Deductive Temporal Reasoning with Constraints

Clare Dixon*, Boris Konev*, Michael Fisher*, Sherly Nietiadi*

Department of Computer Science, University of Liverpool, Liverpool, U.K.

Abstract

When modelling realistic systems, physical constraints on the resources available are often required. For example, we might say that at most N processes can access a particular resource at any moment, exactly M participants are needed for an agreement, or an agent can be in exactly one *mode* at any moment. Such situations are concisely modelled where literals are constrained such that at most N, or exactly M, can hold at any moment in time. In this paper we consider a logic which is a combination of standard propositional linear time temporal logic with cardinality constraints restricting the numbers of literals that can be satisfied at any moment in time. We present the logic and and show how to represent a number of case studies using this logic. We propose a tableau-like algorithm for checking the satisfiability of formulae in this logic, provide details of a prototype implementation and present experimental results using the prover.

Keywords: Temporal logic, Constraints, Theorem Proving, Tableau

1. Introduction

Temporal logic allows the concise specification of *temporal* order. However, if we need to represent cardinality restrictions we have to introduce a large number of formulae to the specification making it hard to read and understand, and difficult for provers to deal with. In addition, while temporal logic has turned out to be a very useful notation across a number of areas, particularly the specification of concurrent and distributed systems [45, 44, 27], the complexity of many temporal logics is often considered to be too high for practical verification (see for example [11, 8]). Consequently, simple modal logics, finite state automata, or even Boolean satisfiability, are typically used in the verification of such systems. This is because the decision problem for propositional linear temporal logic (PTL) is PSPACE-complete [30] whereas techniques in many of the above areas are much simpler.

So, the question we are concerned with in this work is the following: can we represent and reason about such cardinality restrictions (here we use the term constraints) in a compact and transparent way, retaining the useful descriptive powers of temporal logics, while making the reasoning more efficient in practice? Here we propose and utilise a succinct way of specifying cardinality constraints. We show that if examples are in (or close to) a particular normal form, using this representation of constraints simplifies the reasoning. Additionally, we experiment with an implemented prototype prover for this logic.

To specify the constraints we allow statements stating that *up to k* literals, or *exactly k* literals from some subset of literals, are true at any moment in time. Note that this approach involves reasoning *in the presence* of constraints rather than reasoning *about* them. Thus, the resulting logic represents a combination of standard temporal logic with (fixed) constraints that restrict the numbers of literals that can be satisfied at any moment in time. This new approach is particularly useful for (for example):

*Corresponding author

Email addresses: `cldixon@liverpool.ac.uk` (Clare Dixon), `konev@liverpool.ac.uk` (Boris Konev), `mfisher@liverpool.ac.uk` (Michael Fisher), `Sherly.Nietiadi@liverpool.ac.uk` (Sherly Nietiadi)

- ensuring that a fixed bound is kept on the number of propositions satisfied at any moment to prevent overload;
- in finite collections of communicating automata, ensuring that no more than k automata are in a particular state;
- modelling restrictions on resources, for example at most k vehicles are available or there are at most k seats available;
- modelling the necessity to elect exactly k from n participants.

Motivating Example. Consider a fixed number, n, of robots that can each *work*, *rest* or *recharge*. We assume that there are only $k < n$ recharging points and only $j < n$ workstations. Let:

- $work_i$ represent the fact that robot i is working;
- $rest_i$ represent the fact that robot i is resting; and
- $recharge_i$ represent the fact that robot i is recharging.

Now, we typically want to specify that exactly j of the n robots are working at any one time. In the syntax given later, such a logic might be defined as TLC($\mathcal{W}^{=j}, \mathcal{R}^{\leqslant k}$), where

$$\begin{array}{rcl}\mathcal{W}^{=j} & = & \{work_1, \ldots, work_n\}^{=j} \\ \mathcal{R}^{\leqslant k} & = & \{recharge_1, \ldots, recharge_n\}^{\leqslant k}\end{array}$$

This represents the logic with the constraints that exactly j robots must work at any moment and at most k can recharge at any moment.

This paper extends preliminary material from our earlier paper [19]. The contributions of the paper are: to define and analyse a logic which combines both temporal logic and constraints; to show how a number of case studies can be elegantly modelled using this logic; to provide a tableau-like satisfiability algorithm for formulae in this logic giving proofs of correctness; to provide algorithms for a prototype implementation of this; and give experimental results comparing the implementation with other temporal provers.

The paper is organised as follows. Section 2 gives the syntax and semantics of the constrained temporal logic, together with a normal form for this logic. In Section 3 we provide a number of case studies and show how they are specified in this logic. In Section 4 we provide an algorithm for checking satisfiability of this logic and consider its complexity. In Section 5 we give details of an implementation of the satisfiability checker for this logic and experimental details comparing this implementation to other tableau reasoners for propositional linear time temporal logic. Finally, in Section 6, we provide concluding remarks and discuss both related and future work.

2. A Constrained Temporal Logic

Temporal Logic with Cardinality Constraints (TLC) [19] is PTL with some additional constraints, which restrict the numbers of literals that can be satisfied at any moment in time. TLC is parameterised by (not necessarily disjoint) sets $C^{\propto m}$ where $\propto \in \{=, \leqslant\}$ and $m \in \mathbb{N}$. The formulae of TLC($C_1^{\propto_1 m_1}, C_2^{\propto_2 m_2}, \cdots$) are constructed under the restriction that, depending on $\propto_i$, exactly m_i literals from every set C_i are true in every state ($\propto_i$ is $=$) or less than or equal to m_i literals from every set C_i are true in every state ($\propto_i$ is $\leqslant$). For example, consider TLC($C_1^{=2}, C_2^{\leqslant 1}$), where $C_1^{=2} = \{p, q, r\}^{=2}$ and $C_2^{\leqslant 1} = \{x, y, z\}^{\leqslant 1}$. Then, at any moment of time, exactly two of p, q, or r are true, and less than or equal to one of x, y, or z is true. In addition to these constrained sets, there exists a set of propositions, $\mathcal{A}$, which are standard, unconstrained propositions. Note that, the 'less than' constraint $C^{<m}$ can be expressed as $C^{\leqslant m-1}$ and the 'more than or equal to' constraint $C^{\geqslant m}$ can be expressed as $\bar{C}^{\leqslant n-m}$, where n is the number of literals in C and by definition $\bar{C} = \{\bar{x} \mid x \in C\}$, $\bar{p} = \neg p$ and $\overline{\neg p} = p$. Moreover by using both $C^{\leqslant m}$ and $C^{\geqslant m}$ (encoded

as $\bar{C}^{\leqslant n-m}$) we could also obtain $C^{=m}$. We choose not to do this however, as we aim for a clear and intuitive way of expressing constraints and this appears to obscure the meaning. Further, the constraint $C^{=1}$ seems to be common in applications (see Section 3) which gives additional weight for the $=1$ construct to be primitive.

We note that we can express the information in our constrained sets as temporal formulae. For example given the constraint $C_1^{=2} = \{p, q, r\}^{=2}$ above, this can be represented by the following temporal formula.

$$\Box((p \vee q) \wedge (p \vee r) \wedge (q \vee r)) \wedge \Box(\neg p \vee \neg q \vee \neg r).$$

2.1. TLC Syntax

A constraint $C_i^{\propto_i m_i}$ is a tuple $(C_i, \propto_i, m_i)$, where C_i is a set of literals with a cardinality restriction $\propto_i m_i$, such that $\propto_i \in \{=, \leqslant\}$ and $m_i \in \mathbb{N}$. For TLC, the future-time temporal connectives we use include '$\bigcirc$' (in the next moment) and '$\mathcal{U}$' (until). Formally, TLC$(C_1^{\propto_1 m_1}, \cdots, C_n^{\propto_n m_n})$ formulae are constructed from the following elements:

- a set, $\mathsf{Props} = \{p \mid p \in C_i^{\propto_i m_i}\} \cup \{p \mid \neg p \in C_i^{\propto_i m_i}\} \cup \mathcal{A}$ of propositional symbols (where $1 \leqslant i \leqslant n$ and $\mathcal{A}$ are termed 'unconstrained' propositions);
- propositional connectives, **true**, $\neg$, $\wedge$; and
- temporal connectives, $\bigcirc$, and $\mathcal{U}$.

We also write TLC$(\mathcal{C})$, where $\mathcal{C} = \{C_1^{\propto_1 m_1}, \cdots, C_n^{\propto_n m_n}\}$.

The set of well-formed formulae (WFF) of TLC, is defined as the smallest set satisfying the following:

- any elements of Props and **true** are in WFF;
- if φ and ψ are in WFF, then so are $\neg\varphi, \varphi \wedge \psi, \bigcirc\varphi, \varphi\, \mathcal{U}\, \psi$.

A *literal* is defined as either a proposition symbol or the negation of a proposition symbol. We (ambiguously) assume that the negation of $\neg p$ is p.

2.2. TLC Semantics

First we define the satisfiability of a constraint. The notation $\mathcal{L} \models_{PL} \varphi$ denotes the truth of propositional logic formula φ with respect to a set of propositions $\mathcal{L}$. $\mathcal{L} \models_{PL} p$ iff $p \in \mathcal{L}$ where $p \in \mathsf{Props}$ and the semantics of the operators $\neg$, $\wedge$ is as usual. Let $\mathcal{L}$ be a set of propositions, $C^{\propto m}$ a constraint and

$$Eval(\mathcal{L}, C^{\propto m}) = \{p \mid p \in \mathcal{L} \text{ and } p \in C\} \cup \{\neg p \mid p \notin \mathcal{L} \text{ and } \neg p \in C\}$$

then

$$\begin{array}{lll} \mathcal{L} \models_{PL} C^{=m} & \text{iff} & |Eval(\mathcal{L}, C^{=m})| = m, \\ \mathcal{L} \models_{PL} C^{\leqslant m} & \text{iff} & |Eval(\mathcal{L}, C^{\leqslant m})| \leqslant m. \end{array}$$

Note that the operator $\models_{PL}$ is only defined for formulae from propositional logic (not from temporal logic). A set $\mathcal{C}$ of constraints is *satisfiable* ($\models_{PL} \mathcal{C}$) if, and only if, there is a set of propositions $\mathcal{L}$, such that, for each $C_i^{\propto_i m_i} \in \mathcal{C}$ ($i \in \mathbb{N}$), $\mathcal{L} \models_{PL} C_i^{\propto_i m_i}$.

A model for TLC$(\mathcal{C})$ formulae can be characterised as a *sequence of states*, σ, of the form $\sigma = s_0, s_1, s_2, s_3, \ldots$, where each state s_i is a set of propositional symbols representing those propositions, which are satisfied at the i^{th} moment in time. Every s_i should satisfy the set of constraints, $\mathcal{C}$, i.e., for all s_i we have $s_i \models_{PL} \mathcal{C}$ (where s_i is a set of propositions).

The notation $(\sigma, i) \models \varphi$ denotes the truth of formula φ in the model σ at the state of index $i \in \mathbb{N}$ and is defined as follows.

$$\begin{array}{lll}
(\sigma,i) \models & \mathbf{true} & \\
(\sigma,i) \models & p & \text{iff } p \in s_i \text{ where } p \in \mathsf{Props} \\
(\sigma,i) \models & \neg\varphi & \text{iff it is not the case that } (\sigma,i) \models \varphi \\
(\sigma,i) \models & \varphi \wedge \psi & \text{iff } (\sigma,i) \models \varphi \text{ and } (\sigma,i) \models \psi \\
(\sigma,i) \models & \bigcirc\varphi & \text{iff } (\sigma,i+1) \models \varphi \\
(\sigma,i) \models & \varphi\,\mathcal{U}\,\psi & \text{iff } \exists k \in \mathbb{N}.\ k \geqslant i \text{ and } (\sigma,k) \models \psi \text{ and} \\
 & & \forall j \in \mathbb{N}, \text{if } i \leqslant j < k \text{ then } (\sigma,j) \models \varphi
\end{array}$$

Note we can obtain **false** and the other Boolean operators via the usual equivalences and we define '$\square$' (always in the future), '$\diamondsuit$' (sometime in the future) and '$\mathcal{W}$' (unless or weak until) operators as follows.

$$\begin{array}{rcl}
\diamondsuit\varphi & \equiv & \mathbf{true}\,\mathcal{U}\,\varphi \\
\square\varphi & \equiv & \neg\diamondsuit\neg\varphi \\
\varphi\,\mathcal{W}\,\psi & \equiv & (\varphi\,\mathcal{U}\,\psi) \vee (\square\varphi)
\end{array}$$

For any formula φ, model σ, and state index $i \in \mathbb{N}$, either $(\sigma, i) \models \varphi$ holds or $(\sigma, i) \models \varphi$ does not hold, denoted by $(\sigma, i) \not\models \varphi$. If there is some σ such that $(\sigma, 0) \models \varphi$, then φ is said to be *satisfiable*. If $(\sigma, 0) \models \varphi$ for all models, σ, then φ is said to be *valid* and is written $\models \varphi$. A set $\mathcal{N}$ of formulae is *satisfiable* in the model σ at the state of index $i \in \mathbb{N}$ if, and only if, for all $\varphi \in \mathcal{N}, (\sigma, i) \models \varphi$.

