Temporal Logic with Capacity Constraints

Clare Dixon, Michael Fisher, and Boris Konev

Department of Computer Science, University of Liverpool, Liverpool, U.K.
{C.Dixon, M.Fisher, B.Konev}@csc.liv.ac.uk

Abstract. Often when modelling systems, physical constraints on the resources available are needed. For example, we might say that at most N processes can access a particular resource at any moment or exactly M participants are needed for an agreement. Such situations are concisely modelled where propositions are constrained such that at most N, or exactly M, can hold at any moment in time. This paper describes both the logical basis and a verification method for propositional linear time temporal logics which allow such constraints as input. The method incorporates ideas developed earlier for a resolution method for the temporal logic TLX and a tableaux-like procedure for PTL. The complexity of this procedure is discussed and case studies are examined. The logic itself represents a combination of standard temporal logic with classical constraints restricting the numbers of propositions that can be satisfied at any moment in time.

1 Introduction

Although temporal logic is widely used in the specification and verification of concurrent and reactive systems [21, 20] , there are cases where *full* temporal logic is *too* expressive. In particular, if we wish to describe the temporal properties of a restricted number of components, not all of which can occur at every moment in time, then the full temporal language forces us to describe the behaviour of all these components explicitly. In [8], it was shown that, simply by incorporating *"exactly one" constraints* into a propositional temporal logic, much better computational complexity could be achieved. Essentially, the basic set of propositions within the temporal logic was partitioned into *"exactly one" sets*. So

$$\mathsf{Props} \;=\; \mathsf{X}_1 \cup \mathsf{X}_2 \cup \ldots \cup \mathsf{X}_n$$

where each X_i is disjoint. Then the propositions within each "exactly one" set, for example

$$\mathsf{X}_1 \;=\; \{p_1, p_2, p_3, p_4\}$$

were implicitly constrained so that, at any moment in time, exactly one of p_1, p_2, p_3, or p_4 is satisfied. Not only did this allow the concise specification of examples such as the representation of automata, planning problems, and agent negotiation protocols, but also greatly reduced the complexity of the associated decision procedure [7, 8]. Essentially, this is because (as the name suggests) *exactly* one element of each "exactly one" set must be satisfied at every temporal state. So, in the above example, we can only have exactly one of p_1, p_2, p_3, or p_4 in any state; we can never have combinations of these propositions holding at the same moment or none of them holding.

Although being able to constrain the logic so that exactly *one* of a particular set of propositions is satisfied is useful, especially for representing finite automata, we have found this to be quite restrictive at times. So, in this paper, we will generalise the approach from [8] beyond simply "exactly one" sets. We will allow the specifier to constrain the logic so that *up to* k propositions, or *exactly* k propositions from some subset of propositions, are true at any moment in time. Thus, in this paper, rather than working with a temporal logic extended with "exactly one" sets, we instead use more flexible *constrained* sets, which provide more sophisticated restrictions on the capacity within the logic. Note that this approach involves reasoning *in the presence* of constraints rather than reasoning *about* them. That is, the resulting logic represents a combination of standard temporal logic with (fixed) constraints that restrict the numbers of propositions that can be satisfied at any moment in time.

This new approach is particularly useful for:

- 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 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($\mathcal{W}^{=j}, \mathcal{R}^{<k+1}$), where

$$\mathcal{W}^{=j} = \{work_1, \ldots, work_n\}$$
$$\mathcal{R}^{<k+1} = \{recharge_1, \ldots, recharge_n\}$$

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.

The above represents exactly the kind of logical base and case studies we consider in this paper.

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. Section 3 addresses its complexity. In Section 4 we show how to construct a structure representing the underlying models of formulae and relate satisfiability of the formula with a property of the structure. In Section 5 we provide examples and, in Section 6 we provide concluding remarks, incorporating both related and future work.

2 A Constrained Temporal Logic

The logic we consider is called "TLC", and its syntax and semantics essentially follow that of PTL [12], with models being isomorphic to the Natural Numbers, $\mathbb{N}$. The main novelty in TLC is that it is parameterised by (not necessarily disjoint) sets $\mathcal{P}_1^{r_1,d_1}$, $\mathcal{P}_2^{r_2,d_2}$,..., where $r_i \in \{=,<\}$ and $d_i \in \mathbb{N}$ and the formulae of TLC($\mathcal{P}_1^{r_1,d_1}, \mathcal{P}_2^{r_2,d_2}, \ldots$) are constructed under the restriction that, dependent on r_i *exactly* or *less than* d_i propositions from every set $\mathcal{P}_i^{r_i,d_i}$ are true in every state.

