Bounded Model Checking for Unbounded Client-Server Systems

Ramchandra Phawade[1], Tephilla Prince[1], and S. Sheerazuddin[2]

[1] Indian Institute of Technology Dharwad, India
prb@iitdh.ac.in, tephilla.prince.18@iitdh.ac.in
[2] National Institute of Technology Calicut, India
sheeraz@nitc.ac.in

Abstract. Bounded model checking (BMC) is an efficient formal verification technique which allows for desired properties of a software system to be checked on bounded runs of an abstract model of the system. The properties are frequently described in some temporal logic and the system is modeled as a state transition system. In this paper we propose a novel counting logic, $\mathcal{L}_C$, to describe the temporal properties of client-server systems with an unbounded number of clients. We also propose two dimensional bounded model checking ($2D$-BMC) strategy that uses two distinguishable parameters, one for execution steps and another for the number of tokens in the net representing a client-server system, and these two evolve separately, which is different from the standard BMC techniques in the Petri Nets formalism. This $2D$-BMC strategy is implemented in a tool called DCModelChecker which leverages the $2D$-BMC technique with a state-of-the-art satisfiability modulo theories (SMT) solver Z3. The system is given as a Petri Net and properties specified using $\mathcal{L}_C$ are encoded into formulas that are checked by the solver. Our tool can also work on industrial benchmarks from the Model Checking Contest (MCC). We report on these experiments to illustrate the applicability of the $2D$-BMC strategy.

Keywords: Bounded Model Checking · SAT solvers · Petri Nets · Counting Logics · Temporal Logics

1 Introduction

Model checking [3] is a formal verification technique that allows for desired behavioral properties of a given system to be verified based on a suitable model of the system through systematic inspection of all states of the model. The major challenge of model checking is that the state space of systems might be infinite. In bounded model checking, this challenge is overcome by assuming a predetermined bound on the runs, and all possible paths are explored. This technique uses a single parameter for execution steps, to unfold the system behaviour. Bounded Model Checking [5,10,4] is widely used in the industry by restricting

the model checking problem to a bounded problem and verifying properties on bounded runs of the system.

We consider the case of a client-server system with an unbounded number of agents, which we model as unbounded Petri Nets. In this work, we propose a novel extension to standard bounded model checking technique, the two dimensional bounded model checking ($2D$-BMC) strategy for unbounded client-server systems, that uses two distinguishable parameters, one for execution steps and another for the number of tokens in the net and these two evolve separately, which is different from the standard BMC techniques for Petri Nets.

Typically, in standard BMC, temporal logic is used to specify the properties of the system. In this setting of unbounded client-server systems, we would also like to express additional counting properties of the client and the server. We propose a novel counting logic with temporal operators, $\mathcal{L}_C$.

To illustrate the usefulness of the newly introduced $2D$-BMC strategy and counting logic $\mathcal{L}_C$, we implemented them in our new tool DCModelChecker. [3] BMC for Petri Nets while expressing properties using linear temporal logic while leveraging the power of SMT Solvers remains unexplored. DCModelChecker attempts to bridge this gap. We have considered benchmark models from the Model Checking Contest [17], and translated a subset of the properties into $\mathcal{L}_C$ and have obtained the satisfaction or counterexample trace in each case. Details of the tool and experiments are discussed in Section 5.1 and Section 6 respectively.

This paper makes the following contributions:

- We introduce a novel technique - $2D$-BMC to verify the properties of the system.
- We introduce a counting logic language, $\mathcal{L}_C$ for describing the counting properties of the system as well as temporal properties of the system.
- We build a tool, DCModelChecker, that uses $2D$-BMC and the designed counting logic language for verification.
- We perform an experimental evaluation of our tool, with academic and industrial models from the Model Checking Contest (MCC) [17], and compare the results against the state of the art tool ITS-Tools [26]. To the best of our knowledge, ours is the only tool that can perform $2D$-BMC and provide the counterexample as well.

Organization of our paper: We begin with the preliminaries on Petri Nets and their encoding in the next section. We introduce counting logic $\mathcal{L}_C$ in Section 3. In Section 4, we discuss the novel $2D$-BMC strategy and give its encoding and demonstrate the unfolding of formulas in this technique. The rest of the paper discusses the details of the tool DCModelChecker, elaborating on its architecture in Section 5.1, and the workflow of the tool in Section 5.2. Experiments using DCModelChecker are in Section 6, followed by the related work in Section 7, and future work in Section 8.

[3] https://doi.org/10.6084/m9.figshare.19611477.v3

2 Preliminaries

The bounded model checking problem can be described as follows. Given a system S and a specification or property α on bounded runs of the system, decide whether system S satisfies specification α. This consists of checking that all runs of S constitute models for α. It suffices to show that no run of S is a model for $\neg\alpha$, which is the same as checking that the intersection of the language accepted by S and the language defined by $\neg\alpha$ is empty.

We illustrate how client-server systems with unbounded agents can be mathematically encoded as nets in Section 2.1 and in Section 3 we explore a suitable logic to express its properties.

It is to be noted that theoretically, there may be an *unbounded* number of clients in the unbounded client-server system at any instant. Practically, factors such as environment limitations and hardware design may limit the number of clients that are present in an instant.

2.1 Petri Nets

A Petri Net structure is a tuple $N = (P, T, F, W)$ where P is a finite set of places, T is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation and $W : F \to \aleph_0$ is a weight function, where $\aleph_0$ is the set of non-negative integers. For any place (resp. transition) z of the set $P \cup T$, the set $\{x \mid (x, z) \in F\}$ is called *pre-transitions* (resp. *pre-places*) of z, and the set $\{x \mid (z, x) \in F\}$ is called *post-transitions* (resp. *post-places*) of z.

A marking M of a Petri Net is a fuction $M : P \to \aleph_0$. A Petri Net (system) is a tuple $PN = (N, M_0)$ where N is a Petri Net structure and M_0 is an initial marking. A transition t is *enabled* at marking M, if for each pre-place p of t we have $M(p) \geq W(p, t)$. A new marking is obtained when an enabled transition is *fired*, and is obtained by removing $W(p, t)$ tokens from each pre-place p of t, and adding $W(t, p)$ tokens to each post-place p of t, leaving tokens in the remaining places as it is.

Encoding of Petri Nets For the purpose of BMC we need to first encode a Petri Net system in propositional logic. A similar encoding can be found in [1,2]. Let $N = (P, T, F, W)$ be a Petri Net structure. Let $T = \{t_1, \ldots, t_n\}$ and $P = \{p_0, \ldots, p_l\}$. The variables used in the encoding are as follows. For each transition $t_i \in T$, there is a copy of the variable that is used in the encoding. For each place $p_i \in P$, we have two variables p_{x_i} and p_{y_i} in the encoding – p_{x_i} denotes the number of tokens in p_i before firing of some transition and p_{y_i} denotes the number of tokens in the same place after the firing. We have a vector of variables M that keep track of the marking. The propositional encoding of N is defined by the formula $\mathcal{T} = \mathcal{T}_{enabled} \wedge \mathcal{T}_{firability} \wedge \mathcal{T}_{next}$, where:

- the formula $\mathcal{T}_{enabled}$ states that more than one transition can be enabled at a time. i.e, it is an **or** over the preconditions of all enabled transitions and is given by $\mathcal{T}_{enabled} = pre_{t0} \vee pre_{t1} \vee \cdots \vee pre_{t_n}$;