A formula of the form $\diamondsuit\varphi$ or $\psi\,\mathcal{U}\,\varphi$ is called an *eventuality*. A formula of the form $\bigcirc\varphi$ is called a *next-time formula*.

2.3. Normal Form

It is often convenient to operate on formulae in a normal form. Separated Normal Form (SNF) was first introduced for PTL in [26] (see also [28]); the normal form for TLC extends one from [28] with ideas from [16]. To assist in the definition of the normal form we introduce a further (nullary) connective '**start**' that holds only at the beginning of time, i.e.,

$$(\sigma, i) \models \mathbf{start} \qquad \text{iff} \qquad i = 0.$$

This allows the general form of the (temporal clauses of the) normal form to be implications.

In the following, small Latin letters, k_i, l_j, m represent literals in the language Props where $i, j \geqslant 0$. Note, on the left hand side, if $i = 0$ the empty conjunction represents **true** whereas, on the right hand side, if $j = 0$ the empty disjunction represents **false**. A normal form for TLC is of the form

$$\square \bigwedge_h X_h$$

where $h \geqslant 1$ and each X_h is an *initial*, *step*, or *sometime* clause (respectively) as follows:

$$\begin{array}{rcll}
\mathbf{start} & \Rightarrow & \bigvee_j l_j & \textit{(initial)} \\
\bigwedge_i k_i & \Rightarrow & \bigcirc \bigvee_j l_j & \textit{(step)} \\
\mathbf{true} & \Rightarrow & \diamondsuit m & \textit{(sometime)}
\end{array}$$

Sometime clauses defined above are also known as *unconditional* sometime clauses. SNF defined in [28] allows for *conditional* sometime clauses—expressions of the form

$$\bigwedge_i k_i \Rightarrow \diamondsuit m\,.$$

However, it was shown in [16] that any PTL formula can be translated into SNF with only unconditional sometime clauses. We can rewrite a conditional sometime clause of the above form into an unconditional one, of the form we require here, using a new propositional variable w_m. Informally, the new proposition w_m denotes *waiting for m.* The resulting clauses replacing the conditional eventuality are as follows.

$$\begin{array}{rcl} \mathbf{true} & \Rightarrow & \Diamond\neg w_m \\ \mathbf{start} & \Rightarrow & (\bigvee_i \neg k_i \vee m \vee w_m) \\ \mathbf{true} & \Rightarrow & \bigcirc(\bigvee_i \neg k_i \vee m \vee w_m) \\ w_m & \Rightarrow & \bigcirc(m \vee w_m) \end{array}$$

Theorem 1 of [16] shows a set of clauses with the conditional sometime clauses replaced by the unconditional sometime clause and three additional clauses above preserves satisfiability and Lemma 1 of [16] shows this involves a linear increase in the size of the problem relating to the number of eventualities occurring.

Since the constraints do not affect the normal form, we obtain the following theorem as a corollary of [16].

Theorem 1. *Any TLC formula can be transformed into an equi-satisfiable TLC formula in SNF with at most a linear increase in the size of the formula.*

When specifying the behaviour of systems, it is sometime convenient to consider 'traditional' clauses of the form

$$\bigvee_j l_j \qquad \textit{(global)}$$

Every global clause can, if necessary, be represented as a combination of an initial and a step clause:

$$\mathbf{start} \Rightarrow \bigvee_j l_j \qquad \text{and} \qquad \mathbf{true} \Rightarrow \bigcirc \bigvee_j l_j$$

We will use global clauses in Section 3.

Transformation into the normal form may introduce new (unconstrained) propositions; however, as we will see in Section 3, many temporal formulae stemming from realistic specifications are already in the normal form, or very close to the normal form and require few extra variables for the translation (for other examples, also see [24]).

2.4. Encoding of Constraints

In the case of PTL the addition of constraints does not extend the logic, i.e. we can rewrite any constraint $C^{\propto k}$ into the syntax of PTL. However, our representation of constraints is succinct and allows the prover to make use of this information in a global way. To compare with other provers for PTL we must add formulae that represent these constraints. We use a direct encoding of constraints, which does not introduce extra propositions. For S being a set of literals and $k \in \mathbb{N}$ we define

$$\textit{pos}(S,k) = \bigwedge_{\substack{U \subseteq S \\ |U| = |S|+1-k}} \bigvee_{l_i \in U} l_i$$

and

$$\textit{neg}(S,k) = \bigwedge_{\substack{U \subseteq S \\ |U| = k+1}} \bigvee_{l_i \in U} \neg l_i$$

For example, $\textit{pos}(\{p,q,r\},2) = (p \vee q) \wedge (p \vee r) \wedge (q \vee r)$ and $\textit{neg}(\{p,q,r\},2) = \neg p \vee \neg q \vee \neg r$. Intuitively, $\textit{neg}(S,k)$ denotes that for a set of literals S, to make at most k true, for every subset of size $k+1$ at least one of these must be false. Similarly, $\textit{pos}(S,k)$ denotes that for a set of literals S, to make at most $|S| - k$ false, for every subset of size $|S| + 1 - k$ at least one of these must be true.

Lemma 2 ([52]). *Given a set of propositions $\mathcal{L}$, $\mathcal{L} \models_{PL} C^{=k}$ if, and only if, $\mathcal{L} \models_{PL} pos(C,k) \wedge neg(C,k)$ and $\mathcal{L} \models_{PL} C^{\leqslant k}$ if, and only if, $\mathcal{L} \models_{PL} neg(C,k)$.*

To generate PTL formulae equivalent to the constraints, then:

- for a constraint of the form $C^{=k}$ we construct $\square(pos(C,k)) \wedge \square(neg(C,k))$;
- for a constraint of the form $C^{\leqslant k}$ we construct $\square(neg(C,k))$.

For a constraint of the form $C^{\leqslant k}$, such a direct encoding contains $\binom{n}{k+1}$ clauses, which for $k = \lceil n/2 \rceil - 1$ reaches $O\left(2^n/\sqrt{n/2}\right)$ clauses [52].

We note that other translations are possible. For example we can use the fact that an n-bit counter can represent 2^n different values. If we required *exactly one* constraints ($C^{=1}$) we could represent a constraint of the form $\{p_1, p_2, p_3, p_4\}^{=1}$ using just two propositional variables t'_1, t'_2 by adding the following formulae.

$$\begin{array}{lcl} \square(p_1 & \Leftrightarrow & (t'_1 \wedge t'_2)) \\ \square(p_2 & \Leftrightarrow & (t'_1 \wedge \neg t'_2)) \\ \square(p_3 & \Leftrightarrow & (\neg t'_1 \wedge t'_2)) \\ \square(p_4 & \Leftrightarrow & (\neg t'_1 \wedge \neg t'_2)) \end{array}$$

Other translations based on different counter encodings can be found in the literature, see, for example, [2, 5, 52]. The size of such encodings is typically linear in n and k.

3. Case Studies

Next we consider several case studies and show how they can be specified in TLC. For ease of presentation some of the formulae presented are not strictly in SNF but can be transformed into SNF using simple equivalences. The case studies here are necessarily simplified and are included purely to exemplify the approach; clearly much larger examples can be constructed along similar lines.

3.1. Five-A-Side Football

Consider a team of football playing agents. This could be either a team involving human players or a RoboCup team. The rules say that at most 5 players (i.e. a "five-a-side" game) in the team can be on the field of play at any time. However, other players can be off the field of play, either resting or injured.

Let us begin to model such a scenario using temporal logic. We will describe the current activity of a particular player, i, using the propositions $playing_i$, $resting_i$, and $injured_i$. Thus:

- $playing_i$ represents the fact that player i is actually playing;
- $resting_i$ represents the fact that player i is resting; and
- $injured_i$ represents the fact that player i is injured.

Let us assume, for simplicity that our team has 6 players; to begin with, one is just resting:

$$\textbf{start} \Rightarrow playing_1 \wedge playing_2 \wedge playing_3 \wedge playing_4 \wedge playing_5 \wedge resting_6$$

We can describe the possible dynamic behaviours, for each i, as follows.

- when playing, a player can either continue, stop to rest, or become injured:

$$\square(playing_i \Rightarrow \bigcirc(playing_i \vee resting_i \vee injured_i))$$

- when resting, a player can either continue resting, or return to playing:

$$\square(resting_i \Rightarrow \bigcirc(playing_i \vee resting_i))$$

- once injured, a player remains injured:

$$\square(\mathit{injured}_i \;\Rightarrow\; \bigcirc \mathit{injured}_i)$$

We might also want to add some formulae describing details of the particular scenario, for example we might want to state that any player resting will eventually get to play:

$$\square(\mathit{resting}_i \;\Rightarrow\; \lozenge \mathit{playing}_i)$$

Whilst the above formula is not part of the TLC normal form syntax it can be rewritten into the following equi-satisfiable formulae (see Theorem 1).

$$\begin{array}{rcl}
\square(\mathbf{true} & \Rightarrow & \lozenge\neg w_{\mathit{playing}_i}) \\
\square(\mathbf{start} & \Rightarrow & \neg \mathit{resting}_i \vee \mathit{playing}_i \;\vee w_{\mathit{playing}_i}) \\
\square(\mathbf{true} & \Rightarrow & \bigcirc(\neg \mathit{resting}_i \vee \mathit{playing}_i \;\vee w_{\mathit{playing}_i})) \\
\square(w_{\mathit{playing}_i} & \Rightarrow & \bigcirc(\mathit{playing}_i \;\vee w_{\mathit{playing}_i}))
\end{array}$$

Of course there are a number of *structural* constraints that we must also describe for this scenario concerning the relationships between these propositions. For example, we must specify that there are at most 5 players playing at any moment. We do this by describing the constraint:

$$\mathcal{P}^{\leqslant 5} \;=\; \{\; \mathit{playing}_1,\, \mathit{playing}_2,\, \mathit{playing}_3,\, \mathit{playing}_4,\, \mathit{playing}_5,\, \mathit{playing}_6 \;\}^{\leqslant 5}$$

Another obvious constraint is that each player can only be playing, resting or injured. So, for each player, i, we would describe this constraint as

$$\mathcal{M}_i{}^{=1} \;=\; \{\; \mathit{playing}_i,\, \mathit{resting}_i,\, \mathit{injured}_i \;\}^{=1}$$

Thus, in our notation, we can define a logic that has these structural constraints "built-in" as follows:

$$\mathrm{TLC}(\mathcal{P}^{\leqslant 5},\, \mathcal{M}_1^{=1},\, \mathcal{M}_2^{=1},\, \mathcal{M}_3^{=1},\, \mathcal{M}_4^{=1},\, \mathcal{M}_5^{=1},\, \mathcal{M}_6^{=1})$$

This provides us with a logic in which the above structural constraints are implicitly enforced at every moment in time and avoids the need to explicitly encode these as temporal formulae.

3.1.1. Viable Teams

Once we have this scenario we can embellish it in many ways. One is to consider not just the rules of the game (i.e. that at most 5 players can be playing at any time) but the viability of the team itself. For example, what if only 2 players are actually playing? Or even 1? Very likely the team will lose quickly. So, we might add another constraint to ensure that the team is always viable, for example:

$$\mathcal{V}^{\leqslant 3} \;=\; \left\{ \begin{array}{l} \mathit{resting}_1,\, \mathit{resting}_2,\, \mathit{resting}_3,\, \mathit{resting}_4,\, \mathit{resting}_5,\, \mathit{resting}_6, \\ \mathit{injured}_1,\, \mathit{injured}_2,\, \mathit{injured}_3,\, \mathit{injured}_4,\, \mathit{injured}_5,\, \mathit{injured}_6 \end{array} \right\}^{\leqslant 3}$$

This ensures that at most 3 players are resting or injured, and so at least 3 players are actually playing. Adding $\mathcal{V}^{\leqslant 3}$ to our TLC definition ensures that only viable teams are described.