For example, consider TLC($\mathcal{P}^{<4}$, $\mathcal{Q}^{=2}$), where $\mathcal{P}^{<4} = \{a, b, c, d, e, f\}$, and $\mathcal{Q}^{=2} = \{x, y, z\}$. Then, at any moment in time, less than four of a, b, c, d, e, or f are true, and exactly two of x, y, or z are true.

Furthermore, we assume that there exists a set of propositions, $\mathcal{A}$, in addition to those defined by the parameters, and that these propositions are unconstrained as normal in PTL. The set $\mathcal{A}$ is disjoint with

$$\bigcup_i \mathcal{P}_i^{r_i,d_i}\,.$$

Thus, TLC with no parameters, i.e. TLC( ) is essentially a standard propositional, linear temporal logic, while TLC($\mathcal{P}^{=1}$,$\mathcal{Q}^{=1}$,$\mathcal{R}^{=1}$) is a temporal logic containing the sets of propositions $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$, where these sets are constrained by a standard "exactly one" operator as in [7].

2.1 TLC Syntax

The future-time temporal connectives that we use include $\Diamond$ (*sometime in the future*), $\Box$ (*always in the future*), $\bigcirc$ (*in the next moment in time*), $\mathcal{U}$ (*until*), and $\mathcal{W}$ (*unless*, or *weak until*). Formally, TLC($\mathcal{P}_1^{r_1,d_1}, \ldots \mathcal{P}_n^{r_n,d_n}$) formulae are constructed from the following elements:

- a set, $\mathsf{Props} = \mathcal{P}_1^{r_1,d_1} \cup \ldots \mathcal{P}_n^{r_n,d_n} \cup \mathcal{A}$ of propositional symbols;
- propositional connectives, $\mathbf{true}$, $\mathbf{false}$, $\neg$, $\vee$, $\wedge$, and $\Rightarrow$; and
- temporal connectives, $\bigcirc$, $\Diamond$, $\Box$, $\mathcal{U}$, and $\mathcal{W}$.

The set of well-formed formulae of TLC, denoted by WFF, is inductively defined as the smallest set satisfying the following.

- Any element of Props and $\mathbf{true}$ and $\mathbf{false}$ are in WFF.
- If A and B are in WFF then so are

$$\neg A \quad A \vee B \quad A \wedge B \quad A \Rightarrow B \quad \Diamond A \quad \Box A \quad A\mathcal{U}B \quad A\mathcal{W}B \quad \bigcirc A\,.$$

A *literal* is defined as either a proposition symbol or the negation of a proposition symbol.

2.2 TLC Semantics

A model for TLC formulae can be characterised as a sequence of *states* of the form:

$$\sigma = t_0, t_1, t_2, t_3, \ldots$$

where each state, t_i, is a set of proposition symbols, representing those propositions which are satisfied in the i^{th} moment in time. Note that every t_i should satisfy the constraints on propositions. For example, for TLC($\mathcal{Q}^{=2}$), every state t_i must contain exactly two propositions from the constraint set $\mathcal{Q}^{=2}$.

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

$$\begin{array}{ll}
(\sigma,i) \models \mathbf{true} & \\
(\sigma,i) \not\models \mathbf{false} & \\
(\sigma,i) \models p & \text{iff } p \in t_i \text{ where } p \in \mathsf{Props} \\
(\sigma,i) \models A \wedge B & \text{iff } (\sigma,i) \models A \text{ and } (\sigma,i) \models B \\
(\sigma,i) \models A \vee B & \text{iff } (\sigma,i) \models A \text{ or } (\sigma,i) \models B \\
(\sigma,i) \models A \Rightarrow B & \text{iff } (\sigma,i) \models \neg A \text{ or } (\sigma,i) \models B \\
(\sigma,i) \models \neg A & \text{iff } (\sigma,i) \not\models A \\
(\sigma,i) \models \bigcirc A & \text{iff } (\sigma,i+1) \models A \\
(\sigma,i) \models \Diamond A & \text{iff } \exists k \in \mathbb{N}.\ (k \geqslant i) \text{ and } (\sigma,k) \models A \\
(\sigma,i) \models \Box A & \text{iff } \forall j \in \mathbb{N}.\ \text{if } (j \geqslant i) \text{ then } (\sigma,j) \models A \\
(\sigma,i) \models A\mathcal{U}B & \text{iff } \exists k \in \mathbb{N}.\ k \geqslant i \text{ and } (\sigma,k) \models B \\
 & \quad \text{and } \forall j \in \mathbb{N},\ \text{if } i \leqslant j < k \text{ then } (\sigma,j) \models A \\
(\sigma,i) \models A\mathcal{W}B & \text{iff } (\sigma,i) \models A\mathcal{U}B \text{ or } (\sigma,i) \models \Box A
\end{array}$$

For any formula A, model σ, and state index $i \in \mathbb{N}$, then either $(\sigma, i) \models A$ holds or $(\sigma, i) \models A$ does not hold, denoted by $(\sigma, i) \not\models A$. If there is some σ such that $(\sigma, 0) \models A$, then A is said to be *satisfiable*. If $(\sigma, 0) \models A$ for all models, σ, then A is said to be *valid* and is written $\models A$. Note that formulae here are interpreted at t_0; this is an *anchored* definition of satisfiability and validity [9].