- the formula $\mathcal{T}_{next}$ gives us the next transition that will be fired, and is expressed as an **or** expression over each transition **and** its postcondition **and** all other transitions and postconditions are negated.
  $\mathcal{T}_{next} =$
  $(t_0 \wedge \neg t_1 \wedge \cdots \wedge \neg t_n \wedge post_{t_0} \wedge \neg post_{t_1} \wedge \cdots \wedge \neg post_{t_n}) \vee$
  $(\neg t_0 \wedge t_1 \wedge \cdots \wedge \neg t_n \wedge \neg post_{t_0} \wedge post_{t_1} \wedge \cdots \wedge \neg post_{t_n}) \vee \cdots \vee$
  $(\neg t_0 \wedge \neg t_1 \wedge \cdots \wedge t_n \wedge \neg post_{t_0} \wedge \neg post_{t_1} \wedge \cdots \wedge post_{t_n})$;
- the formula $\mathcal{T}_{firability}$ relates the pre condition of a transition with its distinct postcondition and is given as
  $\mathcal{T}_{firability} = (post_{t_0} \rightarrow pre_{t_0}) \wedge (post_{t_1} \rightarrow pre_{t_1}) \wedge \cdots \wedge (post_{t_n} \rightarrow pre_{t_n})$;
  where for each t_i, pre_{ti} is a propositional formula defined over place variables encoding that the precondition of firing t_i is satisfied and $post_{ti}$ is a formula defined over place variables encoding the update in tokens after firing t_i is satisfied.

We denote the encoding of Petri Net system $PN = (N, M_0)$ by $\mathcal{M} = (\mathcal{T}, M_0)$ where M_0 is the initial assignment of the marking vector M.

3 Counting Logic

Our next objective is to identify a suitable logical language for describing properties of the unbounded client-server system. Linear Temporal Logic (LTL) [22,28] is a natural choice to describe temporal properties of systems.

While LTL is useful to us, we cannot express any counting properties, hence, we propose the counting logic language to specify and verify the unbounded client-server system. Logic $\mathcal{L}_C$ is an extension of LTL with a few differences. In the case of LTL, atomic formulas are propositional constants which have no further structure. A monodic formula is a well formed formula with at most one free variable in the scope of a temporal modality. In $\mathcal{L}_C$, there are three types of atomic formulas: (1) describing basic server properties, P_s which are propositional constants (2) **counting** sentences of the kinds $(\#x > \mathfrak{c})\alpha$ and $(\#x \leq \mathfrak{c})\alpha$ over client properties and $\mathfrak{c}$ is a non-negative integer, denoting the number of clients in α, and (3) **comparing** sentences of the kinds $(\#x)\alpha \leq \beta$ and $(\#x)\alpha > \beta$.

Formally, the set of client formulas Δ is given by:

$$\alpha, \beta \in \Delta ::= (\#x > \mathfrak{c})p(x) \mid (\#x \leq \mathfrak{c})p(x) \mid (\#x)p(x) \leq q(x) \mid (\#x)p(x) > q(x) \mid \alpha \vee \beta \mid \alpha \wedge \beta$$

where $p, q \in P_c$ and $\mathfrak{c}$ is a non-negative integer, as noted above. While Δ does not contain an explicit $=$ operator, it is natural to express equality of $p(x)$ and $q(x)$ as follows: $(\#x)p(x) > q(x) \wedge (\#x)q(x) > p(x)$.

The server formulas are defined as follows:

$$\psi \in \Psi ::= q \in P_s \mid \varphi \in \Delta \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid X\psi \mid F\psi \mid G\psi \mid \psi_1 U \psi_2$$

Modalities X, F, G and U are the usual modal operators: Next, Eventually, Globally and Until respectively.

Semantics The logic is interpreted over model sequences. Formally, a model is a sequence $\varrho = m_0, m_1, \ldots$, where for all $i \geq 0$ we have a triple $m_i = (\nu_i, V_i, \xi_i)$ such that:

1. $\nu_i \subset_{fin} P_s$, gives the local properties of the server at instant i.
2. $V_i \subset_{fin} CN$ gives the clients alive at instant i, where CN is a countable set of client names that can be assigned to the processes in the system. Further, for all $i \geq 0$, $V_{i+1} \subseteq V_i$ or $V_i \subseteq V_{i+1}$.
3. $\xi_i : V_i \to 2^{P_c}$ gives the properties satisfied by each live agent at the ith instant.

The truth of a formula at an instant in the model is given by the relations $\models$ and $\models_{\Delta}$ defined by induction over the structure of ψ and α respectively as follows:

1. $\varrho, i \models q$ iff $q \in \nu_i$. Note that q's denote atomic local server propositions. Therefore, a q holds in the model ϱ at instance i if q is in the set ν_i.
2. $\varrho, i \models \varphi$ iff $\varrho, i \models_{\Delta} \varphi$. Recall that φ is a sentence from the set of client formulae Δ. In order to define the satisfiability of φ, we need to use the rules defined for the relation $\models_{\Delta}$.
3. $\varrho, i \models \neg\psi$ iff $\varrho, i \not\models \psi$.
4. $\varrho, i \models \psi \vee \psi'$ iff $\varrho, i \models \psi$ or $\varrho, i \models \psi'$.
5. $\varrho, i \models \psi \wedge \psi'$ iff $\varrho, i \models \psi$ and $\varrho, i \models \psi'$.
6. $\varrho, i \models X\psi$ iff $\varrho, i+1 \models \psi$.
7. $\varrho, i \models F\psi$ iff $\exists j \geq i$, $\varrho, j \models \psi$.
8. $\varrho, i \models G\psi$ iff $\forall j \geq i$, $\varrho, j \models \psi$.
9. $\varrho, i \models \psi_1 U \psi_2$ iff $\exists j \geq i$, $\varrho, j \models \psi_2$ and for all $i \leq j' < j : \varrho, j' \models \psi_1$.
10. $\varrho, i \models_{\Delta} (\#x > \mathfrak{c})p(x)$ iff $|\{a \in V_i \mid p \in \xi_i(a)\}| > \mathfrak{c}$. The client formula $(\#x > \mathfrak{c})p(x)$ holds in model ϱ at instance i if there are strictly more than $\mathfrak{c}$ clients that satisfy the property p at instance i.
11. $\varrho, i \models_{\Delta} (\#x)p(x) \leq q(x)$ iff $|\{a \in V_i \mid p \in \xi_i(a)\}| \leq |\{b \in V_i \mid q \in \xi_i(b)\}|$. The client formula $(\#x)p(x) \leq q(x)$ holds in the model ϱ at instance i if the number of clients satisfying property p is less than the number of clients satisfying the property q, at the same instance i.
12. $\varrho, i \models_{\Delta} (\#x \leq \mathfrak{c})p(x)$ iff $|\{a \in V_i \mid p \in \xi_i(a)\}| \leq \mathfrak{c}$. The client formula $(\#x \leq \mathfrak{c})p(x)$ holds in model ϱ at instance i if there are less than $\mathfrak{c}$ clients that satisfy the property p at instance i.
13. $\varrho, i \models_{\Delta} (\#x)p(x) > q(x)$ iff $|\{a \in V_i \mid p \in \xi_i(a)\}| > |\{b \in V_i \mid q \in \xi_i(b)\}|$. The client formula $(\#x)p(x) > q(x)$ holds in the model ϱ at instance i if the number of clients satisfying property p is strictly more than the number of clients satisfying the property q, at the same instance i.
14. $\varrho, i \models_{\Delta} \alpha \vee \beta$ iff $\varrho, i \models_{\Delta} \alpha$ or $\varrho, \pi, i \models_{\Delta} \beta$.
15. $\varrho, i \models_{\Delta} \alpha \wedge \beta$ iff $\varrho, i \models_{\Delta} \alpha$ and $\varrho, \pi, i \models_{\Delta} \beta$.