3.1.2. Goalkeeper

Another embellishment might be to add a 'goalkeeper'. This is a player who is designated to defend the goal. Let us add another set of propositions, $\mathit{goalkeeper}_i$, which is true if the player i is the goalkeeper. Clearly, the goalkeeper must be playing:

$$\square(\mathit{goalkeeper}_i \;\Rightarrow\; \mathit{playing}_i)$$

But the role of being a goalkeeper can move between players. So, if a player is not a goalkeeper now they will eventually take on this role:

$$\square(\neg goalkeeper_i \Rightarrow \lozenge goalkeeper_i)$$

The above needs to be rewritten into SNF similarly to that previously described. Finally, and most obviously, we need structural constraints concerning the goalkeeper proposition. In particular, at *most* one player can be the goalkeeper at any moment. (Note that we might not have *any* goalkeeper!) So, we add the constraint set $\mathcal{G}^{\leqslant 1}$:

$$\{\, goalkeeper_1,\, goalkeeper_2,\, goalkeeper_3,\, goalkeeper_4,\, goalkeeper_5,\, goalkeeper_6\}^{\leqslant 1}$$

3.1.3. Properties

There are many potential properties to prove. For example, we might show that if all the players eventually become injured then the team is unviable. Thus, if we add

$$\lozenge(injured_1 \wedge injured_2 \wedge injured_3 \wedge injured_4 \wedge injured_5 \wedge injured_6)$$

then the whole set of formulae should be unsatisfiable.

Another property we might show is that, as long as there are *no* injuries, then eventually every player will take a turn as the goalkeeper. Thus the specification so far, together with $\square \bigwedge_i \neg injured_i$, should imply

$$\begin{gathered}\lozenge goalkeeper_1 \wedge \lozenge goalkeeper_2 \wedge \lozenge goalkeeper_3 \wedge \\ \lozenge goalkeeper_4 \wedge \lozenge goalkeeper_5 \wedge \lozenge goalkeeper_6\end{gathered}$$

3.2. Cache Coherence Protocol

We next consider using our logic to specify a simple cache coherence protocol with a finite number of processes. Each processor has its own private cache memory which holds local copies of the main memory blocks. Cache coherence protocols aim to ensure the consistency of the cache for different processors. Abstracting away from low level details we can describe such protocols as a family of identical finite state systems together with a primitive form of communication.

We describe the MSI protocol (for more details see for example [34]) where each process can be in one of the three states invalid (i), shared (s), or modified (m). At any moment one process is active and, depending on its state, may carry out a read r, a write w or a local transition t. The action is then broadcast to all the other processes, which carry out a reaction to what has happened (denoted $\bar{r}$, $\bar{w}$ or $\bar{t}$ respectively). This is shown in the transition system in Figure 1.

$$\begin{array}{lll}
\tau(i,w)=m & \tau(s,w)=m & \tau(m,\bar{w})=i \\
\tau(i,\bar{w})=i & \tau(s,\bar{w})=i & \tau(m,\bar{r})=s \\
\tau(i,r)=s & \tau(s,\bar{r})=s & \tau(m,t)=m \\
\tau(i,\bar{r})=i & \tau(s,t)=s & \tau(m,\bar{t})=m \\
\tau(i,\bar{t})=i & \tau(s,\bar{t})=s &
\end{array}$$

Initially all processes are in the state i and we should show that

- it is never possible that one process is in state m while another is in state s (non co-occurrence of states s and m);
- it is always the case that at most one process can be in state m.

We can represent this for a finite number of processes where

a_j means that process j is active;

m_j means that process j is in the state m;

s_j means that process j is in the state s;

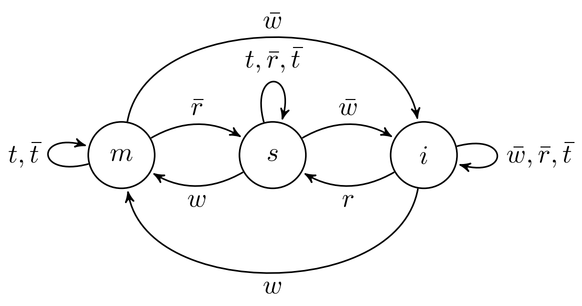

Figure 1: Finite state machine for the MSI protocol.

i_j means that process j is in the state i.

Rather than representing w, r and t for each process we will allow one for the system where if that process is not active it means that the process is reacting, i.e. carrying out a $\bar{w}$, $\bar{r}$, or $\bar{t}$ transition. There are a number of constrained sets. Given n processes, for each process j we have

$$\mathcal{S}_j^{=1} = \{m_j, s_j, i_j\}^{=1}$$

To make exactly one process active at any moment we have

$$\mathcal{P}^{=1} = \{a_1, a_2, \ldots a_n\}^{=1}$$

and to ensure only one of w, r and t holds we have

$$\mathcal{T}^{=1} = \{w, r, t\}^{=1}.$$

Thus for n processes we have TLC$(\mathcal{S}_1^{=1}, \ldots, \mathcal{S}_n^{=1}, \mathcal{P}^{=1}, \mathcal{T}^{=1})$. When a processor is active the transitions can be formalised as follows.

$$\begin{array}{rcl}
\Box((i_i \wedge a_i \wedge w) & \Rightarrow & \bigcirc m_i) \\
\Box((i_i \wedge a_i \wedge r) & \Rightarrow & \bigcirc s_i) \\
\Box((s_i \wedge a_i \wedge w) & \Rightarrow & \bigcirc m_i) \\
\Box((s_i \wedge a_i \wedge t) & \Rightarrow & \bigcirc s_i) \\
\Box((m_i \wedge a_i \wedge t) & \Rightarrow & \bigcirc m_i)
\end{array}$$

When the processor is *not* active the transitions can be formalised as follows.

$$\begin{array}{rcl}
\Box((i_i \wedge \neg a_i \wedge w) & \Rightarrow & \bigcirc i_i) \\
\Box((i_i \wedge \neg a_i \wedge r) & \Rightarrow & \bigcirc i_i) \\
\Box((i_i \wedge \neg a_i \wedge t) & \Rightarrow & \bigcirc i_i) \\
\Box((s_i \wedge \neg a_i \wedge w) & \Rightarrow & \bigcirc i_i) \\
\Box((s_i \wedge \neg a_i \wedge r) & \Rightarrow & \bigcirc s_i) \\
\Box((s_i \wedge \neg a_i \wedge t) & \Rightarrow & \bigcirc s_i) \\
\Box((m_i \wedge \neg a_i \wedge w) & \Rightarrow & \bigcirc i_i) \\
\Box((m_i \wedge \neg a_i \wedge r) & \Rightarrow & \bigcirc s_i) \\
\Box((m_i \wedge \neg a_i \wedge t) & \Rightarrow & \bigcirc m_i)
\end{array}$$

The transitions that can be taken when a processor is active are as follows.

$$\begin{array}{rcl}
\Box((i_i \wedge a_i) & \Rightarrow & (w \vee r)) \\
\Box((s_i \wedge a_i) & \Rightarrow & (w \vee t)) \\
\Box((m_i \wedge a_i) & \Rightarrow & t)
\end{array}$$

The negation of the two required properties for 6 processes are given below, i.e. conjoining the following with the specification should be unsatisfiable

- *non co-occurrence for s and m:*

  $\Diamond((m_1 \wedge s_2) \vee (m_1 \wedge s_3) \vee (m_1 \wedge s_4) \vee (m_1 \wedge s_5) \vee (m_1 \wedge s_6) \vee (m_2 \wedge s_3) \vee (m_2 \wedge s_4) \vee (m_2 \wedge s_5) \vee (m_2 \wedge s_6) \vee (m_3 \wedge s_4) \vee (m_3 \wedge s_5) \vee (m_3 \wedge s_6) \vee (m_4 \wedge s_5) \vee (m_4 \wedge s_6) \vee (m_5 \wedge s_6));$

- *at most one process can be in state m:*

  $\Diamond((m_1 \wedge m_2) \vee (m_1 \wedge m_3) \vee (m_1 \wedge m_4) \vee (m_1 \wedge m_5) \vee (m_1 \wedge m_6) \vee (m_2 \wedge m_3) \vee (m_2 \wedge m_4) \vee (m_2 \wedge m_5) \vee (m_2 \wedge m_6) \vee (m_3 \wedge m_4) \vee (m_3 \wedge m_5) \vee (m_3 \wedge m_6) \vee (m_4 \wedge m_5) \vee (m_4 \wedge m_6) \vee (m_5 \wedge m_6)).$

3.3. Petri Nets

Consider now k-safe Petri Nets (see for example [22]), which are used to model systems with limited resources. In k-safe Nets, every 'place' may contain at most k tokens. This restriction allows us to represent k-safe Petri Nets, for a fixed value of k, in propositional temporal logic. Encoding places as propositions (proposition p_i^j is true if, and only if, place P_i contains j tokens), given a k-safe Petri Net $\mathcal{N}$, one can construct a PTL formula $\phi_{\mathcal{N}}$ of the size polynomial in the size of $\mathcal{N}$, such that models of $\phi_{\mathcal{N}}$ correspond to infinite trajectories of $\mathcal{N}$.

A Net is a tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ where $\mathcal{P}$ is a set of places, $\mathcal{T}$ a set of transitions, $\mathcal{F}$ is the flow relation and M_0 is the initial marking such that

$$\begin{array}{rcl} \mathcal{P} & = & \{P_1, P_2, \dots P_n\}; \\ \mathcal{T} & = & \{x_1, x_2, \dots x_m\}; \\ \mathcal{F} & \subseteq & (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}); \end{array}$$

and $M_0 : \mathcal{P} \to \mathbb{N}$. Generally, a mapping $M : \mathcal{P} \to \mathbb{N}$ is called a *marking* of $\mathcal{N}$. A transition $x \in \mathcal{T}$ is *enabled* at M if for every P such that $(P, x) \in \mathcal{F}$ we have $M(P) \neq 0$. For markings M, M' we write $M \longrightarrow M'$ if, and only if, some transition $x \in \mathcal{T}$ is enabled at M and for every place $P \in \mathcal{P}$ we have

$$M'(P) = M(P) + F(x, P) - F(P, x),$$

where $F(x, y)$ is 1 if $(x, y) \in \mathcal{F}$ and 0 otherwise. We say that marking M' is reachable in $\mathcal{N}$ if $M_0 \longrightarrow^* M'$, where the relation $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. The *reachability problem* is to determine given a Net $\mathcal{N}$ and a marking M whether M' is reachable in $\mathcal{N}$.