2.3 Normal Form

To assist in the definition of the normal form we introduce a further (nullary) connective '$\mathbf{start}$' that holds only at the beginning of time, i.e.,

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

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

Assume we have n sets of constrained propositions $\mathcal{P}_1^{r_1,d_1} = \{p_{11}, \ldots p_{1N_1}\}$, ..., $\mathcal{P}_n^{r_n,d_n} = \{p_{n1}, \ldots p_{nN_n}\}$ and a set of additional propositions $\mathcal{A} = \{a_1, \ldots a_{N_a}\}$. In the following, small Latin letters, k_i, l_j, m represent literals in the language Props. A normal form for TLC is of the form $\Box \bigwedge_i C_i$ where each C_i is an *initial*, *step*, or

sometime clause (respectively) as follows:

$$\begin{aligned} \mathbf{start} &\Rightarrow \bigvee_i l_i && \textit{(initial)} \\ \bigwedge_i k &\Rightarrow \bigcirc \bigvee_j l_j && \textit{(step)} \\ \mathbf{true} &\Rightarrow \Diamond m && \textit{(sometime)} \end{aligned}$$

Theorem 1. *[11] Any TLC formula can be transformed into an equi-satisfiable TLC formula in the normal form with at most a linear increase in the size of the problem.*

Transformation into the normal form may introduce new (unconstrained) propositions; note, however, that 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 [13].

3 Complexity of TLC

We now prove the upper complexity bound on satisfiability of TLC by an explicit construction of a directed graph known as a *behaviour graph*. The notion of a behaviour graph for a set of clauses was introduced in [11]. 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 clauses. Satisfiability of TLC formulae is equivalent to a property of the graph; in what follows, we estimate the size of the graph and time needed both for its construction and for checking the property.

Given a formula φ in the normal form over a set of (both constrained and unconstrained) propositional symbols Props, we construct a finite directed graph G as follows. The nodes of G are interpretations of Props, satisfying the required constraints.

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 , $\bigwedge_i k \Rightarrow \bigcirc \bigvee_j l_j$, if $I \models \bigwedge_i k$ then $I' \models \bigvee_j l_j$.

A node, I, is designated an initial node of G if $I \models \bigvee_i l_i$ for every initial clause $\mathbf{start} \Rightarrow \bigvee_i l_i$ of the given temporal formula.

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 such that

a) I does not have a successor; or
b) for some eventuality clause $\mathbf{true} \Rightarrow \Diamond m$ within φ, there is no path from I to a node J where m is true, that is, $J \models m$.

The following theorem can be obtained by an adaptation of results in [11, 5] to the terminology of this paper.

Theorem 2. *[11, 5] A TLC formula in the normal form φ is satisfied if, and only if, its reduced behaviour graph is non-empty.*

The link between the satisfiability of TLC formulae and properties of the behaviour graph allows us to prove the complexity bound for our logic.

Theorem 3. *Satisfiability of a $TLC(\mathcal{P}_1^{r_1,d_1}, \ldots \mathcal{P}_n^{r_n,d_n})$ formula φ can be decided in time*

$$O\left(|\varphi| \times \left(|\mathcal{P}_1^{r_1,d_1}|^{d_1} \times \ldots \times |\mathcal{P}_n^{r_n,d_n}|^{d_n} \times 2^{|A|}\right)^2\right)$$

where $|\varphi|$ is the length of φ, $|\mathcal{P}_i^{r_i,d_i}|$ is the size of the set $\mathcal{P}_i^{r_i,d_i}$ of constrained propositions, and $|A|$ is the size of the set A of non-constrained propositions.

Proof. There exist $O(|\mathcal{P}^{r_1,d_1}|^{d_1} \times \ldots \times |\mathcal{P}_n^{r_n,d_n}|^{d_n} \times 2^{|A|})$ different interpretations of propositions from Props; moreover, they can all be enumerated in time $O(|\mathcal{P}^{r_1,d_1}|^{d_1} \times \ldots \times |\mathcal{P}_n^{r_n,d_n}|^{d_n} \times 2^{|A|})$. Therefore, one can explicitly build the reduced behaviour graph H_R in time $O(|\varphi| \times (|\mathcal{P}^{r_1,d_1}|^{d_1} \times \ldots \times |\mathcal{P}_n^{r_n,d_n}|^{d_n} \times 2^{|A|})^2)$.