To illustrate the expressibility of the counting logic, consider the property of an infinite state system, such that at some time in the future, the number of tokens at place p_1 is greater than the number of tokens in place p_0, is written as $F(\#x)p_1(x) > p_0(x)$. Details of expressing the properties with respect to case studies are discussed in Appendix A.2 and Appendix B.2.

4 The Two Dimensional Bounded Model Checking Strategy

In this section, we set out to enunciate the novel two dimensional bounded model checking strategy that we have devised and implemented to check $\mathcal{L}_C$ specifications against Petri Net models. To quote Biere et al. [6], "The basic idea behind bounded model checking (BMC) is to restrict the general model checking problem to a bounded problem. Instead of asking whether the system M violates the property Ψ, we ask whether the system M has any counterexample of length k to Ψ. This bounded problem is encoded into SAT." In $2D$-BMC, we ask whether the system M has any counterexample of length λ and number of clients κ to Ψ, in the unbounded client-server setting, and encode this bounded problem into SMT.

First, we describe the notation that we have used. Let $\mathcal{M}$ be the propositional encoding of the Petri Net model of the system. Also, let ϕ be the $\mathcal{L}_C$ property that we want to verify. As usual we negate the property ϕ and let $\psi = \neg\phi$. We also assume that ψ is used in its negation normal form.

The $2D$-BMC encoding of $\mathcal{M}$ against ψ for the bound $k = \lambda + \kappa$ (where $k \geq 0$) is denoted by $[\mathcal{M}, \psi]_{\langle\lambda,\kappa\rangle}$ and defined as follows:

$$[\mathcal{M}, \psi]_{\langle\lambda,\kappa\rangle} = [\mathcal{M}]_{\langle\lambda,\kappa\rangle} \wedge \Big(\big(\neg L_{\langle\lambda,\kappa\rangle} \wedge [\psi]^0_{\langle\lambda,\kappa\rangle} \big) \vee \bigvee_{l=0}^{k} ({}_lL_{\langle\lambda,\kappa\rangle} \wedge {}_l[\psi]^0_{\langle\lambda,\kappa\rangle}) \Big)$$

The bound k has two parts λ and κ: λ gives the bound for time instances and κ gives the bound for number of clients. $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$ is the propositional formula encoding the runs of $\mathcal{M}$ of bound $k = \lambda + \kappa$. The formulae $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$, for $0 \leq \lambda, \kappa \leq k$ are defined later in this section.

The formulas ${}_lL_{\langle\lambda,\kappa\rangle}$ $(0 \leq l \leq \lambda)$ and $L_{\langle\lambda,\kappa\rangle}$ are loop conditions that are mentioned in the encoding above. For any $0 \leq l \leq \lambda$, ${}_lL_{\langle\lambda,\kappa\rangle} = \mathcal{T}(s_\lambda, s_l)$ where s_l is the lth state in the run of $\mathcal{M}$ and s_λ is the λth state. Note, here state is defined in terms of value of the marking vector M. So, for any instance i, s_i corresponds to the state of vector M at i. Clearly, when ${}_lL_{\langle\lambda,\kappa\rangle}$ holds it means there is a transition from s_λ to s_l which denotes a back loop to the lth state. The other loop condition is defined as follows: $L_{\langle\lambda,\kappa\rangle} = \bigvee_{0 \leq l \leq \lambda} {}_lL_{\langle\lambda,\kappa\rangle}$. When $L_{\langle\lambda,\kappa\rangle}$ holds, it means there is a back loop to some state in the bounded run of $\mathcal{M}$.

Formula $[\psi]^0_{\langle\lambda,\kappa\rangle}$ is the propositional encoding of ψ for the bound $k = \lambda + \kappa$ when ψ is asserted at initial instance $i = 0$ and there is no loop in the run of $\mathcal{M}$. On the other hand, ${}_l[\psi]^0_{\langle\lambda,\kappa\rangle}$ is the propositional encoding of ψ for the bound $k = \lambda + \kappa$ when ψ is asserted at initial instance $i = 0$ and there is a back loop to the lth state in the run of $\mathcal{M}$. Formulas $[\psi]^i_{\langle\lambda,\kappa\rangle}$ and ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$, for any i, are extensions of similar mappings defined in [5].

The bound $k = \lambda + \kappa$ starts from 0 and is incremented by 1 in each (macro-)step. For a fixed k, λ may start from 0, incremented by 1 in each (micro-)step till k. Simultaneously, κ may move from k to 0 and decrement by 1 in each

(micro-)step. We look at each (micro-)step. Let the variables being used in the Boolean encoding of Petri Net be:

Transitions: $t_0, t_1, \ldots, t_{n_t}$ and Places: $p_0, p_1, \ldots, p_{n_p}$.

We use copies of transition variables $t_{0i}, \ldots, t_{n_t i}$, place variables $p_{0j}, \ldots, p_{n_p i}$, where $0 \leq i \leq \lambda$, in $[\mathcal{M}]_{\langle \lambda, \kappa \rangle}$. For any $\kappa \geq 0$, we define

$$[\mathcal{M}]_{\langle 0,\kappa \rangle} = I(s_0) \wedge (\bigwedge_{0 \leq j \leq n_p} p_{j0} \leq \kappa).$$

Inductively, for any $\lambda > 0$,

$$[\mathcal{M}]_{\langle \lambda,\kappa \rangle} = [\mathcal{M}]_{\langle \lambda-1,\kappa \rangle} \wedge (T(s_{\lambda-1}, s_\lambda) \wedge (\bigwedge_{0 \leq j \leq n_p} p_{j\lambda} \leq \kappa)).$$

Now that we have formally described the notations for encoding of $\mathcal{M}$, we give the propositional encoding of $\mathcal{L}_C$ for $\mathcal{M}$ in the subsequent section.

4.1 Propositional encoding of the logic $\mathcal{L}_C$

We need to introduce counter variables for each place p in the input Petri Net in order to define the propositional encodings of the property ψ. These extra variables are as follows: $\{c_p^i \mid i \geq 0,$ and p is a place in the Petri Net$\}$. Now we are ready to define ${}_l[\psi]^i_{\langle \lambda,\kappa \rangle}$ and $[\psi]^i_{\langle \lambda,\kappa \rangle}$. We define ${}_l[\psi]^i_{\langle \lambda,\kappa \rangle}$ inductively as follows:

1. ${}_l[q]^i_{\langle \lambda,\kappa \rangle} \equiv q_i$
2. ${}_l[\neg q]^i_{\langle \lambda,\kappa \rangle} \equiv \neg q_i$
3. ${}_l[\psi_1 \vee \psi_2]^i_{\langle \lambda,\kappa \rangle} \equiv {}_l[\psi_1]^i_{\langle \lambda,\kappa \rangle} \vee {}_l[\psi_2]^i_{\langle \lambda,\kappa \rangle}$
4. ${}_l[\psi_1 \wedge \psi_2]^i_{\langle \lambda,\kappa \rangle} \equiv {}_l[\psi_1]^i_{\langle \lambda,\kappa \rangle} \wedge {}_l[\psi_2]^i_{\langle \lambda,\kappa \rangle}$
5. ${}_l[X\psi_1]^i_{\langle \lambda,\kappa \rangle} \equiv \begin{cases} {}_l[\psi_1]^{i+1}_{\langle \lambda,\kappa \rangle} & \text{if } i < \kappa \\ {}_l[\psi_1]^{l}_{\langle \lambda,\kappa \rangle} & \text{if } i = \kappa \end{cases}$
6. ${}_l[F\psi_1]^i_{\langle \lambda,\kappa \rangle} \equiv \bigvee\limits_{j=\min(l,i)}^{\kappa} {}_l[\psi_1]^j_{\langle \lambda,\kappa \rangle}$
7. ${}_l[G\psi_1]^i_{\langle \lambda,\kappa \rangle} \equiv \bigwedge\limits_{j=\min(l,i)}^{\kappa} {}_l[\psi_1]^j_{\langle \lambda,\kappa \rangle}$
8. ${}_l[\psi_1 U \psi_2]^i_{\langle \lambda,\kappa \rangle} \equiv \bigvee\limits_{j=i}^{\kappa} ({}_l[\psi_2]^j_{\langle \lambda,\kappa \rangle} \wedge \bigwedge\limits_{n=i}^{j-1} {}_l[\psi_1]^n_{\langle \lambda,\kappa \rangle}) \vee \bigvee\limits_{j=l}^{i-1} ({}_l[\psi_2]^j_{\langle \lambda,\kappa \rangle} \wedge \bigwedge\limits_{n=i}^{\kappa} {}_l[\psi_1]^n_{\langle \lambda,\kappa \rangle} \wedge \bigwedge\limits_{n=l}^{j-1} {}_l[\psi_1]^n_{\langle \lambda,\kappa \rangle})$
9. ${}_l[(\#x > \mathfrak{c})p(x)]^i_{\langle \lambda,\kappa \rangle} \equiv c_p^i > \mathfrak{c}$
10. ${}_l[(\#x \leq \mathfrak{c})p(x)]^i_{\langle \lambda,\kappa \rangle} \equiv c_p^i \leq \mathfrak{c}$
11. ${}_l[(\#x)p(x) > q(x)]^i_{\langle \lambda,\kappa \rangle} \equiv c_p^i > c_q^i$
12. ${}_l[(\#x)p(x) \leq q(x)]^i_{\langle \lambda,\kappa \rangle} \equiv c_p^i \leq c_q^i$
13. ${}_l[\alpha_1 \vee \alpha_2]^i_{\langle \lambda,\kappa \rangle} \equiv {}_l[\alpha_1]^i_{\langle \lambda,\kappa \rangle} \vee {}_l[\alpha_2]^i_{\langle \lambda,\kappa \rangle}$

14. ${}_l[\alpha_1 \wedge \alpha_2]^i_{\langle\lambda,\kappa\rangle} \equiv {}_l[\alpha_1]^i_{\langle\lambda,\kappa\rangle} \wedge {}_l[\alpha_2]^i_{\langle\lambda,\kappa\rangle}$

We define $[\psi]^i_{\langle\lambda,\kappa\rangle}$ inductively as follows:

1. $[q]^i_{\langle\lambda,\kappa\rangle} \equiv q_i$
2. $[\neg q]^i_{\langle\lambda,\kappa\rangle} \equiv \neg q_i$
3. $[\psi_1 \vee \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\psi_1]^i_{\langle\lambda,\kappa\rangle} \vee [\psi_2]^i_{\langle\lambda,\kappa\rangle}$
4. $[\psi_1 \wedge \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\psi_1]^i_{\langle\lambda,\kappa\rangle} \wedge [\psi_2]^i_{\langle\lambda,\kappa\rangle}$
5. $[X\psi_1]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} [\psi_1]^{i+1}_{\langle\lambda,\kappa\rangle} & \text{if } i < \kappa \\ [\psi_1]^{l}_{\langle\lambda,\kappa\rangle} & \text{if } i = \kappa \end{cases}$
6. $[F\psi_1]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee\limits_{j=\min(l,i)}^{\kappa} {}_l[\psi_1]^j_{\langle\lambda,\kappa\rangle}$
7. $[G\psi_1]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge\limits_{j=\min(l,i)}^{\kappa} [\psi_1]^j_{\langle\lambda,\kappa\rangle}$
8. $[\psi_1 U \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee\limits_{j=i}^{\kappa} ([\psi_2]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{n=i}^{j-1} [\psi_1]^n_{\langle\lambda,\kappa\rangle})$
9. $[(\#x > \mathfrak{c})p(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c^i_p > \mathfrak{c}$
10. $[(\#x \leq \mathfrak{c})p(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c^i_p \leq \mathfrak{c}$
11. $[(\#x)p(x) > q(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c^i_p > c^i_q$
12. $[(\#x)p(x) \leq q(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c^i_p \leq c^i_q$
13. $[\alpha_1 \vee \alpha_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha_1]^i_{\langle\lambda,\kappa\rangle} \vee [\alpha_2]^i_{\langle\lambda,\kappa\rangle}$
14. $[\alpha_1 \wedge \alpha_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha_1]^i_{\langle\lambda,\kappa\rangle} \wedge [\alpha_2]^i_{\langle\lambda,\kappa\rangle}$

To determine the encoding of $[\mathcal{M}]_{\langle 0,\kappa\rangle}$, we need to perform **and** operation on the initial state and the place constraints. The place constraints are due to the κ variable, for each of the places.

$$[\mathcal{M}]_{\langle 0,\kappa\rangle} = I(s_0) \wedge (\bigwedge_{0 \leq j \leq n_p} p_{j0} \leq \kappa).$$

Similarly, to determine $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$, we can make use of the previously encoded $[\mathcal{M}]_{\langle\lambda-1,\kappa\rangle}$ and unfold the transition relation by one length, and **and** it with the place constraints to obtain the following relation:

$$[\mathcal{M}]_{\langle\lambda,\kappa\rangle} = [\mathcal{M}]_{\langle\lambda-1,\kappa\rangle} \wedge (T(s_{\lambda-1}, s_\lambda) \wedge (\bigwedge_{0 \leq j \leq n_p} p_{j\lambda} \leq \kappa)).$$

The encoding of the $\neg, \wedge, \vee$ operators are straightforward. We also define the encoding using the temporal operators. Notice that $[(\#x > k)p(x)]^i_{\langle\lambda,\kappa\rangle}$ is encoded using the counter c_p at the place p, and this is translated as $c_p > \kappa$. Similarly, we have $[(\#x > k)p(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c_p > \kappa$.

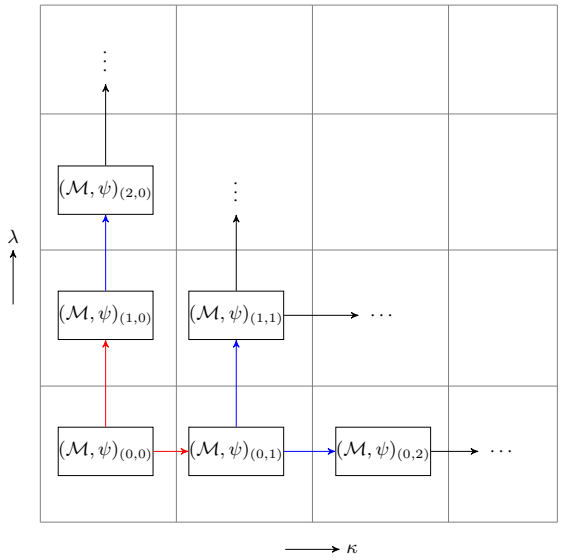

Fig. 1. Unfolding of the formula $(\mathcal{M},\psi)_{(\lambda,\kappa)}$ with respect to λ (time instance), κ (number of clients)

Unfolding the encoded formula The unfolding of the formula $(\mathcal{M},\psi)_{(\lambda,\kappa)}$ for each k, upto $k=2$ is depicted pictorially in Fig. 1. Initially, when $k=\lambda+\kappa=0$, $k=0$ (bound), $\lambda=0$ (time instance), $\kappa=0$ (number of clients), the formula $(\mathcal{M},\psi)_{(0,0)}$ is evaluated. If this is found to be satisfiable, then, a witness is obtained, and we are done. If not, in the next macro-step of the unfolding, where $k=\lambda+\kappa=1$, there are possibly two micro-steps to be explored $\langle\lambda,\kappa\rangle=\langle 0,1\rangle$ and $\langle\lambda,\kappa\rangle=\langle 1,0\rangle$. Firstly, κ is incremented and the formula $(\mathcal{M},\psi)_{(0,1)}$ is evaluated. If this found to be unsatisfiable, λ is incremented and the formula $(\mathcal{M},\psi)_{(1,0)}$ is evaluated. In each micro-step of the unfolding, either λ or κ are incremented, and the resulting formula is verified. Details of the unfolding with respect to the tool are in Section 5.2.

