to determine whether each $\alpha(T\Gamma) = I_1 = I \setminus \{\varphi_i\}$ is PLTL-satisfiable. Iterating through $\Gamma$ to find a formula to remove also requires only linear time, giving us everything required to implement the algorithm described in Section 4.1 using only one call to pltlbdd. Note that the BDD $\exists X. [TT]$ must be rebuilt every time a formula is removed, however, the caching performed by the BDD library reduces the amount of repetitive computation required.

In order to extract the required information from the BDD prover pltlbdd, certain optimisations inside it had to be disabled. In particular, the prover pltlbdd was no longer permitted to turn $G \psi$ formulae into assumptions that were considered true at every state, as we needed to be able to query subsets of the input where those formulae were not included. As we shall see, this has major implications on performance.

5 Experimental Results

We built an implementation PLTL-MUP of this algorithm on top of the pltlbdd prover as described in Section 4.2. The core algorithm was built in OCaml, using the BuDDy library for BDD operations. We then compared PLTL-MUP to two similar provers, TRP++UC and procmine. Our experimental method is described in Section 5.1, before results are presented and discussed in Section 5.2.

5.1 Experimental Method

A set of 635 benchmark cases were chosen from several families, based off those used in [5]. In particular, the lift, genbuf and forobots are application-oriented benchmarks and $O_1$ formula and $O_2$ formula evaluate how well large
inputs can be handled. A number of randomly generated (pltl) benchmarks were also used. All of the test cases are PLTL-unsatisfiable.

We executed these benchmarks over a total of five prover configurations:

**PLTL-MUP:** This is our minimal PLTL-unsatisfiability checker, running on top of pltlbdd with certain optimisations turned off.

**pltlbdd:** This is the BDD prover running with all optimisations turned on, in order to determine the performance impact of disabling these optimisations.

**procmine:** This is the method described in [2], which uses tableaux with back-jumping to extract an PLTL-unsatisfiable subset, and then uses repeated calls to the theorem prover to reduce this to a minimal PLTL-unsatisfiable subset.

**TRP++UC:** This is the method described in [5], which uses temporal resolution to construct a proof of PLTL-unsatisfiability, and then traces back through the proof to find parts of the input which directly contribute to the PLTL-unsatisfiability. Note that this method does not guarantee minimality (unlike PLTL-MUP and procmine).

**Hybrid:** During early testing, it was found that the primary bottleneck in PLTL-MUP was the call to the theorem prover. This configuration is an attempt to reduce this bottleneck with a hybrid approach which runs TRP++UC over the input to obtain a reduced (but not necessarily minimal) subset, before passing this subset to PLTL-MUP for minimisation.

All tests were performed on a Intel Core i5 3570K 3.4GHz processor. Execution of each test case was limited to 600 seconds of runtime and 6GB of memory usage. If two test cases from a sub-family failed by running out of time or memory, all remaining test cases from that sub-family were skipped. Table 1 shows the total number of test cases in each family and the number of test cases from each family successfully executed by each prover.

Some slight variation can be seen between executions within these results. For example, TRP++UC solved 17 genbuf examples while the hybrid approach solved 18, even though Hybrid calls TRP++UC internally. This is caused by a test case which took very close to the 600 second time limit successfully finishing in one instance, but failing in the other.

### 5.2 Discussion of results

These results show that TRP++UC gives better performance on these benchmarks than pltlbdd running with all optimisations turned on. When we turn off these optimisations in order to allow the minimisation procedure to run, performance of the prover decreases further. In almost all test cases which PLTL-MUP completed within the time limit, running time was dominated by time spent proving PLTL-unsatisfiability, while minimisation generally completed in less than a second. Finally, procmine performed the worst of the three solvers on these benchmarks.
Table 1. Number of successful test cases for each prover over each benchmark family

<table>
<thead>
<tr>
<th>Family</th>
<th>Name</th>
<th>Total Size</th>
<th>PLTL-MUP</th>
<th>pltlbdd</th>
<th>Procmine</th>
<th>TRP++UC</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>forobots</td>
<td>25</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>25</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>genbuf</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>lift</td>
<td>102</td>
<td>11</td>
<td>40</td>
<td>36</td>
<td>43</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>O1 formula</td>
<td>26</td>
<td>23</td>
<td>26</td>
<td>26</td>
<td>21</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>O2 formula</td>
<td>15</td>
<td>13</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>pltl</td>
<td>10</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>pltl-12x</td>
<td>175</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>175</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>pltl-12y</td>
<td>67</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pltl-5x</td>
<td>109</td>
<td>64</td>
<td>109</td>
<td>4</td>
<td>109</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>pltl-5y</td>
<td>46</td>
<td>46</td>
<td>46</td>
<td>0</td>
<td>46</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>635</td>
<td>160</td>
<td>257</td>
<td>82</td>
<td>515</td>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

However, note that TRP++UC solves a far simpler problem than PLTL-MUP, as TRP++UC does not guarantee that its results will be minimal. Due to this difference in approaches, a fair comparison cannot be drawn between the two.