Corollary 1 *If the number n of sets of constrained propositions, the number d_i of propositions from the set $\mathcal{P}^{r_i,d_i}$ that can be true at any time, and the size $|A|$ of the set of non-constrained propositions is fixed, satisfiability of $TLC(\mathcal{P}_1^{r_1,d_1}, \ldots \mathcal{P}_n^{r_n,d_n})$ formulae can be decided in polynomial time.*

4 Checking Satisfiability

Based on the proof of complexity of TLC given in Section 3, one can provide an algorithm checking the satisfiability of TLC formulae as follows.

4.1 Incremental Algorithm

A straightforward approach is to construct the graph G representing all possible interpretations of Props, 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, do excess work. In this section we present an incremental, tableaux-like algorithm, which avoids building these unnecessary nodes.

Let $\mathsf{Assignments}(\varphi, \mathsf{cons})$ be a procedure, which, when given a formula, φ, and a set of constraints on variables, cons, returns the set of *all* interpretations within the language Props that both satisfy the conditions cons and make φ true. Clearly, $\mathsf{Assignments}(\varphi, \{\mathcal{P}_1^{r_1,d_1}, \ldots, \mathcal{P}_n^{r_n,d_n}\})$ can be computed deterministically in time $O(int)$ where $int = (|\mathcal{P}^{r_1,d_1}|^{d_1} \times \ldots \times |\mathcal{P}_n^{r_n,d_n}|^{d_n} \times 2^{|A|})$ returning at most $O(int)$ interpretations for any φ.

Example. If $\mathsf{Props} = \{p, q, r, s\}$ then $\mathsf{Assignments}(p \vee q, \{\{p, q, r, s\}^{=1}\})$ will return two interpretations: $\{p, \neg q, \neg r, \neg s\}$ and $\{\neg p, q, \neg r, \neg s\}$; whereas $\mathsf{Assignments}(p \vee q, \{\{p, q\}^{=1}, \{q, r, s\}^{=2}\})$ will return three: $\{p, \neg q, r, s\}$, $\{\neg p, q, r, \neg s\}$, and $\{\neg p, q, \neg r, s\}$.

Now, we use $\mathsf{Assignments}(\varphi, \mathsf{cons})$ to construct nodes of the behaviour graph H for a formula φ incrementally. Nodes of H can be *marked* or *unmarked*. A node is marked if all its successors are already represented in H, otherwise, it is unmarked. The incremental algorithm is given in detail in Fig. 1. Note that if the set of clauses contains no initial clauses, then the formula ψ in line 1 of the algorithm is $\mathbf{true}$, and if the conjunction in line 7 is empty then χ is $\mathbf{true}$. After the behaviour graph is constructed, we compute the reduced behaviour graph in time quadratic in the size of the behaviour graph.

1: Let $\psi = \bigwedge\{C_j \mid \mathbf{start} \Rightarrow C_j$ is an initial clause$\}$
2: **for** all I in $\mathsf{Assignments}(\psi, \mathsf{cons})$ **do**
3: Add an unmarked node I to H
4: **end for**
5: **while** Not all nodes in H are marked **do**
6: Pick an unmarked node I and mark I
7: Let $\chi = \bigwedge\{D_k \mid C_k \Rightarrow \bigcirc D_k$ is a step clause, $I \models C_k\}$
8: **for** all J in $\mathsf{Assignments}(\chi, \mathsf{cons})$ **do**
9: **if** J is not already in H **then**
10: Add an unmarked node J to H
11: **end if**
12: Add an edge (I, J) to H
13: **end for**
14: **end while**

Fig. 1. Incremental behaviour graph construction algorithm.

The following theorem follows from the Incremental Algorithm given in Figure 1.

Theorem 4. *Given a TLC formula ϕ, the incremental procedure terminates and builds the behaviour graph H of ϕ.*

5 Case Studies

In this section we will consider case studies and show how they can be specified and verified in TLC.

5.1 Multiprocessor Job-shop Scheduling

The first problem we consider is a generalisation of the classic job-shop scheduling problem, called the *Multiprocessor Job-shop Scheduling* (MJS) problem [4, 2]. Here,

a set of jobs (j_1, j_2, ..., j_n) have to be processed on a set of machines running in parallel. In general, each job might require several processor steps to complete but, to begin with we will assume each job is completed after only one step; see Fig. 2. While in these settings deciding if a schedule exists at all is comparatively easy, once we add constraints such that one job *must* be run before another, deciding the existence of a schedule becomes exponentially hard [2].