5 The 2*D*-BMC Model Checker Tool: DCModelChecker

DCModelchecker is an easy-to-use tool for performing two dimensional bounded model checking of Petri Nets using the logic $\mathcal{L}_C$. We give the architecture and the overview of the tool in Section 5.1. We describe the workflow in Section 5.2, and in Section 6, we report the experiments. The experiments are easily reproducible by directly executing the scripts in our artifact. It is a self-contained virtual machine image —additional installations or tools are unnecessary for replicating our results.

5.1 Architecture

We introduce a tool to perform two dimensional bounded model checking, DCModelChecker. While several tools perform bounded model checking, ours is unique in the model checking strategy as well as displaying the counterexample. We describe the details of related work in the Section 7.

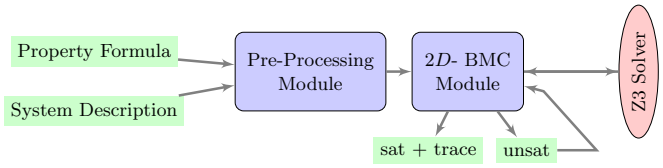

Fig. 2. DCModelChecker architecture

The general architecture of the tool is shown in Fig. 2. Our tool has two primary inputs- system description in standard PNML format and the property to be tested expressed in counting logic $\mathcal{L}_C$. The system description using Petri Nets in PNML format can be obtained from the vast collection of industrial and academic benchmarks available at MCC [17] or created using a Petri Net Editor [30]. We make use of MCC benchmarks, for comparative testing. Currently, we have translated a subset of the MCC properties into $\mathcal{L}_C$.

Firstly, the two inputs are fed to the pre-processing module and consequently to the DCModelChecker tool. The objective of the pre-processing Module is to read and validate the two inputs. We make use of ANTLR [20] to achieve this. We give the grammar of the nets and properties to the tool so that it can recognize it against its respective grammar as shown in Fig. 3. ANTLR generates a parser for that language that can automatically build parse trees representing how a grammar matches the input. The parse trees can be walked to construct the data structures that our tool requires. We have manually written the grammars for the PNML format of both types and the grammar of $\mathcal{L}_C$, to be used by ANTLR.

Our tool reads the output of the pre-processing module and checks if the model satisfies the property or not. We make use of the Z3 SAT/SMT Solver [19], to solve the encoded formula and give us unsat, or sat with a counterexample trace. If unsat, the tool can increment the bound and look further, until the external termination bound is hit. We picked Z3, for its wide industrial applications, developer community support, and ease of use. The detailed workflow is discussed in the subsequent section.

5.2 Workflow

The system description M, the property formula ϕ, and external termination bound k are given to us. First, we negate the property, $\psi = \neg\phi$. Note that the negation normal form of ψ is always used in the BMC process. For the bound

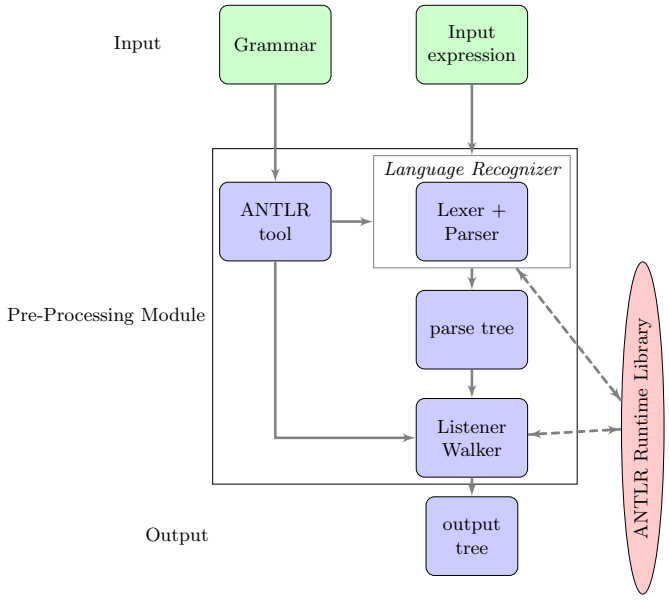

Fig. 3. Pre-Processing the model and formula using ANTLR

$k = 0$ and the corresponding micro-step $\langle \lambda, \kappa \rangle = \langle 0, 0 \rangle$, we construct the formula $[M, \psi]_{\langle 0,0 \rangle}$ and feed it to the solver.

If the above formula is satisfiable, the property ϕ is violated in the initial state and we have a **witness** at $k = 0$ and we can stop our search. If the base case is unsatisfiable, we consider the next micro-step $\langle \lambda, \kappa \rangle$ – with the bound $k = \lambda + \kappa = 1$ – and construct the formula $[M, \psi]_{\langle \lambda, \kappa \rangle}$ and feed it to the solver. If this formula is satisfiable, the property ϕ is violated for $k = \lambda + \kappa$ and we have a **witness** for this k and we can stop our search. Otherwise, we may continue with the next micro-step $\langle \lambda', \kappa' \rangle$ and so on. The order in which the micro-steps are considered is illustrated in Fig. 1. We shall continue until we have considered all micro-steps for the termination bound k.

6 Experiments

To illustrate the expressibility of $\mathcal{L}_C$ and the applicability of $2D$-BMC, we performed two phases of experiments, their results are described in this section. Firstly, we considered benchmarks models from the Model Checking Contest [17] and expressed the properties in $\mathcal{L}_C$. We considered the following property categories and translated them to $\mathcal{L}_C$ —LTLFireability, LTLCardinality, ReachabilityFireability, ReachabilityCardinality. In particular, the results of verifying a subset of properties are consolidated in Table 1. The table shows the number of sat and unsat properties in each property category (consisting of 16 properties each per category), and the time taken to execute (in ms) for DCModelChecker

as well as the state-of-the-art model checking tool, ITS-Tools [26]. Our tool DCModelChecker was executed with the bound 1. ITS-Tools was executed with a timeout of 600 seconds. It can be observed that the DCModelChecker performs well in some property categories. While some properties were unsat for the above bound and timeout period constraints, they may have become sat, on incrementing the bound. Hence there is a difference in the number of sat and unsat properties found by DCModelChecker versus ITS-Tools. For LTL Fireability and LTL Cardinality properties, DCModelChecker takes comparatively less execution time than ITS-Tools. However, the number of sat and unsat properties differ slightly. As part of future work, we would like to identify the property types where DCModelChecker has an advantage. Our tool DCModelChecker is available online [21] and the results in this section can be easily reproduced.