The hybrid approach provides a good middle ground between the performance of TRP++UC and the guaranteed minimality of PLTL-MUP. Performing this test case has also allowed us to analyse how well TRP++UC reduces formulae. Table 2 groups each test case according to the four possible results from the hybrid theorem prover:

1. Successfully minimised by TRP++UC and then checked by PLTL-MUP
2. Reduced but not minimised by TRP++UC and then minimised by PLTL-MUP
3. Failed (time or memory) within TRP++UC
4. Passed by TRP++UC, but failed by PLTL-MUP (so we are unable to determine whether the result from TRP++UC is minimal)

It can be seen that 136 of the 236 the test cases successfully completed by the hybrid approach were not minimised by TRP++UC but were minimised by the subsequent call to PLTL-MUP instead.

Finally, Figure 1 compares the output size of TRP++UC to that of the hybrid prover for each test case. The further below the $y = x$ line a point lies, the more that test case was reduced by the hybrid. It can be seen that the hybrid approach achieved a significant reduction for many of the test cases.

6 Related Work

While several approaches have been taken to extract minimal PLTL-unsatisfiable subsets of classical propositional logic [4, 7], relatively little work has been done within this domain for linear temporal logic. Two such approaches have already been introduced.
Table 2. Breakdown of results from Hybrid into each of the four possibilities

<table>
<thead>
<tr>
<th>Family Total size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>forobots</td>
<td>25</td>
<td>10</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>genbuf</td>
<td>60</td>
<td>7</td>
<td>11</td>
<td>42</td>
</tr>
<tr>
<td>lift</td>
<td>102</td>
<td>39</td>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>O1_formula</td>
<td>26</td>
<td>21</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>O2_formula</td>
<td>15</td>
<td>8</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>phltl</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>pltl-12x</td>
<td>175</td>
<td>4</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>pltl-12y</td>
<td>67</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pltl-5x</td>
<td>109</td>
<td>23</td>
<td>86</td>
<td>0</td>
</tr>
<tr>
<td>pltl-5y</td>
<td>46</td>
<td>5</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>635</td>
<td>120</td>
<td>136</td>
<td>119</td>
</tr>
</tbody>
</table>

Fig. 1. Reduction in TRP++UC output size performed by hybrid approach. Each point represents an individual test case.
In [2], Awad et al. describe the method used within **procmine**, using tableaux and back-jumping and then performing multiple calls to the theorem prover to reduce the result to a minimal PLTL-unsatisfiable subset. They focus heavily on the application of their method to business process modelling, and thus are able to optimise by considering domain knowledge and compliance rules individually, only considering their conjunction when necessary.

The alternative approach taken by **TRP++UC** is described in [5]. Due to the use of Separated Normal Form, Schuppan’s notion of a PLTL-unsatisfiable subset is somewhat different to ours. As noted in Section 5.1, Schuppan’s approach does not guarantee minimality, making these results less useful for applications such as business process modelling than those given by **PLTL-MUP** and **procmine**. Schuppan extends his approach by annotating results with a set of time states at which each formula in the PLTL-unsatisfiable subset is used [8]. This additional information may assist with debugging, however, without a guarantee of minimality it may also add additional noise to the process of debugging.

### 7 Future Work

A number of avenues for future work within this topic are available. Firstly, as discussed in section 5, it is possible that significant performance improvements could be made by attempting an initial coarse reduction in the size of the input before sending it through the full theorem prover and minimisation procedure. One possible way to achieve this would be to partition the formula into subsets operating on distinct sets of propositional atoms. Each subset can then be tested to see if it is PLTL-unsatisfiable. If neither subset is PLTL-unsatisfiable, they can be combined and tested as one large set. As soon as an PLTL-unsatisfiable subset of the input is found, it can be sent through the remainder of the minimisation process. This procedure can be recursively applied to each subset in order to allow small PLTL-unsatisfiable subsets to be easily located. The amount of work that is performed by repeatedly calling the theorem prover can be reduced, as **pltlbdd** allows the fixpoint calculation to be started with an existing BDD. In this case, we can use the fixpoint of a subset of $\Gamma$ to start the fixpoint calculation for $\Gamma$, and thus preventing this work from being recomputed.

Performance could also be improved by employing early quantification techniques. Early quantification is used within symbolic satisfiability solvers for classical propositional logic, which also involves existentially quantifying a large conjunction of BDDs [9]. In order to make this operation more efficient, quantification is ‘pushed down’ in to the conjuncts, so that the size of the intermediate BDDs is reduced. The same technique could be employed when calculating $\exists X.\llbracket T \Gamma \rrbracket$ in our procedure. However, this is unlikely to give significant improvements until the main bottleneck in the theorem prover can be reduced.
8 Conclusion

We have presented a method which can be used to find minimal PLTL-unsatisfiable subsets of input PLTL formulae. This method uses a BDD-based theorem prover to deduce PLTL-satisfiability information of the input set, which allows the PLTL-satisfiability of subsets of this input to be queried efficiently. We then augment the input set by tagging each formula so that we can simulate turning off a single formula at a time. By combining these two inputs with standard BDD procedures, we are able to determine whether the input is already PLTL-unsatisfiable, and if not, find formulae which can be safely removed.

We have implemented and experimentally evaluated this procedure against other similar approaches, finding that while our approach performs well for some inputs, it is held back by the need to disable optimisations performed by the theorem prover. However, a hybrid approach combining our work with TRP++UC provided an effective middle ground in terms of performance while still guaranteeing minimality. Future work to address this issue has been suggested, including performing preprocessing to reduce the amount of work done by the theorem prover.
Bibliography