A marking is k-safe if for every $P \in \mathcal{P}$ we have $M(P) \leq k$. We say that a Petri Net is k-safe (or k-bounded) if all its reachable markings are k-safe. Deciding if a Net is k-safe is a PSPACE-complete problem [41], however, for many interesting Nets k-safety can be guaranteed by construction [7]. We can represent a k-safe Petri Net as follows. We abuse notation by letting P_i denote both a place in the Petri Net and a proposition in its logical representation meaning that the place contains one or more tokens. Similarly x_j denotes both a transition in the Petri Net and a proposition representing that transition x_j is fired. Firstly the following constraint, for each $x_i \in \mathcal{T}$, states that exactly one transition fires at any moment in time.

$$\{x_1, \dots x_m\}^{=1}$$

For each $P_i \in \mathcal{P}$ exactly one of the propositions representing the number of tokens in this place can be true.

$$\{p_i^0, \dots p_i^k\}^{=1}$$

We use P_i to show when there are tokens in a place. For $P_i \in \mathcal{P}$

$$\Box((p_i^1 \vee \dots p_i^k) \Leftrightarrow P_i)$$

For each $(P_j, x_i) \in \mathcal{F}$ the pre-condition of transitions is represented as follows.

$$\Box(x_i \Rightarrow P_j)$$

For each $(P_j, x_i) \in \mathcal{F}$ the effect of each transition is represented as follows for each $1 \leqslant h \leqslant k$

$$\Box((p_j^h \wedge x_i) \Rightarrow \bigcirc(p_j^{h-1}))$$

Recall that for k-safe Petri Nets and $(x_i, P_j) \in \mathcal{F}$ it is not possible that a transition x_i fires when place P_j already contains k tokens. Thus, for each $(x_i, P_j) \in \mathcal{F}$ the effect of each transition can be represented as follows for each $0 \leqslant h < k$

$$\Box((p_j^h \land x_i) \Rightarrow \bigcirc(p_j^{h+1})).$$

Additionally we must add frame conditions so that the number of tokens in places unrelated to the firing transition remain the same for each $P_j \in \mathcal{P}$, for each $0 \leqslant h \leqslant k$.

$$\begin{array}{lcl} \Box((p_j^h \land \bigwedge_{(P_j,x_i)\in\mathcal{F}} \neg x_i \land \bigwedge_{(x_i,P_j)\in\mathcal{F}} \neg x_i) & \Rightarrow & \bigcirc p_j^h) \\ \Box((\neg p_j^h \land \bigwedge_{(P_j,x_i)\in\mathcal{F}} \neg x_i \land \bigwedge_{(x_i,P_j)\in\mathcal{F}} \neg x_i) & \Rightarrow & \bigcirc \neg p_j^h) \end{array}$$

For example, consider the 1-safe Petri Net (similar to one in [48]), given in Fig. 2, representing the dining philosophers problem [36] for the case of four philosophers. This problem relates to

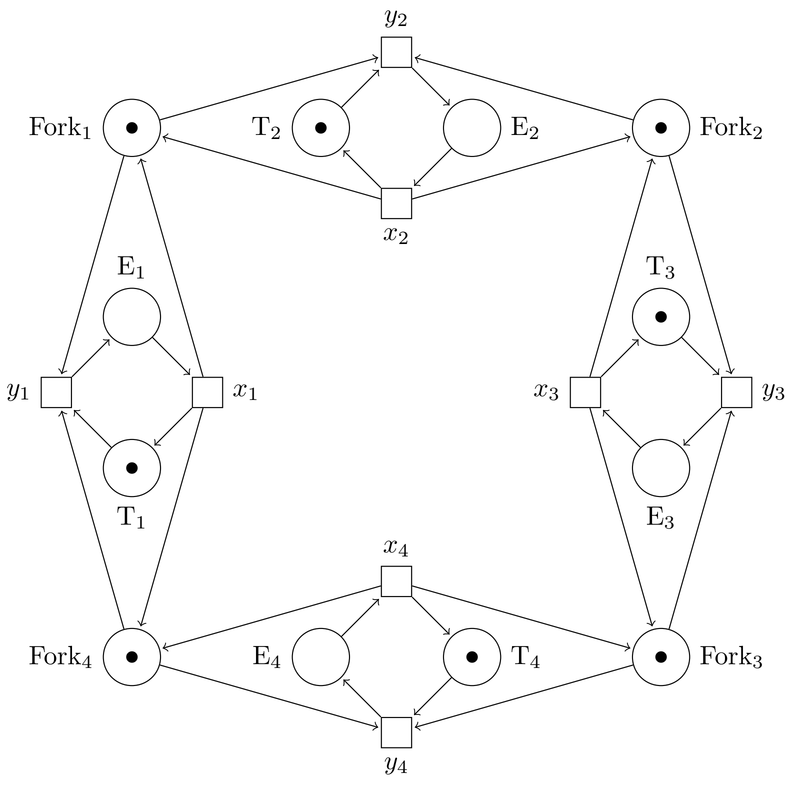

Figure 2: Four dining philosophers Petri Net.

providing processes with concurrent access to a limited number of resources. Here as there is only at most one token in each place we do not introduce the propositions (p_i^j) above but let the propositions representing place names (eg Fork$_1$) denote the presence of a token. This Petri Net can be represented as the conjunction of transition representations

$\Box(x_1 \Rightarrow \mathrm{E}_1)$	Pre-condition for transition x_1
$\Box(x_1 \Rightarrow \bigcirc(\neg\mathrm{E}_1 \land \mathrm{Fork}_1 \land \mathrm{Fork}_4 \land \mathrm{T}_1))$	Effect of transition x_1
$\Box(y_1 \Rightarrow \mathrm{Fork}_1 \land \mathrm{Fork}_4 \land \mathrm{T}_1)$	Pre-condition for transition y_1
$\Box(y_1 \Rightarrow \bigcirc(\neg\mathrm{Fork}_1 \land \neg\mathrm{Fork}_4 \land \neg\mathrm{T}_1 \land \mathrm{E}_1))$	Effect of transition x_1

... *(similarly for other transitions)*

frame conditions

$\square(\mathrm{Fork}_1 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg y_1 \wedge \neg y_2 \Rightarrow \bigcirc \mathrm{Fork}_1)$	Fork_1 can only change due to
$\square(\neg \mathrm{Fork}_1 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg y_1 \wedge \neg y_2 \Rightarrow \bigcirc \neg \mathrm{Fork}_1)$	transition x_1, x_2, y_1 and y_2
... *(similarly for other places)*	

and the constraint

$$\{x_1, \ldots, x_4, y_1, \ldots, y_4\}^{=1},$$

which states that exactly one transition fires at any moment in time. Note that the representation of the transitions (the first four implications) shows the pre-conditions and effects of taking particular transitions. For example given the situation shown in Figure 2 we can make the transition y_1 hold as the pre-conditions for taking transition y_1 are satisfied (the third formula). The constraint means that no other transition can be taken at the same same. The effects are (in the next moment in time) to remove the tokens from places Fork_1, Fork_4 and T_1 and for a token to be put at E_1 so philosopher one can eat. Note that due to the well known frame problem we must explicitly state that the tokens remain in all other places not affected by the selected transition.

Reachability in such Nets, for example the reachability of the state E_4, corresponds to the satisfiability of $\diamondsuit E_4$ from an initial state. Since the reachability problem (as well as many other interesting problems) already for 1-safe Nets is PSPACE-complete [22], such a translation is optimal.

We can then use cardinality constraints to impose *place invariants*: for a subset of places in a Petri Net, the total number of tokens in places from this subset remains constant. Such invariants are used, for example, in the verification of distributed protocols with Petri Nets [42, 43]. Note that imposing such extra restrictions actually makes the complexity of reasoning *lower*.

3.4. Other Examples

Other examples that have commonly been specified using temporal logics that may benefit from the use of constraints are systems such as the modelling of train movement, see for example [25, 23, 47]. Typical constraints here, for example, require that each station can be occupied by at most m trains, each section of the track can contain at most one train, the signals cannot be red and green at the same time, etc. Similarly the specification of lift controllers for example [3, 58] exhibit a number of constraints such as the lift may be at exactly one floor at any moment etc.

Robot swarms provide another area of possible applications. Swarm robots are a group of simple robots often with simple control mechanisms that aim to stay together as a connected group and carry out some task such as collecting food. In [56] we used temporal logic to specify a particular robot control algorithm, known as the alpha algorithm [46], that depends on local wireless communication, with the aim at showing that the robots maintain a connected group. The robots have positions on a grid and move in particular directions. The problem gives rise to a number of constraints for example that each robot is in exactly one location, each grid square contains at most one robot, each robot is travelling is exactly one direction etc. Similarly in [4] we formalise the algorithm for a swarm of foraging robots using a number of transition systems for each robot and the food it is collecting. As well as the constraints that each robot can be in exactly one mode at any moment there are additional constraints that synchronise the transition systems for the food with those for the robots.

4. Satisfiability for TLC

Next we provide an algorithm for checking satisfiability of TLC formulae and also give the upper complexity bound on satisfiability of TLC by the explicit construction of a directed graph known as a *behaviour graph.*

4.1. Behaviours Graphs

The notion of a behaviour graph for a set of temporal clauses was introduced in [28] and adapted to the unconditional sometime clauses in [16]. It is a directed graph for a set of temporal clauses such that (after reductions) any infinite path through the graph is a model for the set of temporal clauses. Satisfiability of TLC formulae can be checked by being able to construct a non-empty graph. In what follows, we estimate the size of the graph and time needed both for its construction and for checking satisfiability.

Definition 3. *(Behaviour Graph/Reduced Behaviour Graph) Given a formula φ in the normal form over a set of (both constrained, $\mathcal{C}$, and unconstrained) literals* Props, *we construct a finite directed graph G as follows. The nodes I of G are interpretations (subsets) of* Props, *satisfying the required constraints, i.e. $I \models_{PL} \mathcal{C}$.*
A node, I, is designated an initial node of G if $I \models_{PL} \bigvee_i l_i$ for every initial clause **start** $\Rightarrow \bigvee_i l_i$ *of the given temporal formula. For each node, I, we construct an edge in G to a node I' if, and only if, the following condition is satisfied:*

- *For every step rule, $\bigwedge_i k_i \Rightarrow \bigcirc \bigvee_j l_j$, if $I \models_{PL} \bigwedge_i k$ then $I' \models_{PL} \bigvee_j l_j$.*

The behaviour graph, *H, of φ is the maximal subgraph of G given by the set of all nodes reachable from initial nodes. The* reduced behaviour graph, *H_R, of φ is a graph obtained from the behaviour graph of φ by repeated deletion of nodes I where*

a) I does not have a successor; or

b) for some sometime clause **true** $\Rightarrow \Diamond m$ *within φ, there is no path from I to a node J where m is true, that is, $J \models_{PL} m$.*

Theorem 4. *A TLC formula in the normal form φ is satisfied if, and only if, its reduced behaviour graph is non-empty.*

Proof. The definition of a TLC behaviour graph given above differs from the behaviour graph for PTL, defined in [16], in that we never construct nodes that do not satisfy the set of constraints. Since the expressive power of PTL and TLC is the same, as shown in Lemma 2, this restriction can be imposed by encoding the constraint as a PTL formula. Let ψ be a propositional logic formula *equivalent* to the constraints $\mathcal{C}$, i.e. for any I, $I \models_{PL} \mathcal{C}$ if, and only if, $I \models_{PL} \psi$. Translating $\Box\psi$ into temporal clauses will result in a number of initial and step clauses. In terms of the behaviour graph in [16] by construction every initial node must satisfy the initial clauses derived from ψ and every non-initial node must satisfy the step clauses derived from ψ. Hence we would never construct an initial node that does not not satisfy ψ, further we would never construct an edge to a node not satisfying ψ. That is, nodes not satisfying ψ would be unreachable and thus could not form part of the behaviour graph. $\Box$

The link between the satisfiability of TLC formulae and properties of the behaviour graph allows us to investigate the complexity of our logic. First we notice that the satisfiability problem for PTL with just one proposition is already PSPACE-complete [17]; therefore, the complexity of the satisfiability problem for TLC is also PSPACE-complete. Notice further that if we restrict our consideration to formulae in Separated Normal Form, the size of the TLC behaviour graph is exponential in the number of unconstrained propositions and only polynomial in the number of constrained propositions.

Theorem 5. *Satisfiability of a TLC($\mathcal{C}_1^{\propto_1 m_1}, \dots \mathcal{C}_n^{\propto_n m_n}$) formula φ in Separated Normal Form can be decided in time*

$$O\left(|\varphi| \times \left(|\mathcal{C}_1^{\propto_1 m_1}|^{m_1} \times \cdots \times |\mathcal{C}_n^{\propto_n m_n}|^{m_n} \times 2^{|\mathcal{A}|}\right)^3\right)$$

where $|\varphi|$ is the length of φ, $|\mathcal{C}_i^{\propto_i m_i}|$ is the size of the set $\mathcal{C}_i^{\propto_i m_i}$ of constrained literals, and $|\mathcal{A}|$ is the size of the set $\mathcal{A}$ of unconstrained propositions occurring in φ.

Proof. There exist $O(|\mathcal{C}^{\propto_1 m_1}|^{m_1} \times \cdots \times |\mathcal{C}_n^{\propto_n m_n}|^{m_n} \times 2^{|\mathcal{A}|})$ different interpretations of propositions from Props; moreover, they can all be enumerated in time $O(|\mathcal{C}^{\propto_1 m_1}|^{m_1} \times \cdots \times |\mathcal{C}_n^{\propto_n m_n}|^{m_n} \times 2^{|\mathcal{A}|})$. Let N be the number of such different interpretations of propositions from Props. The behaviour graph G for ϕ can be constructed in $O(N^2)$ time.