Model Name	Property Category	DCModelChecker			ITS-Tools		
		sat	unsat	time(s)	sat	unsat	time(s)
Dekker-PT-010	LTL Cardinality	3	13	11.219	4	12	15.7
	LTL Fireability	2	14	10.353	2	14	18.372
	Reachability Cardinality	0	16	10.497	5	11	3.45
	Reachability Fireability	0	16	11.628	4	12	6.061

Table 1. Results of comparative testing

Secondly, we considered two infinite-state systems —firstly, the Autonomous Parking System (APS) and secondly, the Unbounded Process Scheduler (UPS), modeled as a net, primarily to take a closer look at how exactly the $2D$-BMC strategy works, given that UPS is an extensively researched system [18] and has a much smaller net representation. The APS and UPS case studies are discussed in Appendix A and Appendix B respectively. Some of the properties of the Autonomous Parking System verified using DCModelChecker are given in Table 2. A sample of the properties that were written using the counting logic $\mathcal{L}_C$ for the UPS to demonstrate its expressibility are in Table 3. It is natural to see that we can expand this to more complex properties.

With respect to the UPS, consider the property $F(\#x)p_1(x) > p_0(x)$, written in $\mathcal{L}_C$. It is a counting property with a finally operator at the root. Hence, this property gets violated when there is an instance where the number of processes at place p_1 is less than the number of processes/tokens at place p_0. In the model, this happens at time instant $\lambda = 3$, when the number of processes $\kappa = 1$, and a counterexample trace is obtained as well.

To the best of our knowledge, ours is the only tool that uses $2D$-BMC in the Petri Net setting, for an extension of linear temporal logic while making use of an SMT solver and gives a counterexample trace. In Section 7 we discuss the contributions of other tools that verify the properties of Petri Nets.

S.No	APS Property	Result
1	$G((\#x)p2(x) > p3(x) \mid (\#x)p3(x) > p2(x))$	$k = 1, \kappa = 1, \lambda = 0$. SAT.
2	$F((\#x)p2(x) <= p3(x) \& (\#x)p3(x) <= p2(x))$	External Termination Bound reached. UNSAT.
3	$G((\#x > 0)p1(x) \mid (\#x > 0)p7(x) \mid (\#x > 0)p8(x))$	External Termination Bound reached. UNSAT.
4	$F((\#x)p6(x) > p2(x))$	$k = 4, \kappa = 1, \lambda = 3$ .SAT.
5	$((\#x > 0)p1(x) \mid (\#x > 0)p7(x) \mid (\#x > 0)p8(x))U$ $((\#x)p2(x) <= p3(x) \& (\#x)p3(x) <= p2(x))$	External Termination Bound reached. UNSAT.

Table 2. Results of testing APS using DCModelChecker

S.No	UPS Property	Result
1	$F(\#x)p_1(x) > p_0(x)$	$k = 4$, $\kappa = 1$, $\lambda = 3$, SAT.
2	$G(\#x)p_2(x) <= p_1(x) <= p_0(x)$	$k = 3$, $\kappa = 1$, $\lambda = 2$, SAT.
3	$G(t_0 \& t_1 \& t_7)$	$k = 0$, $\kappa = 0$, $\lambda = 0$, SAT
4	$F(t_0 \& t_1 \& t_7)$	External Termination Bound reached. UNSAT.
5	$F(\#x <= 1)p_2(x) \& (\#x <= 3)p_1(x)$ $\& (\#x <= 2)p_0(x)$	External Termination Bound reached. UNSAT.

Table 3. Results of testing UPS using DCModelChecker

7 Related Work

Recently, there have been several practical implementations of tools verifying specific properties of Petri Nets. KREACH [12] is a recent implementation of Kosaraju's algorithm for deciding reachability in Petri Nets (and equivalently Vector Addition Systems with States). QCOVER [7] decides the coverability of Petri Nets leveraging the SMT solver. ICOVER [14] decides coverability using inductive invariants. In [23] SMT-based reachability of timed Petri Nets and in [8] CTL^* model checking for time Petri Nets are explored. Petrinizer [13] implements an SMT-based approach for coverability.

BMC using SMT Solvers has been an active area of research [24]. In particular, while applying BMC in the Petri Net setting, techniques such as net reductions, and structural reductions [27] are applied. Recently, in [1,2] the generalized reachability of Petri Net is encoded into a BMC problem and solved using SMT solvers.

However, BMC for Petri Nets while expressing properties using LTL and its extensions while leveraging the power of SMT Solvers remains unexplored. DCModelChecker attempts to bridge this gap while preserving the original Petri Net structure. In Section 4.1 we provide the propositional encoding of the counting logic $\mathcal{L}_C$ which distinguishes our work. In [25] a similar counting linear temporal logic for specifying properties of multi-robot applications is proposed. In contrast, $\mathcal{L}_C$ uniquely supports specifying properties of client-server systems.

The Model Checking Contest (MCC) [17] has attracted many tools for formal verification of concurrent systems, and also industrial and academic benchmarks. Tools participating in MCC, use portfolio approach and give answers in binary, based on the requirement of the contest. In particular, we looked at

ITS-Tools [26] which has 99.99% tool confidence across categories in the contest and makes use of structural reduction techniques [27] to arrive at the solution quickly. ITS-Tools uses a layer of SMT when solving LTL, however, it does not perform BMC.

While ITS-Tools is much faster than our tool, it gives only binary answers (sat or unsat), as this conforms with the expectations of the MCC unlike our tool which gives a counterexample trace, if the property is not satisfied. The counterexample trace is quite useful for property verification. To the best of our knowledge, DCModelChecker is the only tool that performs bounded model checking for Petri Nets using an extension of linear temporal logic using SMT solvers and provides a counterexample trace, that can be easily interpreted. Currently, the goal of this tool is to demonstrate the usefulness of the $2D$- BMC strategy and the logic $\mathcal{L}_C$. To ensure the correctness of the results, we executed the same benchmarks on ITS-Tools as well. The results of comparative testing are in Section 6. These experiments can be replicated using the scripts available in our artifact [21].

8 Conclusion and Future Work

We have successfully explored the reachability, fireability, and cardinality properties of the unbounded client-server systems using the proposed counting logic $\mathcal{L}_C$. We have also proposed a novel bounded model checking strategy $2D$-BMC applicable for Petri Nets. We have implemented a tool DCModelChecker that uses the proposed counting logic and the $2D$-BMC strategy and demonstrated its usefulness concerning benchmarks from MCC. To the best of our knowledge, DCModelChecker is the only tool that uses $2D$-BMC for Petri Nets, specifying properties in an extension of linear temporal logic while utilizing an SMT solver and additionally providing a counterexample trace. Future research directions include optimizing our tool by implementing an efficient linear size encoding in the bound k for the logic [6]. The BMC technique only provides the counterexamples when the property is unsatisfiable. It would be worth the effort to leverage inductive reasoning to have a full decision procedure, for when the property is satisfiable. QCOVER [7] and ICOVER [14] already have implemented such a procedure for a specific set of properties of Petri Nets. As part of future work, we would like to establish the completeness thresholds of BMC [11]. Another work in progress is an extension of $\mathcal{L}_C$ to account for identifiable clients, which will enable us to verify richer infinite-state systems.