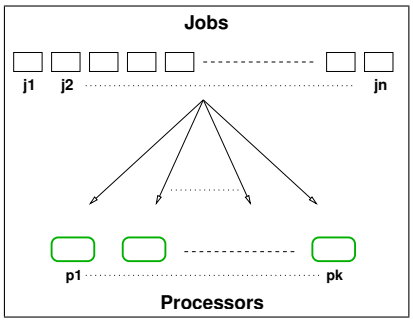


Fig. 2. Basic Multiprocessor Job Scheduling.

In what follows, we will begin with a basic specification (in TLC) corresponding to the easy form of the problem, but will then add formulae constraining particular jobs, thus giving a harder form of problem.

Basic Specification We will now model this in TLC. Assume that we have just one constrained set

$$\{run_1, run_2, \ldots, run_n\}^{=k}.$$

Since this constrained set is of dimensionality k, then we know that at any moment in time, exactly k propositions from run_1, run_2, ..., run_n are satisfied. (N.B., later we will relax this constraint and allow $< (k+1)$ propositions to be satisfied at any moment.)

Now, we define other propositions using the following clauses in the normal form (for every $1 \leq i \leq n$):

$$\begin{aligned} \mathbf{start} &\Rightarrow \neg hasrun_i \\ (\neg hasrun_i \wedge \neg run_i) &\Rightarrow \bigcirc \neg hasrun_i \\ run_i &\Rightarrow \bigcirc hasrun_i \\ hasrun_i &\Rightarrow \bigcirc hasrun_i \end{aligned}$$

N.B., the last two clauses effectively define $run_i \Rightarrow \bigcirc \Box hasrun_i$.

So, we have defined $hasrun_i$ to be true if j_i has been run in the past.

Now, to allow us to establish some simple properties, we will also constrain each job to only run once. So, we add (though we must translate to the normal form), for each $1 \leq i \leq n$

$$run_i \Rightarrow \bigcirc \Box \neg run_i \, .$$

Synthesising a Schedule Let us term the basic system, comprising the above clauses, as 'φ'. Now, we can simply ask whether

$$\mathbf{start} \Rightarrow (\varphi \Rightarrow \Diamond \bigwedge_{i=1}^{n} hasrun_i)$$

is satisfiable or not. In other words, is there a point in the future by which time all jobs $j_1, \ldots, j_n$ have run?

If this is satisfiable, then there *is* a way to run the jobs such that exactly k run at every moment in time. If the above is not satisfiable, then it is not possible to find such a moment.

However, the strict k bound on the constrained set is a little restrictive, especially when n is *not* a multiple of k. So, we can reformulate the problem with a constrained set $\{run_1, run_2, \ldots, run_n\}^{<(k+1)}$, allowing *at most* k jobs to be run at any moment in time. Thus, checking that

$$\mathbf{start} \Rightarrow (\varphi \Rightarrow \Diamond \bigwedge_{i=1}^{n} hasrun_i)$$

is satisfiable tells us whether there is a way to schedule the jobs successfully on k processors. Note that, since we build a behaviour graph for this, then if the above is satisfiable we can extract a satisfying linear path from the graph. This path corresponds to the schedule for achieving the required job runs.

Now, once we know that $\Diamond \bigwedge_{i=1}^{n} hasrun_i$ is satisfied, we can go further. We can check

$$\mathbf{start} \Rightarrow (\varphi \Rightarrow \bigcirc^m \bigwedge_{i=1}^{n} hasrun_i) \, .$$

If this is satisfiable, then decrease m and try again; if it is unsatisfiable, increase m and try again. In this way, we find the minimum value of m such that

$$\bigcirc^m \bigwedge_{i=1}^{n} hasrun_i$$

is satisfied. Thus, there is no way to schedule the jobs any faster than in m steps — in this sense, we generate an *optimal* schedule through carrying out such satisfiability checks. Note that finding an optimal schedule for the MJS problem is typically an NP-hard problem [4].

Refined Specification Now let us constrain this scenario still further. It is only to be expected that there will be dependencies between some of the jobs. We can also specify these simply in TLC. In order to show this, below are some examples.

- *job y must run immediately after job x*: $run_x \Rightarrow \bigcirc run_y$
- *job b must not run before job a has run*: $run_a \Rightarrow \bigcirc \diamondsuit run_b$
- *jobs p and q must only run at exactly the same time*: $\Box(run_p \Leftrightarrow run_q)$
- *jobs f and g must never run simultaneously* $\Box(\neg run_f \vee \neg run_g)$
- *and so on ...*

In this way we can specify job interdependencies and, via satisfiability checking, can extract the (optimal) schedule (if there is one)[1].