To reduce the behaviour graph we do the following. A node without successors can be found in $O(N^2)$ time. To find all nodes satisfying condition b), for every eventuality $\mathbf{true} \implies \Diamond m$ we first mark all nodes containing m and then work backwards to mark all nodes with a path to a marked node. Every unmarked node satisfies condition b). This can be done in $O(|\phi| \times N^2)$ time. (The $|\phi|$ factor comes from the necessity to check this condition for every eventuality $\mathbf{true} \implies \Diamond m$.) This process should be repeated until there is no change. Clearly, the maximal number of nodes to delete is N and thus the process can be repeated at most N times.

Overall, the complexity of reducing the behaviour graph, as well as the complexity of the entire procedure, is $O(|\phi| \times N^3)$. □

As the TLC formulae stemming from our case studies are all in, or close to, the normal form. In practice, this result suggests that, by exploiting cardinality constraints in TLC, we can achive a better performance on relevant problems than algorithms that have no built in facilities to handle such constraints.

4.2. Incremental Algorithm

Based on Theorem 4 one can provide an algorithm checking the satisfiability of TLC formulae. A straightforward approach is to construct the graph G representing all possible interpretations of Props that satisfy the constraints, and then 'carve' the behaviour graph H from G. However, such a procedure might consider some nodes that are actually unreachable from the initial nodes and, thus, carry out excess work. Instead, in Algorithm 1, we present an incremental, tableaux-like algorithm, which avoids building these unnecessary nodes.

Let Assignments$(\varphi, \mathcal{C})$ be a function which, when given a formula φ and a set of constraints $\mathcal{C}$, returns the set of *all* interpretations within the language Props that both satisfy $\mathcal{C}$ and make φ true. Clearly, Assignments$(\varphi, \{C_1^{\propto_1 m_1}, \ldots, C_n^{\propto_n m_n}\})$ can be computed deterministically in time $O(int)$ where $int = (|C^{\propto_1 m_1}|^{m_1} \times \cdots \times |C_n^{\propto_n m_n}|^{m_n} \times 2^{|\mathcal{A}|})$ returning at most $O(int)$ interpretations for any φ.

Example. If Props $= \{p, q, r, s\}$ then Assignments$(p \vee q, \{\{p, q, r, s\}^{=1}\})$ will return two interpretations (where propositions not explicitly mentioned are assumed to be false): $\{p\}$ and $\{q\}$; whereas Assignments$(p \vee q, \{\{p, q\}^{=1}, \{q, r, s\}^{=2}\})$ will return three: $\{p, r, s\}$, $\{q, r\}$, and $\{q, s\}$.

We use Assignments$(\varphi, \mathcal{C})$ to construct nodes of the behaviour graph H for a formula φ incrementally (see Algorithm 1 *ConstructBehaviourGraph*$(\varphi, \mathcal{C})$). Notice that the size of graph H is still worst-case quadratic in *int*.

Nodes of H can be *marked* or *unmarked*. The for-loop in lines 8-13 selects an unmarked node, marks it and adds its successors to H where they have not yet been added. Thus (outside this loop) a node is marked if all its successors are already represented in H, otherwise, it is unmarked. Note that if the set of temporal clauses contains no initial clauses, then the formula ψ in line 1 of the algorithm is **true**, and if the conjunction in line 7 is empty then χ is **true**. After the behaviour graph has been constructed, we compute the reduced behaviour graph in time cubic in the size of the behaviour graph.

To construct the reduced behaviour graph we carry out deletions of nodes with no successors and for any eventuality clause $\mathbf{true} \Rightarrow \Diamond m$ deletion of nodes where there is no path to a node satisfying m (see Algorithm 2). To achieve the latter we must find a *terminal subgraph* B_m, where $\neg m$ holds at each node, from the behaviour graph H. A terminal subgraph B_m is one such that any edge from a node in B_m leads to a node also in B_m. Thus $\Diamond m$ cannot be satisfied on any path from any node in B_m. So the terminal subgraph must be deleted as it doesn't satisfy the eventuality clause $\mathbf{true} \Rightarrow \Diamond m$. In Algorithm 2 Delete$(B_m)$, where B_m is a subset of nodes in H, is a procedure that deletes all the nodes in B_m from H and any edges to, or from, a node in B_m. Algorithm 3 constructs the graph and carries out the deletions.

Algorithm 1 ConstructBehaviourGraph($\varphi, \mathcal{C}$)

```
Let $\psi = \bigwedge\{R_j \mid \textbf{start} \Rightarrow R_j$ is an initial clause in $\varphi\}$
for all $I$ in Assignments($\psi, \mathcal{C}$) do
  Add an unmarked node $I$ to $H$
end for
while Not all nodes in $H$ are marked do
  Pick an unmarked node $I$ and mark $I$
  Let $\chi = \bigwedge\{R_k \mid L_k \Rightarrow \bigcirc R_k$ is a step clause in $\varphi$ s.t. $I \models_{PL} L_k\}$
  for all $J$ in Assignments($\chi, \mathcal{C}$) do
    if $J$ is not already in $H$ then
      Add an unmarked node $J$ to $H$
    end if
    Add an edge $(I, J)$ to $H$
  end for
end while
```

Algorithm 2 EventualityDelete(H, m)

```
Let $B_m = \{$nodes $I \in H \mid I \models_{PL} \neg m\}$
repeat
  for all nodes $I$ in $B_m$ do
    if $(I, J)$ in $H$ and $J \notin B_m$ then
      $B_m = B_m - \{I\}$
    end if
  end for
until $B_m$ does not change
Delete($B_m$)
```

Algorithm 3 ReducedBehaviourGraph($\varphi, \mathcal{C}$)

```
H = ConstructBehaviourGraph($\varphi, \mathcal{C}$)
repeat
  if there exists $I$ in $H$ s.t. there is no $J$ in $H$ where $(I, J)$ is an edge in $H$ then
    Delete($\{I\}$)
  end if
  for each true $\Rightarrow \Diamond m$ in $\varphi$ do
    EventualityDelete($H, m$)
  end for
until $H$ does not change
```

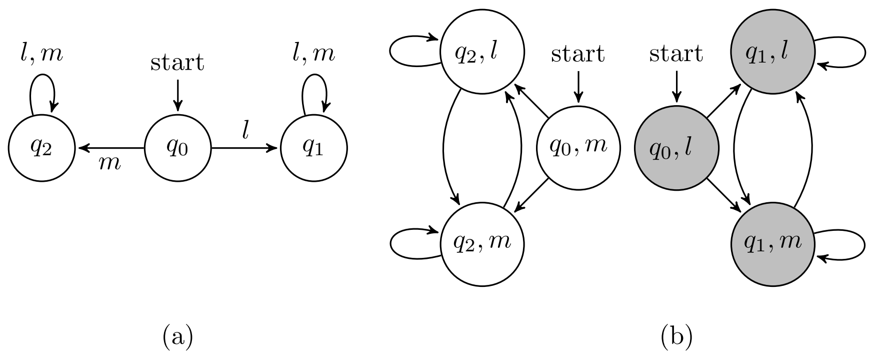

Figure 3: Labelled transition system (a) and behaviour graph (b).

Theorem 6. *Given a TLC($\mathcal{C}$) formula φ in Separated Normal Form Algorithm 3 terminates and constructs the reduced behaviour graph H_R of φ.*

Proof. First we show that the outcome of Algorithm 3 is the reduced behaviour graph for φ. Let G, H and H_R be as in Definition 3 and let H' be the outcome of Algorithm 1 and H'_R be the outcome of Algorithm 3, respectively. It should be clear that H' is a subgraph of H.

Conversely, consider $I_1, \ldots, I_n$, a path in G such that I_1 is an initial node. It can be seen that $I_1 \in \mathsf{Assignments}(\psi, \mathcal{C})$, where $\psi = \bigwedge\{R_j \mid \textbf{start} \Rightarrow R_j$ is an initial clause in $\varphi\}$ and $I_{i+1} \in \mathsf{Assignments}(\chi_i, \mathcal{C})$, where $1 \leqslant i < n$ and $\chi_i = \bigwedge\{R_k \mid L_k \Rightarrow \bigcirc R_k$ is a step clause in φ s.t. $I_i \models_{PL} L_k\}$. Therefore, by lines 10 and 12 of Algorithm 1, every I_i is a node in H' and $I_1, \ldots, I_n$ is a path in H'. Thus H is a subgraph of H'. It can also be seen that if Algorithm 3 deletes a node I from H', then $I \notin H_R$. Finally, it can be seen that if no node can be deleted from H'_R by Algorithm 3 H'_R is a reduced behaviour graph.

Next we show that Algorithms 1–3 terminate. Algorithm 1 constructs the behaviour graph. This uses a finite set of clauses and constraints over a finite set of propositions (Props). Nodes are interpretations of Props that satisfy the constraints so there are a finite number. In Algorithm 1 the nodes and edges are constructed incrementally. Nodes are marked once they are selected for expansion (line 6) so will not be selected again. Additionally when successors are constructed (lines 7-12) new nodes are only constructed if they do not already exist in the graph. Together this ensures that the algorithm will terminate. Algorithm 2 takes a (finite) subset of nodes of the behaviour graph and deletes those with the required property until there is no change. Hence Algorithm 2 terminates. Algorithm 3 calls Algorithm 1 which terminates and repeatedly carries out deletions until the graph does not change. Again, as the graph is finite this must terminate. □

4.3. Example

We illustrate the concept of a behaviour graph and the working of the algorithms by modelling a labelled transition system. Consider the transition system given in Fig. 3(a). Its evolution can be characterised by the following TLC($\mathcal{C}$) formula in Separated Normal Form

$$
\begin{array}{lllllllllll}
\textbf{start} \Rightarrow q_0 & (q_0 \wedge l) & \Rightarrow & \bigcirc q_1 & (q_1 \wedge l) & \Rightarrow & \bigcirc q_1 & (q_2 \wedge l) & \Rightarrow & \bigcirc q_2 \\
 & (q_0 \wedge m) & \Rightarrow & \bigcirc q_2 & (q_1 \wedge m) & \Rightarrow & \bigcirc q_1 & (q_2 \wedge m) & \Rightarrow & \bigcirc q_2,
\end{array}
$$

where the set of constraints is

$$\mathcal{C} = \{\{q_0, q_1, q_2\}^{=1}, \{l, m\}^{=1}\}.$$

(That is, at every moment the system is in exactly one state performing exactly one transition.) In addition, we require that the state q_1 is visited infinitely often.

$$\textbf{true} \Rightarrow \Diamond q_1$$

Now, for $\psi = q_0$ (notice that **start** $\Rightarrow q_0$ is the only initial clause), $\mathsf{Assignments}(q_0, \mathcal{C}) = \{\{q_0, l\}, \{q_0, m\}\}$. Thus, the behaviour graph for ϕ contains two initial nodes, $\{q_0, l\}$, $\{q_0, m\}$, both of which are initially unmarked. Suppose $\{q_0, l\}$ is chosen first in line 6 of Algorithm 1. Then $\chi = q_1$ and $\mathsf{Assignments}(q_1, \mathcal{C}) = \{\{q_1, l\}, \{q_1, m\}\}$, which are added to the graph. Similarly, $\{\{q_2, l\}, \{q_2, m\}\}$ are introduced as successors of $\{q_0, m\}$. The algorithm proceeds and constructs the behaviour graph depicted in Fig. 3(b). It is not hard to see that the reduced behaviour graph obtained in Algorithm 3 by eliminating nodes not satisfying the eventuality only contains nodes $\{q_0, l\}$, $\{q_1, l\}$ and $\{q_1, m\}$ shaded grey in Fig. 3(b), i.e. the formula and the constraints are satisfiable.

5. BeTL: a Satisfiability Checker for TLC

BeTL (Behaviour graph for Temporal Logic) is a prototype satisfiability checker we have developed that implements the incremental behaviour graph construction algorithm given in Algorithm 3 of Section 4. As input it accepts a TLC formula and a set of constraints. BeTL is written in Java$^{\text{TM}}$ programming language using JDK (J2SE Development Kit) 5.0.

5.1. Core Algorithm

BeTL implements the algorithms presented in Section 4. The function $\mathsf{Assignments}(\psi, \mathcal{C})$, where ψ is a formula in conjunctive normal form (CNF) and $\mathcal{C}$ a set of constraints, is implemented using the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [14] to assign the truth values of each proposition in the nodes. Here we require all the truth assignments of the propositions satisfying both ψ and $\mathcal{C}$ so we need to modify DPLL to return these assignments rather than just the truth or falsity of ψ and $\mathcal{C}$. The sets of assignments that satisfy a constraint $\mathcal{C}$ is enumerated and then merged with the set of assignments satisfying the already calculated constraints. Each assignment is used to apply the usual unit propagation to ψ.