References

1. Amat, N., Berthomieu, B., Dal-Zilio, S.: On the combination of polyhedral abstraction and smt-based model checking for petri nets. In: PETRI NETS. pp. 164–185. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_9
2. Amat, N., Dal-Zilio, S., Hujsa, T.: Property directed reachability for generalized petri nets. In: TACAS. pp. 505–523. LNCS, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_28

3. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. pp. 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS. pp. 193–207. LNCS, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Log. Methods Comput. Sci. (2006). https://doi.org/10.2168/LMCS-2(5:5)2006
7. Blondin, M., Finkel, A., Haase, C., Haddad, S.: Approaching the coverability problem continuously. In: TACAS. pp. 480–496. LNCS, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_28
8. Boucheneb, H., Hadjidj, R.: CTL^* model checking for time petri nets. Theor. Comput. Sci. pp. 208–227 (2006). https://doi.org/10.1016/j.tcs.2005.11.002
9. Cai, Z., Zhou, Y., Qi, Y., Zhuang, W., Deng, L.: A millimeter wave dual-lens antenna for iot-based smart parking radar system. IEEE Internet Things J. pp. 418–427 (2021). https://doi.org/10.1109/JIOT.2020.3004403
10. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. pp. 7–34 (2001). https://doi.org/10.1023/A:1011276507260
11. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Computational challenges in bounded model checking. Int. J. Softw. Tools Technol. Transf. pp. 174–183 (2005). https://doi.org/10.1007/s10009-004-0182-5
12. Dixon, A., Lazic, R.: Kreach: A tool for reachability in petri nets. In: TACAS. pp. 405–412. LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_22
13. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P.J., Niksic, F.: An smt-based approach to coverability analysis. In: CAV. pp. 603–619. LNCS, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_40
14. Geffroy, T., Leroux, J., Sutre, G.: Occam's razor applied to the petri net coverability problem. Theor. Comput. Sci. (2018). https://doi.org/10.1016/j.tcs.2018.04.014
15. Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: PNML framework: An extendable reference implementation of the petri net markup language. In: PETRI NETS. pp. 318–327. LNCS, Springer (2010). https://doi.org/10.1007/978-3-642-13675-7_20
16. Hsu, C., Shih, M.H., Huang, H.Y., Shiue, Y.C., Huang, S.: Verification of smart guiding system to search for parking space via DSRC communication. In: ITST. pp. 77–81. IEEE (2012). https://doi.org/10.1109/ITST.2012.6425287
17. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, ., Walner, A., Wolf, K.: Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (June 2021)
18. Mazur, T., Lowe, G.: Counter abstraction in the CSP/FDR setting. Electron. Notes Theor. Comput. Sci. (2009). https://doi.org/10.1016/j.entcs.2009.08.012
19. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. pp. 337–340. LNCS, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Parr, T., Fisher, K.: Ll(*): the foundation of the ANTLR parser generator. In: PLDI. pp. 425–436 (2011). https://doi.org/10.1145/1993498.1993548
21. Phawade, R., Prince, T., Sheerazuddin, S.: Artifact and instructions to generate experimental results for two dimensional bounded model checking of unbounded client-server systems (2022). https://doi.org/10.6084/m9.figshare.19611477.v3

22. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
23. Pólrola, A., Cybula, P., Meski, A.: Smt-based reachability checking for bounded time petri nets. Fundam. Informaticae (2014). https://doi.org/10.3233/FI-2014-1135
24. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in sat-based formal verification. Int. J. Softw. Tools Technol. Transf. pp. 156–173 (2005). https://doi.org/10.1007/s10009-004-0183-4
25. Sahin, Y.E., Nilsson, P., Ozay, N.: Multirobot coordination with counting temporal logics. IEEE Trans. Robotics **36**(4), 1189–1206 (2020). https://doi.org/10.1109/TRO.2019.2957669
26. Thierry-Mieg, Y.: Symbolic model-checking using its-tools. In: TACAS. pp. 231–237. LNCS, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_20
27. Thierry-Mieg, Y.: Structural reductions revisited. In: PETRI NETS. pp. 303–323. LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-51831-8_15
28. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS. pp. 322–331 (1986)
29. Yan, G., Yang, W., Rawat, D.B., Olariu, S.: Smartparking: A secure and intelligent parking system. IEEE Intell. Transp. Syst. Mag. pp. 18–30 (2011). https://doi.org/10.1109/MITS.2011.940473
30. Zahoransky, R.M., Holderer, J., Lange, A., Brenig, C.: Process analysis as first step towards automated business security. In: ECIS. p. Research Paper 46 (2016), `http://aisel.aisnet.org/ecis2016_rp/46`

A Case Study : Autonomous Parking System

In crowded urban spaces, parking one's vehicle can be a hassle, given that there is no guarantee of finding an available parking lot in a reasonable amount of time. There have been various approaches to resolve this, from the industry as well as academia [29,9]. The automobile industry has implemented quite successfully, the autonomous parking system, wherein the idea is that the vehicle can be guided into a parking lot, without the need for a valet through communication between the vehicle and the parking space environment. This is especially useful in smart city projects, and futuristically, the driverless vehicle eco system [16].

In this section, we explore how to verify an Autonomous Parking System (APS) where the number of parking requests is unbounded. This is an example from client-server paradigm, more specifically, a single server multiple client system —the parking system is a server whereas the vehicles (say, cars) requesting parking lots are clients. This can be represented by a state-transition system as given Fig. 4 and Fig. 5, and consequently, as a Petri Net as in Fig. 6.

In any configuration (marking) of the Petri Net execution, the number of tokens in any place corresponds to those many clients (cars) in that state. We will now look at the formal description of the APS as a net and express its properties using $\mathcal{L}_C$.

A.1 Introduction to Autonomous Parking System

Typically, any parking space in a busy area, such as the city square, or the airport, services hundreds of vehicles. It may employ valets to assist the drivers. It is often the case, that the parking space may not be available, and navigating to a parking lot can be cumbersome. Finding a parking lot and arriving at it, is a process that can be autonomous. The system itself can be made autonomous, by using technology to track the vehicles and respond to requests for parking. In this case study, the autonomous parking system is the server, and the vehicles making requests for parking are the clients. Theoretically, there could be an unbounded number of clients. We assume that there is enough parking space available, to demonstrate this unbounded client-server system.

Modeling of Autonomous Parking System using Petri Nets. We will now model the interactions between the parking system and the vehicle requesting parking space as in Fig. 4 and Fig. 5. Initially, the system is ready to service the requests, which is the *server_ready* state. We assume a steady inflow of requests for parking, to keep the system sufficiently occupied. When a vehicle inquires for parking space, the vehicle is in the *parking_requested* state. Now, the system may non-deterministically choose to either grant or reject the parking request based on local information such as space availability, the priority of incoming requests, and the like. We assume two disjoint workflows for each scenario. Firstly, if the system accepts the request, the system is in *request_granted* state and simultaneously, the vehicle goes to *occupy_parking_lot* state. At some point, the vehicle