Note. It is obviously possible to generate a set of job interdependencies such that the specification is unsatisfiable, for example

$$\Box(run_1 \Leftrightarrow run_2) \wedge \Box(run_2 \Leftrightarrow run_3) \wedge \ldots \wedge \Box(run_{h-1} \Leftrightarrow run_h)$$

where $h > k$. Thus, it is natural to check satisfiability of the basic specification, φ, before checking $\diamondsuit \bigwedge_{i=1}^{n} \Box\neg run_i$, etc.

5.2 Robots

Consider the robots example outlined earlier. Let us assume there are 5 robots and at any moment 3 must work and up to 2 may recharge. Further, each robot must do exactly one of *work*, *rest* or *recharge* at any moment. In the terminology we have provided, the problem is defined as TLC($\mathcal{W}^{=3}, \mathcal{R}^{<3}, \mathcal{S}_1^{=1}, \mathcal{S}_2^{=1}, \mathcal{S}_3^{=1}, \mathcal{S}_4^{=1}, \mathcal{S}_5^{=1}$), where

$$\begin{aligned} \mathcal{W}^{=j} &= \{work_1, \ldots, work_5\} \\ \mathcal{R}^{<3} &= \{recharge_1, \ldots, recharge_5\} \\ \mathcal{S}_i^{=1} &= \{work_i, rest_i, recharge_i\} \end{aligned}$$

We assume that each robot has a specification relating to when it works, rests and recharges. For example, we could assume that each robot has the same specification and that after working for one time unit it must recharge for one time unit.

$$\Box(work_i \Rightarrow \bigcirc recharge_i)$$

We assume initially that robots 1,2 and 3 are working, i.e.

$$\begin{aligned} \mathbf{start} &\Rightarrow work_1 \\ \mathbf{start} &\Rightarrow work_2 \\ \mathbf{start} &\Rightarrow work_3 \end{aligned}$$

[1] We could also have specified jobs of varying durations (rather than just one step) in TLC. However, this would have taken more space than was available.

Informally we can see that this specification plus the constraints are unsatisfiable. This is because at any moment there must be exactly three robots working. The specification will then require that at the following moment all the three robots that were working in the previous moment are now recharging which will contradict the constraint relating to recharging.

Applying the Incremental Algorithm where $\psi = work_1 \wedge work_2 \wedge work_3$ and

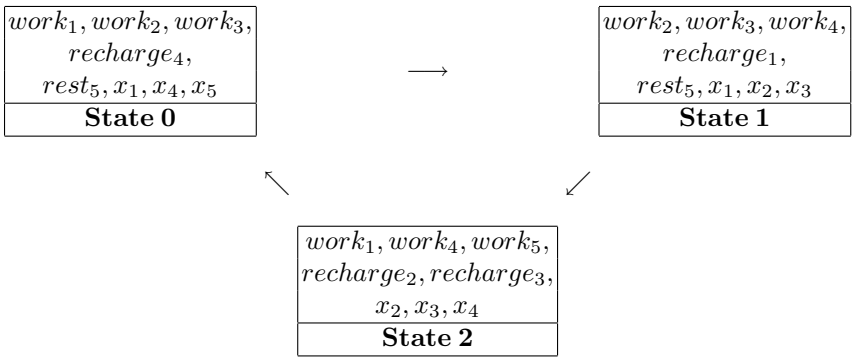
$$\mathsf{Assignments}(\psi, \mathsf{cons}) = \left\{ \begin{array}{l} \{work_1, work_2, work_3, recharge_4, recharge_5\}, \\ \{work_1, work_2, work_3, rest_4, rest_5\}, \\ \{work_1, work_2, work_3, recharge_4, rest_5\}, \\ \{work_1, work_2, work_3, rest_4, recharge_5\} \end{array} \right\}$$

we add an unmarked node for each assignment to the behaviour graph. Consider the first of these assignments and call its related node I. Extending the structure we construct χ from the robot specification. Here $\chi = recharge_1 \wedge recharge_2 \wedge recharge_3$. Now when we try construct $\mathsf{Assignments}(\chi, \mathsf{cons})$ we obtain an empty set as χ and the constraints cannot be satisfied together. Reasoning is similar from the other unmarked nodes, hence the reduced behaviour graph is empty and the specification plus constraints must be unsatisfiable.

Next we loosen the robot specifications to say that if a robot has worked for two moments in time it must recharge in the next moment. To specify this we need an additional proposition for each robot, x_i which holds at the moment after $work_i$ holds. Informally, if x_i is true then either we are at the start of time, or the i-th robot has worked in the previous moment.

$$\Box(work_i \Rightarrow \bigcirc x_i)$$
$$\Box(work_i \wedge x_i \Rightarrow \bigcirc recharge_i)$$

Again we assume that robots 1, 2 and 3 must work initially. Now our behaviour graph construction has an extra five propositions. The specification is now satisfiable, a sample model being the following.