Algorithm 4 gives the function call_DP$_{mod}(\Gamma, \Sigma)$ which implements $\mathsf{Assignments}(\psi, \mathcal{C})$. This function takes Γ, a set of sets of literals which satisfy the constraints $\mathcal{C}$, and Σ, a set of sets of literals representing the CNF formula ψ. For example, to compute

$$\mathsf{Assignments}((\neg a \vee b) \wedge (\neg b \vee \neg c), \{\{a, b, c\}^{=1}\})$$

we would call call_DP$_{mod}(\Gamma, \Sigma)$ where

$$\Gamma = \{\{a, \neg b, \neg c\}, \{\neg a, b, \neg c\}, \{\neg a, \neg b, c\}\} \qquad \text{and} \qquad \Sigma = \{\{\neg a, b\}, \{\neg b, \neg c\}\}.$$

For each set of the literals that satisfy the constraints (α), the usual unit propagation algorithm, UP(l, Σ) in Algorithm 5 is called for each $l \in \alpha$. This deletes sets of literals from Σ that contain l and deletes $\neg l$ from any other sets. Δ is used to store the sets of assignments that satisfy both the sets of constraints and the CNF formula represented by Σ.

Algorithm 4 call_DP$_{mod}(\Gamma, \Sigma)$

```
1: Let $\Delta = \{\}$
2: for each $\alpha \in \Gamma$ do
3:     $\Sigma_\alpha = \Sigma$
4:     for each literal $l \in \alpha$ do
5:         $\Sigma_\alpha := \mathrm{UP}(l, \Sigma_\alpha)$
6:     end for
7:     $\Delta = \Delta \cup \mathrm{DP}_{mod}(\alpha, \Sigma_\alpha)$
8: end for
9: return $\Delta$
```

In Algorithm 6 the DP$_{mod}$ algorithm is given which is almost identical to the original DPLL algorithm [14]. The difference is that, instead of returning 'satisfiable' or 'unsatisfiable', DP$_{mod}$ returns all the possible assignments of propositions that both satisfy Σ and the given constraints. This is called recursively until Σ is empty ($\Sigma = \{\}$) or unsatisfiability is derived, i.e. $\{\} \in \Sigma$.

Algorithm 5 UP(l, Σ)

```
1: for each $\alpha \in \Sigma$ do
2:     if $\alpha = \{\}$ then return $\{\{\}\}$
3:     if $l \in \alpha$ then $\Sigma := \Sigma - \alpha$
4:     if $\neg l \in \alpha$ then $\alpha = \alpha - \{\neg l\}$
5: end for
6: return $\Sigma$
```

Algorithm 6 DP$_{mod}(\beta, \Sigma)$

```
1: (Sat) if $\Sigma = \{\}$ then return $\{\beta\}$
2: (Empty) if $\{\} \in \Sigma$ then return $\{\}$
3: (Unit propagation) if $\{l\} \in \Sigma$ then return DP$_{mod}(\beta \cup \{l\}, \text{UP}(l, \Sigma))$
4: (Split) if $\alpha \in \Sigma$ and $l \in \alpha$ then
   return DP$_{mod}(\beta \cup \{l\}, \text{UP}(l, \Sigma)) \cup$ DP$_{mod}(\beta \cup \{\neg l\}, \text{UP}(\neg l, \Sigma))$
```

5.2. Example

Continuing the example in section 4.3. Recall that

$$\mathcal{C} = \{\{q_0, q_1, q_2\}^{=1}, \{l, m\}^{=1}\}.$$

Using a semantic tree construction we evaluate the literals satisfying the constraints as follows.

$$\begin{aligned}\Gamma &= \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{q_0, \neg q_1, \neg q_2, \neg l, m\}, \{\neg q_0, q_1, \neg q_2, l, \neg m\}, \{\neg q_0, q_1, \neg q_2, \neg l, m\},\\ &\quad \{\neg q_0, \neg q_1, q_2, l, \neg m\}, \{\neg q_0, \neg q_2, q_3, \neg l, m\}\}\end{aligned}$$

To construct the initial nodes we must evaluate $\mathsf{Assignments}(q_0, \mathcal{C})$ and do this by calling $call\_DP_{mod}(\Gamma, \{\{q_0\}\})$. In step 1 of the Algorithm 4 we set $\Delta = \{\}$. Then in step 2 let $\alpha = \{q_0, \neg q_1, \neg q_2, l, \neg m\}$, and in step 3 set $\Sigma_\alpha = \{\{q_0\}\}$. Steps 4, 5, 6, unit propagate q_0, $\neg q_1$, $\neg q_2$, l and $\neg m$ through Σ_α in turn as follows.

$$\begin{aligned}\Sigma_\alpha &= UP(q_0, \Sigma_\alpha) = \{\}\\ \Sigma_\alpha &= UP(\neg q_1, \{\}) = UP(\neg q_2, \{\}) = UP(\neg m, \{\}) = UP(l, \{\}) = \{\}\end{aligned}$$

On line 7 of Algorithm 4

$$\text{DP}_{mod}(\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{\}) = \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}\}$$

so

$$\Delta = \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}\}.$$

Next time round the loop where

$$\alpha = \{q_0, \neg q_1, \neg q_2, \neg l, m\}$$

similarly on line 7 we obtain

$$\text{DP}_{mod}(\{q_0, \neg q_1, \neg q_2, \neg l, m\}, \{\}) = \{\{q_0, \neg q_1, \neg q_2, \neg l, m\}\}$$

so

$$\Delta = \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{q_0, \neg q_1, \neg q_2, \neg l, m\}\}.$$

Next when

$$\alpha = \{\neg q_0, q_1, \neg q_2, l, \neg m\}$$

on line 6

$$\Sigma_\alpha = UP(\neg q_0, \{\{q_0\}\}) = \{\{\}\}$$

and at the end of the for loop on line 6

$$\Sigma_\alpha = \{\{\}\}.$$

Now on line 7

$$\mathrm{DP}_{mod}(\{\neg q_0, q_1, \neg q_2, \neg l, m\}, \{\{\}\} = \{\}$$

so

$$\begin{aligned}\Delta &= \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{q_0, \neg q_1, \neg q_2, \neg l, m\}\} \cup \{\} \\ &= \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{q_0, \neg q_1, \neg q_2, \neg l, m\}\}.\end{aligned}$$

The remaining values for α are similar and at the end of line 6, $\Sigma_\alpha = \{\{\}\}$ in each case. In line 7 $\mathrm{DP}_{mod}(\alpha, \{\{\}\} = \{\}$ and so

$$\Delta = \{\{q_0, \neg q_1, \neg q_2, l, \neg m\}, \{q_0, \neg q_1, \neg q_2, \neg l, m\}\}$$

is returned. This is the same as the result returned from $\mathsf{Assignments}(q_0, \mathcal{C})$ in Section 4.3. Similar steps are carried out each time $\mathsf{Assignments}(\chi, \mathcal{C})$ is called for different values of χ.

5.3. Experiments

We have successfully applied BeTL to the temporal specifications stemming from case studies considered in Section 3[1]. In all cases it took BeTL under 600 seconds to compute the expected answer on a PC with a 2.13 GHz Intel Core 2 Duo E6400 processor, 3GB main memory, and 5GB virtual memory running Fedora release 9 with a 32-bit Linux kernel. However, it is important to re-affirm that BeTL is essentially a *prototype*. It is intended to be the first implementation of a TLC satisfiability checker, and to provide a benchmark for subsequent, more efficient, systems.

In order to evaluate BeTL's performance, we compared it with two existing PTL tableau theorem provers, the Logics Workbench (LWB) [39] and the Tableau Work Bench (TWB) [1], on a number of randomly generated benchmark problems. The Logics Workbench [39] is a suite of logical tools for propositional modal and temporal logics including PTL. Here we use the *satisfiability* function in the PTL module that implements Janssen's tableau algorithm [40] which constructs a tableau using a two pass style algorithm, first constructing the tableau and then deleting nodes. The implementation of Janssen's algorithm from the LWB is selected over that of Schwendimann's One Pass Tableau [50] (the LWB *model* function) as our implementation is also a two pass style. The Tableau Workbench [1] is a framework for constructing tableau provers for arbitrary propositional logics. It has a several modules with pre-defined calculi for a number of modal and temporal logics including PTL. We selected these implementations as they are both tableau-based, stable and easily available. We have focussed on tableau-based implementations rather than other styles of prover, eg resolution, so as to focus on the effect of the constraints rather than the algorithms or engineering of particular implementations. There are other implementations of PTL provers we could have compared with which are discussed further in Section 6.1. We translate the constraints into PTL formulae as described in Section 2.4.

All of the experiments have been performed on the above mentioned PC. The time was limited to 600 seconds so a time in the table of > 600 indicates it did not finish within the allocated time. Any row in the table labelled constraints with entries n/m denotes that the problem contained n constrained propositions and m unconstrained propositions.

Randomly Generated Formulae. We considered 10 sets of benchmarks, where each set contains 100 randomly generated formulae. All formulae are generated using the following criteria:

- The total number of propositions is 10.
- The total number of initial and step clauses added together is 4 clauses.

[1]Problem files can be found at `http://www.csc.liv.ac.uk/~clare/software/constraints.html`

- The maximum length of each clause is 5, i.e., there can be at most 5 propositions in a clause and, in the case of step clause, the number of propositions on the right and left hand sides are randomly determined.
- Each proposition generated may be negated with the probability 0.5.
- There is only 1 constrained set.
- There are 10 sometimes clauses; one for each proposition in the specification.

The difference between each set is the number of constrained literals, meaning, in the first set, 1 literal is constrained and 9 unconstrained, where, in the sixth set, say, there are 6 constrained and 4 unconstrained propositions.

Below is the results of running BeTL, LWB [39] and TWB [1] on the 10 benchmark sets mentioned above. Note that in most cases, TWB did not finish within the 10-minute running time limit. Thus, in such cases, the value of '>600' seconds is used for the purpose of this comparison.

	1	2	3	4	5	6	7	8	9	10
BeTL	7.220	8.026	4.487	1.833	0.902	0.387	0.142	0.078	0.055	0.043
LWB	1.960	3.482	4.732	2.319	0.792	0.432	0.097	0.065	0.048	0.044
TWB	102.037	194.001	>600	>600	198.012	198.005	196.001	195.185	>600	196.034

Table 1: Average running time (in seconds) of BeTL, LWB and TWB on the benchmark sets

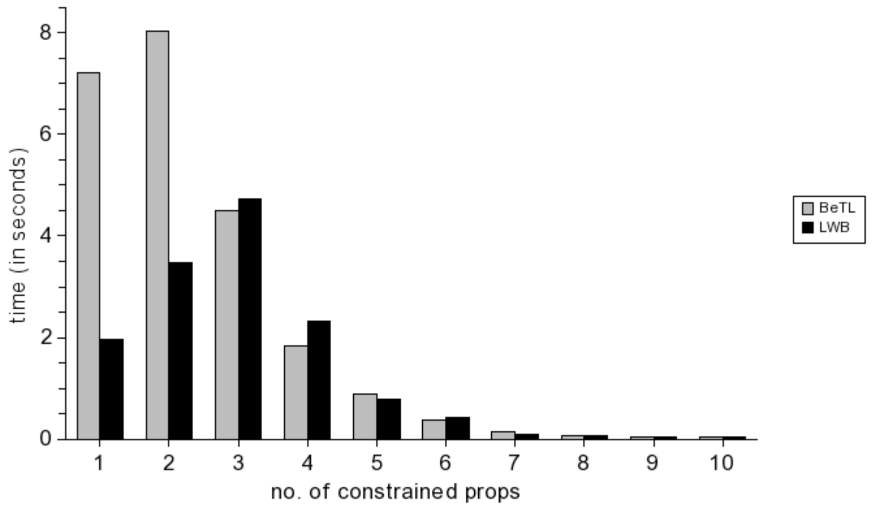

Figure 4: Comparison graph between BeTL and LWB on the benchmark sets

The experimental results in Table 1 show that the performance of BeTL gradually improves as the number of constrained literals in the specification increases. Since the timings for TWB are greater than for either BeTL or LWB, it is excluded in the comparison graph (Figure 4).