Further, we may want to strengthen the specification to ensure that each robot gets a chance to work infinitely often.

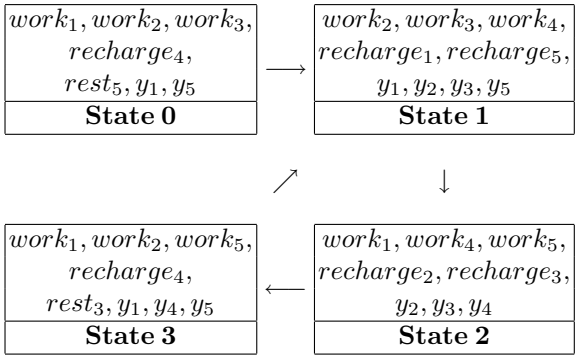
$$\mathbf{true} \Rightarrow \Diamond work_i$$

We can observe that this is satisfied in the above model.

Finally observing the model suggested above we can see that robot 5 never recharges as it just works intermittently. To avoid such a situation we may want to specify that the robot needs to recharge after working for two moments in time since last recharging even if these moments are not immediately one after the other. Again informally, if y_i is true then either we are at start of time or the i-th robot has worked one time unit since the last recharge.

$$\begin{gathered}\Box(work_i \Rightarrow \bigcirc y_i)\\ \Box(work_i \wedge y_i \Rightarrow \bigcirc recharge_i)\\ \Box(rest_i \wedge y_i \Rightarrow \bigcirc y_i)\\ \Box(recharge_i \Rightarrow \bigcirc \neg y_i)\end{gathered}$$

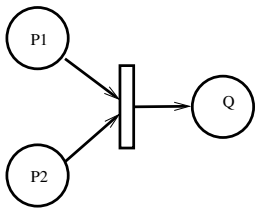
Now extending the above model with suitable y_i propositions and removing the x_i propositions won't satisfy the new specification and constraints as robot 5 works infinitely often but never gets to recharge. However we can easily specify a model which does satisfy the new specification and constraints.



5.3 Petri Nets

Consider now 1-safe Petri nets (see for example [10]), which are used to model systems with limited resources. In 1-safe nets, every place may contain at most one token. This restriction allows us to represent 1-safe Petri nets in propositional temporal logic. Encoding places with propositions (proposition p_i is true if, and only if, a token is at place P_i), given a 1-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 $\mathcal{N}$.

For example, the following transition



can be represented as

$$\Box(p_1 \wedge p_2 \Rightarrow \bigcirc(q \wedge \neg p1 \wedge \neg p2))$$

i.e. the transition fires if both P_1 and P_2 contain a token, plus suitable frame axioms to prevent tokens from arbitrarily appearing or disappearing. Similarly, reachability in these nets, for example the reachability of the state P_F corresponds to the satisfiability of $\Diamond p_F$ from an initial state. Since the reachability problem (as well as many other interesting problems) for 1-safe nets is PSPACE-complete [10], such translation is optimal.

We can then use capacity 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 [18, 19]. Note that imposing such extra restrictions actually makes the complexity of reasoning lower.

6 Concluding Remarks

In this paper we have introduced TLC, a propositional temporal logic that allows the specifier to put powerful additional constraints on how many propositions 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 propositions, for particular subsets of propositions, at each moment in time. This is not only an interesting extension of PTL, but can potentially be decided in polynomial, rather than exponential time. This improved complexity makes TLC a strong candidate for practical verification based upon temporal satisfiability.

We provide a graph construction algorithm to check satisfiability by enumerating only the reachable nodes that satisfy the required constraints. The definition of resolution rules incorporating these constraints appear complex and non-trivial. Similarly, the adaption of tableau algorithms for PTL [24] to this constrained situation lead, in some cases, to the generation of exponentially many successors to nodes in the tableau.

There is little related work in this area. Refinements of PTL have been considered, particularly in relation to model checking, by Demri and Schnoebelen [6]. Mutually exclusive conditions (stemming e.g. from automata representation) 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 [23] and SAT [1] — however, 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 just on XOR extensions of PTL [7, 8].

Our explicit graph construction has a similar flavour to that of tableau algorithms developed for PTL [24, 15] which attempt to explicitly construct a model for formulae. Here the expansion rules focus on formulae in the normal form rather than any well-formed formula. Implementations of the tableau procedures [22, 17] are available within the logics workbench [16], and powerful tools for constructing automata from PTL formulae now exist [3, 14].

Future Work. 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 propositions, 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.

Furthermore, the incremental algorithm in Fig. 1 can potentially be modified to allow for *dynamic* changes of the set of constraints. This allows us to accommodate constraints *into* the language as a logical connective. Let $\oplus^{=s}$ denote a logical operator of flexible arity, which states that exactly s of its arguments is true, while $\oplus^{<d}$ states that fewer than d of its arguments can be true. Expressions of the form

$$\bigwedge_i k \Rightarrow \bigcirc \oplus_j^{=d} l_j$$

are called $\oplus^{=d}$*-step clauses* ($\oplus^{<d}$-step clauses are defined similarly). If in the algorithm in Fig. 1, lines 6–12, $I \models \bigwedge_i k$, we look for interpretations, which make χ true and satisfy *both* constraints, cons, and the right-hand side of the of $\oplus^{=d}$ step clause. A more elaborate language even allows us to dynamically *disallow* existing constraints as well as introduce new ones.

The development of an implementation of the algorithms discussed in this paper together with practical verification case studies, form the basis for our future work. Further, the extension to dynamic constraints and a deeper comparison to related methods are future work.

Acknowledgements The authors were partially supported by EPSRC grants GR/S63182 (Dixon, Konev) and EP/D052548 (Fisher), and would also like to thank both Bo Chen and Leslie Goldberg for their input on complexity issues.

References

1. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Computing Surveys*, 38(4), 2006.
2. B. Chen. Parallel Scheduling for Early Completion. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 9. Chapman & Hall/CRC Press, 2004.
3. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proc. Eleventh International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 1999.
4. S. Dauzére-Pérès and J. Paulli. An Integrated Approach for Modeling and Solving the General Multiprocessor Job-shop Scheduling Problem using Tabu Search. *Annals of Operations Research*, 70:281–306, 1997.
5. A. Degtyarev, M. Fisher, and B. Konev. A Simplified Clausal Resolution Procedure for Propositional Linear-Time Temporal Logic. In *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 2381 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2002.

6. S. Demri and P. Schnoebelen. The Complexity of Propositional Linear Temporal Logic in Simple Cases. *Information and Computation*, 174(1):84–103, 2002.
7. C. Dixon, M. Fisher, and B. Konev. Is There a Future for Deductive Temporal Verification? In *Proc. Thirteenth International Symposium on Temporal Representation and Reasoning (TIME)*. IEEE Computer Society Press, June 2006.
8. C. Dixon, M. Fisher, and B. Konev. Tractable Temporal Reasoning. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 2007.
9. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.
10. J. Esparza. Decidability and Complexity of Petri Net Problems - An Introduction. In *Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996.
11. M. Fisher, C. Dixon, and M. Peim. Clausal Temporal Resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, Jan. 2001.
12. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The Temporal Analysis of Fairness. In *Proc. Seventh ACM Symposium on the Principles of Programming Languages (POPL)*, pages 163–173, January 1980.
13. M.-C. F. Gago, U. Hustadt, C. Dixon, M. Fisher, and B. Konev. First-Order Temporal Verification in Practice. *Journal of Automated Reasoning*, 34(3):295–321, April 2005.
14. D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proc. Twenty Second IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326. Springer, November 2002.
15. G. D. Gough. Decision Procedures for Temporal Logic. Master's thesis, Department of Computer Science, University of Manchester, October 1984. Also University of Manchester, Department of Computer Science, Technical Report UMCS-89-10-1.
16. 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, and H. Zimmermann. LWB–The Logics Workbench 1.1. `http://www.lwb.unibe.ch`, 2002. University of Berne, Switzerland.
17. G. Janssen. *Logics for Digital Circuit Verification: Theory, Algorithms, and Applications*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1999.
18. E. Kindler. Petri Nets, Situations, and Automata. In *Proc. 23rd International Conference on Applications and Theory of Petri Nets (ICATPN)*, volume 2360 of *Lecture Notes in Computer Science*, pages 217–236. Springer, 2002.
19. E. Kindler, W. Reisig, H. Völzer, and R. Walter. Petri Net Based Verification of Distributed Algorithms: An Example. *Formal Aspects of Computing*, 9(4):409–424, 1997.
20. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley Professional, 2003.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
22. S. Schwendimann. *Aspects of Computational Logic*. PhD thesis, University of Bern, Switzerland, 1998.
23. R. Sebastiani, S. Tonetta, and M. Y. Vardi. Symbolic Systems, Explicit Properties: On Hybrid Approaches for LTL Symbolic Model Checking. In *Proc. Seventeenth International Conference on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2005.
24. P. Wolper. The Tableau Method for Temporal Logic: An Overview. *Logique et Analyse*, 110–111:119–136, June-Sept 1985.