Random State Machines. In Table 2, we show BeTL's performance on specifications representing randomly generated state transition systems with 5, 10, 15 and 20 states. For example, if a state n_1 in the transition system had edges to states n_2 and n_3 this would be represented by the temporal formula, $\Box(p_{n_1} \Rightarrow \bigcirc(p_{n_2} \vee p_{n_3}))$ where the proposition p_{n_i} denotes being at the state n_i in the transition system. There is one constraint $\{p_{n_1}, p_{n_2}, \ldots p_{n_k}\}^{=1}$ where k represents the number of states in the transition system. Additionally we include formulae that specify that all the states in the system are visited infinitely often. Note that TWB is not included, because its overall results are very slow.

	5 states	10 states	15 states	20 states
BeTL	0.017	0.053	0.104	0.375
LWB	0.054	0.131	0.130	14.694

Table 2: Average running time (in seconds) of BeTL, LWB for the transition systems

Overlapping Constraints. We next consider three randomly generated sets of formulae where some literals occur in more than one constraint. We provide three parameters for each experiment of the form (sum, coverage, overlap) where if the three sets are C_1, C_2, C_3

$$\begin{aligned} \mathit{sum} &= |C_1| + |C_2| + |C_3| \\ \mathit{coverage} &= |(C_1 \cup C_2 \cup C_3)| \\ \mathit{overlap} &= |(C_1 \cap C_2)| + |(C_1 \cap C_3)| + |(C_2 \cap C_3))| \end{aligned}$$

One set of formulae is considered in each of Tables 3-5 with differing sets of constraints in each column (a), (b) and (c).

Set1 contains 10 propositions, 10 initial clauses, 20 step clauses, 4 eventualities, maximum clause length is 5. Each set has three constraints $C_1^{=1}$, $C_2^{=1}$, $C_3^{=1}$. All sets are satisfiable.

Set2 contains 10 propositions, 10 initial clauses, 20 step clauses, 4 eventualities, maximum clause length is 5. Each set has three constraints $C_1^{=2}$, $C_2^{=2}$, $C_3^{=2}$. All sets are satisfiable.

Set3 contains 15 propositions, 10 initial clauses, 20 step clauses, 5 eventualities, maximum clause length is 5. Each set has three constraints $C_1^{=1}$, $C_2^{=1}$, $C_3^{=1}$. All sets are satisfiable.

	a	b	c
constraints	(6,6,0)	(9,6,3)	(12,6,6)
BeTL	0.91	0.43	0.44
LWB	35.67	7.24	2.61
TWB	>600	> 600	> 600

Table 3: Running time (in seconds) of BeTL, LWB and TWB on overlapping constraints Set1

	a	b	c
constraints	(9,9,0)	(12,9,3)	(15,9,6)
BeTL	0.77	0.40	0.39
LWB	2.60	3.24	1.54
TWB	>600	> 600	> 600

Table 4: Running time (in seconds) of BeTL, LWB and TWB on overlapping constraints Set2

In each case TWB did not finish within the 10-minute running time limit and BeTL outperforms LWB.

Petri Nets. We give timings for running the four person dining philosophers problem. The columns marked (a) is the Petri Net protocol (satisfiable), the column marked (b) is the Petri Net protocol conjoined with the property that infinitely often each philosopher has a chance to eat (satisfiable) (c) the Petri Net protocol conjoined with the property that sometime philosopher 1 and philosopher 2 will eat (at the same time) (unsatisfiable). Whilst LWB is faster than BeTL all times are less than a second.

constraints	a (12,12,0)	b (15,12,3)	c (18,12,6)
BeTL	4.30	2.12	1.48
LWB	273.37	155.27	59.38
TWB	>600	> 600	> 600

Table 5: Running time (in seconds) of BeTL, LWB and TWB on overlapping constraints Set3

	a	b	c
BeTL	0.42	0.33	0.44
LWB	0.05	0.05	0.05

Table 6: Running time (in seconds) of BeTL and LWB on the Petri Net with Four Philosophers

5.4. Discussion.

We must yet again highlight that BeTL is a prototype and was developed as a first implementation not focusing on any fine tuning of performance. However, we have found that BeTL outperforms TWB on all the problems we considered—we note that that TWB was designed primarily for modularity and extensibility rather than efficiency [1]. Further, from Table 1 where the constraints include 8 or more of the 10 propositions BeTL's performance is comparable to LWB, which is highly optimised, both being less than 0.1 second. Table 1 also shows that BeTL's performance improves as the set of constraints includes a higher proportion of the set of propositions. We note that Table 1 and Figure 4 show the times for both LWB and BeTL increasing until 2-3 constrained propositions and then decreasing. We believe that this is probably due to the *phase transition* effect that has been studied in the area of SAT [31]. Essentially under-constrained problems are easily found to be satisfiable by provers and over constrained problems are easily found to be unsatisfiable. Here, adding more constraints makes the sets of formulae more likely to be unsatisfiable and hence easier for both provers to solve. We believe that the time to solve the problems goes up initially for both provers because the phase transition occurs at around 2-3 constraints.

Table 2 shows that BeTL has good performance results even compared to the LWB for formulae derived from the transition system examples. Similarly the results from the overlapping constraints, Tables 3-5 show BeTL having good performance as compared to LWB. We note that BeTL is slower than LWB on the Petri Net examples possibly due to the fact that there are more unconstrained propositions.

Whilst these results show the idea of incorporating constraints is promising, we note some issues with the underlying algorithm. Firstly, formulae are required to be in a particular normal form. This is not an issue for problems that are already or nearly in the normal form (e.g., the state transition examples in Table 2). For problems that require translation into the normal form, this may require introduction of some new propositions (to rename complex subformulae) and, this adds to the set of unconstrained propositions, whereas the results above show, BeTL works best when most propositions are constrained.

Regarding the algorithm, it constructs nodes that satisfy the set of constraints and a set of (classical logic) clauses. This has two problems. First, we have to explicitly enumerate sets of propositions that satisfy the set of constraints. Second, although the algorithm only generates reachable states, we are often forced to construct *all* the states, e.g., when there are no initial clauses. This immediately results in the worst case complexity. Inherently, the incremental behaviour graph construction adopts a breadth-first style that requires us to construct, and keep in memory, a large number of nodes, resulting in an unavoidable inefficiency.

To try and overcome these problems whilst still maintaining the benefits of dealing with constraints separately we aim to develop a more traditional tableau algorithm that does not require input in the normal form (see Section 6.2). Additionally, we could adopt a "branch by branch" construction method to avoid the breadth-first construction in the algorithm here. Finally such a tableau would not always have to explicitly evaluate the sets of propositions satisfying the constraints.

6. Concluding Remarks

In this paper we have introduced TLC, a propositional temporal logic that allows the specifier to define constraints on how many literals from some set can be satisfied at any one time. This logic represents a combination of standard propositional linear-time temporal logic with constraints relating to restrictions on the number of literals, for particular subsets of literals, at each moment in time. Work on TLC has uncovered a new, and potentially very sophisticated, approach to temporal specification. Rather than concentrating solely on the behaviour of components, the use of TLC encourages specifiers to partition the literals, and also to consider what constraints need to be put upon these partitioned sets. Thus, this leads us towards the approach of *engineering* the sets and constraints first, *before* even addressing the temporal specification of the component behaviours.

We provide a graph construction algorithm to check satisfiability by enumerating only the reachable nodes that satisfy the required constraints. Experiments show that an implementation of this outperforms an existing temporal logic tableau reasoner, TWB. Additionally if a high proportion of the propositions are constrained it can also outperform LWB. However we have identified some issues with the algorithm in that it explicitly evaluates the constraints; in many cases it is forced to construct all the states and it adopts a breath-first expansion style. Overall, BeTL works as an initial prototype, providing the basis for more efficient future developments.

6.1. Related Work

We are not aware of others who have explicitly studied constraints directly *in the logic itself*, such as those described in this paper, apart from ourselves in earlier work on constrained and *exactly one* extensions of PTL [18, 19, 20]. Firstly we consider other provers for propositional linear time temporal logic. We discuss two main approaches to theorem proving in PTL: tableau and resolution. Both are refutation based i.e. to prove a formula (φ) is valid it is negated and if the negation ($\neg\varphi$) is unsatisfiable then the original must be valid.

Tableau algorithms for PTL, for example see [33, 57] generally have a construction phase followed by a deletion phase. During the construction phase tableau rules are applied to formulae occurring at nodes in the structure which expand the structure and (mostly) simplify formulae. The deletion phase removes parts of the structure which could *never* be used to construct a model, for example where eventualities cannot be satisfied. Tableau algorithms for PTL have been implemented as part of both the Logics Workbench [39] and the Tableau Workbench [1].

The Logics Workbench [39] is a suite of logical tools for several propositional modal and temporal logics including PTL. The PTL module offers a number of functions to manipulate formulae including the *satisfiability* and *model* functions. The *satisfiability* function implements Janssen's tableau algorithm [40]. This is a two phase style algorithm as described above. The *model* function implements Schwendimann's One Pass Tableau [50] where the construction and deletion phases are combined.

The Tableau Workbench [1] is a framework for constructing tableau provers for arbitrary propositional logics. It allows users to specify their own tableau rules and provers based on these rules. It has several modules with pre-defined calculi for a number of modal and temporal logics including PTL. Implementations of both Wolper's and Schwendimann's tableau are also available from the Automated Reasoning Group at the Australian National University[2].

We have compared our prototype prover with LWB and TWB as they are tableau procedures and ours is tableau-like in that it constructs a structure and then deletes parts of this. Our

[2]`users.cecs.anu.edu.au/ rpg/PLTLProvers/`

construction method is different in that it requires formulae to be in a particular normal form, and in the worst case may have to build all the nodes in the construction which will be exponential in the number of unconstrained propositions.

Resolution approaches for PTL tend to first translate formulae into some normal form, termed temporal clauses, for example the one provided in Section 2. Then resolution rules are applied to formulae in the normal form that add new temporal clauses to the temporal clause set. The process terminates when either false is derived or no new temporal clauses can be derived. The temporal resolution algorithm from [28] has been implemented in the prover TRP++ [38]. The behaviour graph construction we use here (without the constraints) was proposed in [28] as part of the completeness proof. It was proposed as a way of dealing with constraints in [19]. A new resolution decision procedure based on labelled superposition is described in [53]. Additionally powerful tools for constructing automata from PTL formulae also exist [13, 32].

In recent years there has been an interest in the development and application of model checkers [12], for example, NuSMV [9], SPIN [37], Java Pathfinder [54] and PRISM [35]. Model checkers take a model of the system, usually some form of directed graph, and a formula, often in a temporal logic such as PTL or the branching time temporal logic CTL, and check that the formula is satisfied on the model. Whilst, in general, there is no way to explicitly input constraints as we have done here it may be possible to encode some types of constraints using the input language of the model. For example, in NuSMV enumerated types are allowed which correspond with *exactly one* of these propositions holding at any moment. For example the constraint

$$\mathcal{M}_i{}^{=1} \;=\; \{\; \mathit{playing}_i,\; \mathit{resting}_i,\; \mathit{injured}_i \;\}^{=1}$$

in Section 3.1 could be defined as an enumerated type, `mode`, for each player

```
mode:  {play, rest, injured};
```

Other constraints may be encoded using the INVAR (invariant) command. Formula in the scope of the INVAR command must not contain any temporal operators so this could be achieved using the $pos(C, k)$ and $neg(C, k)$ formulae defined in Section 2.4 for constraints of the form $C^{\propto k}$. Note that Model Checking and deduction (for example tableau and resolution calculi) aim to solve different problems. Deductive methods require a formula or set of formulae as input which are shown to be satisfiable or unsatisfiable. Model Checking requires a model and formula as input and show whether the model satisfies the formula.

Propositional satisfiability (SAT) [15] is the problem of deciding whether it is possible to assign true or false to the propositions in a formula of propositional logic that makes the formula evaluate to true. The Davis-Putnam-Logemann-Loveland (DPLL) procedure [14] mentioned in Section 5 has been used to implement SAT solvers but other methods have also been applied. SAT has been successfully applied to a number of areas and efficient implementations that can deal with thousands of variables have been developed. In this paper we use the DPLL algorithm when generating nodes in the behaviour graph that satisfy the sets of constraints and the right hand sides of a set of step clauses. We have implemented our own version of DPLL here as we want to return all the satisfying assignments rather than just returning just one of these. Another alternative would be to call a SAT solver which allows this. Efficient representation of cardinality constraints as a Boolean satisfiability problem has been extensively studied in the literature [2, 5, 52]. A relevant problem is the *weighted satisfiability problem*, where one checks if a Boolean circuit $\mathcal{C}$ produces 1 as output on some input values in which exactly k values are 1 and the others are 0, where k is a fixed *parameter* [29].

Cardinality type constraints also appear in Constraint Satisfaction Problems(CSP), see for example [49]. The paper [55] considers encodings of propositional logic into CSP. The additional constraints on propositions we use in this paper could be encoded similarly. To generate nodes in the behaviour graph an alternative approach would be to encode this as a CSP problem and call a CSP solver.

Mutually exclusive conditions (stemming e.g. from automata representations) and numbers from a fixed range can often be handled through efficient *translation* — consider, for example, logarithmic encoding or property-driven partitioning used in model checking [51] and SAT [6].

6.2. Future Work

Efficient Implementation. The BeTL implementation is a prototype. It was not intended to compete with other temporal provers but was developed as a first implementation against which to measure refined versions. For this reason we have not extended our experiments further to compare BeTL to one pass tableau systems such as [50] or resolution based systems such as TPR++ [38]. We are currently working on such improved versions of the basic tableau approach and expect these to be orders of magnitude faster than BeTL.

As our experimental results seem to show that using input constraints is beneficial, we are also developing a more standard tableau algorithm, similar to those described in [57, 50], but allowing *both* a temporal formula and constraints as input. Input formulae would not have to be transformed into a specific normal form and the usual tableau rules would be applied to the temporal formulae. In addition new tableau rules would be provided to deal with inferences between the propositional part at each moment in time and the tableau constraints. The advantages of this are not having to first translate any temporal formula into normal form; that it does not explicitly require us to construct sets of propositions that satisfy the constraints; and it gives us the opportunity to develop depth-first type algorithms (i.e. exploring one tableau branch at a time). See, for example [50].

Resolution for "Exactly One". One interesting variation on our work here involves restricting the logic to allow only "exactly one" sets (and as usual some unconstrained propositions), i.e. all the constraints are of the form $C_i^{=1}$. As a further restriction we can insist that the sets of propositions in each C_i are disjoint. We term the new logic TLX where $\mathrm{TLX}(C_1, \ldots, C_n) = \mathrm{TLC}(C_1^{=1}, \ldots, C_n^{=1})$ with the additional restriction of disjointness.

In [20] we devised a temporal resolution calculus for TLX, and established its completeness and complexity. Specifically, if a set of TLX clauses is unsatisfiable, then a contradiction will be deduced within time polynomial in $N_1 \times N_2 \times \cdots \times N_n \times 2^{\mathcal{A}}$ where N_1 is the size of C_1, N_2 is the size of C_2, etc, while $\mathcal{A}$ is the number of unconstrained propositions. TLX has a number of potential applications, and its relatively low complexity makes fast analysis feasible. Thus, part of our future work is to implement and evaluate such resolution calculi.

The application to other logics. We have studied constraints applied to a combination of temporal and epistemic logics in [21]. We could use similar techniques as developed here to apply to other temporal logics such as CTL [10].

Acknowledgements

The authors were partially supported by EPSRC grants GR/S63182 (Dixon, Konev) and EP/D052548 (Fisher). Nietiadi was supported by an ORS scholarship with additional support from the Department of Computer Science at the University of Liverpool. The authors would like to thank the reviewers for their constructive remarks.

References

[1] P. Abate, R. Gore, The tableaux work bench, in: TABLEAUX 03: Automated Reasoning with Analytic Tableaux and Related Methods, volume 2796 of *LNAI*, Springer, 2003, pp. 230–236.

[2] O. Bailleux, Y. Boufkhad, Efficient CNF encoding of boolean cardinality constraints, in: Ninth International Conference on Principles and Practice of Constraint Programming (CP), volume 2833 of *LNCS*, pp. 108–122.

[3] H. Barringer, Up and down the temporal way, The Computer Journal 30 (1987) 134–148.

[4] A. Behdenna, C. Dixon, M. Fisher, Deductive verification of simple foraging robotic behaviours, International Journal of Intelligent Computing and Cybernetics 2 (2009) 604–643.

[5] B. Benhamou, L. Sais, P. Siegel, Two proof procedures for a cardinality based language in propositional calculus, in: Proceedings Symposium on Theoretical Aspects of Computer Science (STACS), Springer, London, UK, 1994, pp. 71–82.

[6] L. Bordeaux, Y. Hamadi, L. Zhang, Propositional satisfiability and constraint programming: A comparative survey, ACM Computing Surveys 38 (2006).

[7] A. Cheng, J. Esparza, J. Palsberg, Complexity results for 1-safe nets, Theoretical Computer Science 147 (1995) 117–136.

[8] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, Z. Nevo, Incremental formal verification of hardware, in: P. Bjesse, A. Slobodová (Eds.), Proceedings of the 11th Conference on Formal Methods in Computer Aided Design, IEEE, 2011, pp. 125–143.

[9] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An opensource tool for symbolic model checking, in: Proceedings of International Conference on Computer-Aided Verification (CAV), pp. 359–364.

[10] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, Workshop, volume 131 of *LNCS*, Springer, Yorktown Heights, New York, USA, 1981, pp. 52–71.

[11] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (1986) 244–263.

[12] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 2000.

[13] M. Daniele, F. Giunchiglia, M.Y. Vardi, Improved automata generation for linear temporal logic, in: Proceedings of the 11th Conference on Computer Aided Verification (CAV), volume 1633 of *LNCS*, Springer, 1999, pp. 249–260.

[14] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, ACM Communications 5 (1962) 394–397.

[15] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.

[16] A. Degtyarev, M. Fisher, B. Konev, A simplified clausal resolution procedure for propositional linear-time temporal logic, in: Automated Reasoning with Analytic Tableaux and Related Methods, volume 2381 of *LNCS*, Springer, 2002, pp. 85–99.

[17] S. Demri, P. Schnoebelen, The complexity of propositional linear temporal logics in simple cases, Information and Computation 174 (2002) 84–103.

[18] C. Dixon, M. Fisher, B. Konev, Is there a future for deductive temporal verification?, in: Proceedings of the 13th International Symposium on Temporal Representation and Reasoning (TIME), IEEE Computer Society Press, 2006, pp. 11–18.

[19] C. Dixon, M. Fisher, B. Konev, Temporal logic with capacity constraints, in: Proceedings of the 6th International Symposium on Frontiers of Combining Systems (FroCoS), volume 4720 of *LNCS*, Springer, 2007, pp. 163–177.

[20] C. Dixon, M. Fisher, B. Konev, Tractable temporal reasoning, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), AAAI Press, 2007, pp. 318–323.

[21] C. Dixon, M. Fisher, B. Konev, Taming the complexity of temporal epistemic reasoning, in: Proceedings of the 7th International Symposium on Frontiers of Combining Systems (FroCoS), LNAI, 2009, pp. 198–213.

[22] J. Esparza, Decidability and complexity of Petri net problems - an introduction, in: Petri Nets, volume 1491 of *LNCS*, Springer, 1996, pp. 374–428.

[23] C.F. Fan, C.H. Sun, S. Yih, A dual language approach to software formal specifications and safety analysis, in: Proceedings of the 2002 International Computer Symposium (ICS), pp. 1126–1133.

[24] M. Fernández Gago, U. Hustadt, C. Dixon, M. Fisher, B. Konev, First-order verification in practice, Journal of Automated Reasoning 34 (2005) 295–321.

[25] M. Finger, M. Fisher, R. Owers, Metatem at work: Modelling reactive systems using executable temporal logic, in: Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE), Gordon and Breach, Edinburgh, U.K, 1993, pp. 209–218.

[26] M. Fisher, A resolution method for temporal logic, in: J. Myopoulos, R. Reiter (Eds.), Proceedings of IJCAI'91, Morgan Kaufman, 1991, pp. 99–104.

[27] M. Fisher, An Introduction to Practical Formal Methods Using Temporal Logic, Wiley, 2011.

[28] M. Fisher, C. Dixon, M. Peim, Clausal temporal resolution, ACM Transactions on Computational Logic 2 (2001) 12–56.

[29] J. Flum, M. Grohe, Parameterized Complexity Theory, Springer, 2006.

[30] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, The Temporal Analysis of Fairness, in: Proceedings of the 7th ACM Symposium on the Principles of Programming Languages (POPL), ACM Press, 1980, pp. 163–173.

[31] I. Gent, T. Walsh, The SAT phase transition, in: Proccedings of the 11th European Conference on Artificial Intelligence (ECAI), Wiley, 1994, pp. 105–109.

[32] D. Giannakopoulou, F. Lerda, From states to transitions: Improving translation of LTL formulae to Büchi automata, in: Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), volume 2529 of *LNCS*, Springer, 2002, pp. 308–326.

[33] G.D. Gough, Decision Procedures for Temporal Logic, Master's thesis, Department of Computer Science, University of Manchester, 1984. Also University of Manchester, Department of Computer Science, Technical Report UMCS-89-10-1.

[34] J. Handy, The Cache Memory Book, Academic Press, 1993.

[35] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 3920 of *LNCS*, Springer, 2006, pp. 441–444.

[36] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[37] G.J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.

[38] U. Hustadt, B. Konev, TRP++ 2.0: A temporal resolution prover, in: Automated Deduction—CADE-19, volume 2741 of *LNAI*, Springer, 2003, pp. 274–278.

[39] G. Jaeger, P. Balsiger, A. Heuerding, S. Schwendimann, M. Bianchi, K. Guggisberg, G. Janssen, W. Heinle, F. Achermann, A.D. Boroumand, P. Brambilla, I. Bucher, H. Zimmermann, LWB–The Logics Workbench 1.1, `www.lwb.unibe.ch`, 2002. University of Berne, Switzerland.

[40] G. Janssen, Logics for Digital Circuit Verification - Theory, Algorithms, and Applications, Ph.D. thesis, University of Eindhoven, 1999.

[41] N.D. Jones, L.H. Landweber, Y.E. Lien, Complexity of some problems in Petri nets, Theoretical Computer Science 4 (1977) 277–299.

[42] E. Kindler, Petri nets, situations, and automata, in: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets (ICATPN), volume 2360 of *LNCS*, Springer, 2002, pp. 217–236.

[43] E. Kindler, W. Reisig, H. Völzer, R. Walter, Petri net based verification of distributed algorithms: An example, Formal Aspects of Computing 9 (1997) 409–424.

[44] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison Wesley, 2003.

[45] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer, 1992.

[46] J. Nembrini, Minimalist Coherent Swarming of Wireless Networked Autonomous Mobile Robots, Ph.D. thesis, University of the West of England, 2005.

[47] W. Reif, F. Ortmeier, A. Thums, G. Schellhorn, Integrated formal methods for safety analysis and train systems, in: Building the Information Society, IFIP 18th World Computer Congress, Kluwer, 2004, pp. 637–642.

[48] W. Reisig, Elements of distributed algorithms: modeling and analysis with Petri nets, Springer, 1998.

[49] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson, 2010.

[50] S. Schwendimann, Aspects of Computational Logic, Ph.D. thesis, Universität Bern, Switzerland, 1998.

[51] R. Sebastiani, S. Tonetta, M. Vardi, Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking, in: Proceedings of the 17th International Conference on Computer Aided Verification (CAV), volume 3576 of *LNCS*, Springer, 2005, pp. 350–363.

[52] C. Sinz, Towards an optimal CNF encoding of boolean cardinality constraints, in: Proceedings of the 11th Intl. Conf. on Principles and Practice of Constraint Programming (CP), pp. 827–831.

[53] M. Suda, C. Weidenbach, Labelled superposition for PLTL, in: Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR), LNCS, Springer, 2012.

[54] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Automated Software Engineering 10 (2003) 203–232.

[55] T. Walsh, Sat v csp, in: R. Dechter (Ed.), Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, volume 1894, Springer, 2000, pp. 441–456.

[56] A. Winfield, J. Sa, M.C. Fernández-Gago, C.Dixon, M. Fisher, On formal specification of emergent behaviours in swarm robotic systems, International Journal of Advanced Robotic Systems 2 (2005) 363–370.

[57] P. Wolper, The tableau method for temporal logic: An overview, Logique et Analyse 110–111 (1985) 119–136.

[58] W.G. Wood, Temporal logic case study, in: Proceedings of the 1st International Workshop on Automatic Verification Methods for Finite State Systems, volume 407 of *LNCS*, Springer, 1989, pp. 257–263.