gives up its allocated parking space, is in *exit_parking_lot_successfully* and simultaneously the system is in *deallocate_parking_lot* state. This marks the successful exit of the vehicle from the system. Secondly, if the system rejects the request, the vehicle is in *parking_unavailable* state and the system is in *request_rejected* state. The only option is for the vehicle to exit. At any point, the system can either accept or reject the request, after granting the request, the server can go to *server_busy* state. Theoretically, this description allows for an unbounded number of vehicle requests to be processed by the system, albeit there may be limitations on the availability of the parking space. We assume that the autonomous parking system can reasonably guide the vehicle manoeuvres within the parking lot. It is not difficult to observe that the combined interactions between the system and vehicles described above can be interleaved and modeled as a single Petri Net as in Fig. 6. In this Petri Net, the successful scenario is marked with a green rectangle, and the places p_0, p_2 and p_4 correspond to the vehicle states *parking_requested*, *occupy_parking_lot* and *exit_parking_lot*, the places p_1, p_7, p_3 and p_5 correspond to the server states *server_ready*, *server_busy*, *request_granted* and *deallocate_parking_lot*, transitions t_1, t_2, t_3 represent the request, entry and exit of the vehicle respectively. The firing of transition t_0 acts as the source, where an unbounded number of vehicle requests can be spawned. The firing of transition t_6 represents the server completing the processing of a request and returning to its ready state. In the unsuccessful scenario, marked with a red rectangle, the transition t_4 is fired when the server rejects the request, which brings the vehicle to *parking_unavailable* represented by place p_6, and the server goes to *request_rejected* represented by place p_8. The firing of transition t_7 ensures that the rejected request is processed by the server, and it is ready to take on more vehicle requests. Notice that the initial marking of the system is given as follows, assuming an ordering on the places $M_0 = (0, 1, 0, 0, 0, 0, 0, 0, 0)$, where there is exactly one token in place p_1 indicating the availability of the server. In any marking of the net, there is exactly one token in either of the server places p_1, p_7, or p_8 due to the structure of the net.

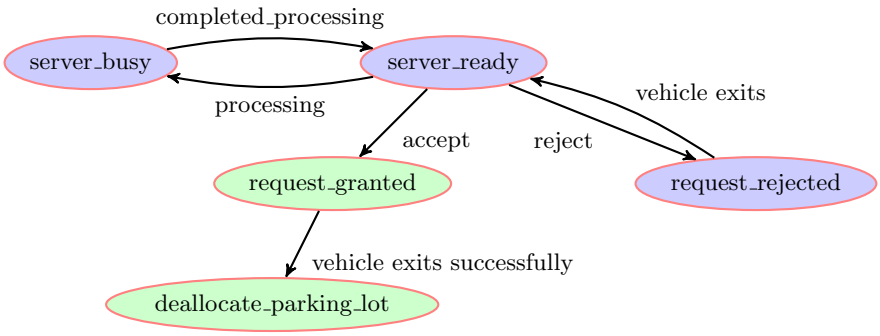

Fig. 4. State diagram of autonomous parking system (server)

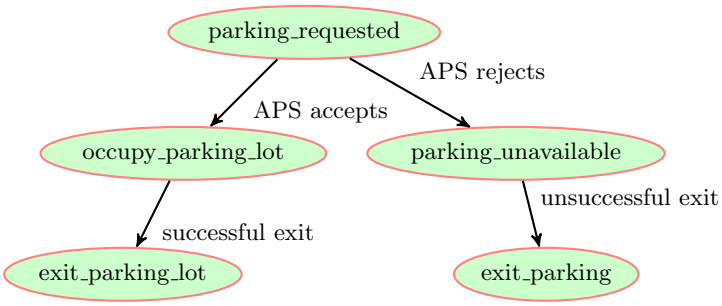

Fig. 5. State diagram of vehicle in APS (client)

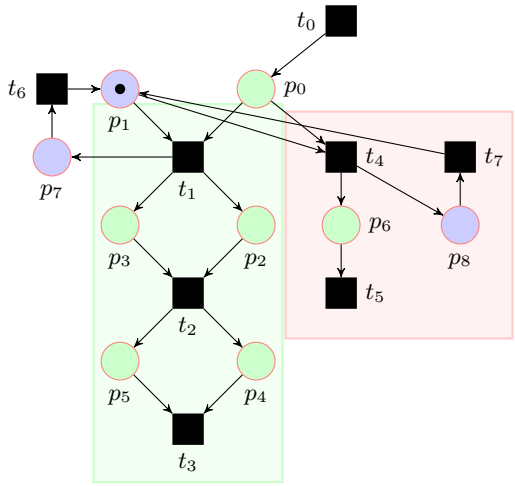

Fig. 6. Petri Net depicting client-server interactions in APS

The standard representation of the Petri Net is the PNML format [15]. We made use of Wolfgang Tool [30], a Petri Net editor to draw and visualize the configurations of the net. The PNML file for APS and the executable (jar file) for the tool are available in the artifact.

A.2 Expressing properties in $\mathcal{L}_C$ and the results of 2D-BMC

1. Consider the property of the APS where there are equal number of accept responses by the server to the number of accepted parking requests by the clients. This can be paraphrased as always the number of tokens in place p_2

equals the number of tokens in p_3. Consider the following property written in $\mathcal{L}_C$:
$G((\#x)p2(x) > p3(x)|(\#x)p3(x) > p2(x))$
Our algorithm looks for a counterexample trace. This is satisfiable initially, at $k = 0$ where $\kappa = 1$ (number of tokens) and $\lambda = 0$ (time instance).

2. $F((\#x)p2(x) <= p3(x)\&(\#x)p3(x) <= p2(x))$
We expect that this formula will be true for all bounds. Hence while having bound $k = 5$ or $k = 100$, we get the same output on reaching the bound, without any counterexample.
3. Consider the property, where there is atleast one token in either place p_1 or p_7 or p_8. From the net, $p_1 + p_7 + p_8 = 1$ is an invariant. These places represent the server being available, busy and server rejecting the request.
$G((\#x > 0)p1(x)|(\#x > 0)p7(x)|(\#x > 0)p8(x))$.
On verifying with our tool, we do not find any counterexample for this property for bound $k = 5$.
4. Comparison between the number of rejected requests and accepted requests.
$F((\#x)p6(x) > p2(x))$.
At $k = 4,\ \kappa = 1,\ \lambda = 3$ we encounter a loop, and the property is unsatisfiable for all paths of length $k = 4$.
5. Consider this property using the Until operator, essentially this is $true\ U\ p_2 = p_3$.
$((\#x > 0)p1(x)|(\#x > 0)p7(x)|(\#x > 0)p8(x))U((\#x)p2(x) <= p3(x)\&(\#x)p3(x) <= p2(x))$. As expected, this property holds true for bound $k = 5$, and no counterexample is found.

B A Case Study : Unbounded Process Scheduler

In this section, we explore how to verify a process scheduler with unbounded processes in an operating system. This is an example from the client-server paradigm: the scheduler is a server whereas the processes requesting CPU time are clients. Each process has a behaviour that can be represented by a state-transition system [18] as given in the Fig. 7.

Unbounded Process Scheduler (UPS), that we have considered in our paper, is a kind of single server multiple client system, where one server and an unbounded number of clients collaborate. In the process model we map each state from $\{new, runnable, running, blocked, terminated\}$ to a place in the Petri Net. In any configuration (marking) of the Petri Net execution, the number of tokens in any place corresponds to as many UPS clients in that state. We will now look at the formal description of the UPS as a net and express its properties using $\mathcal{L}_C$.

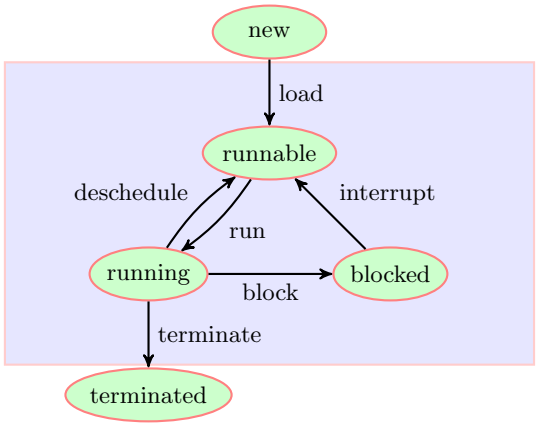

Fig. 7. State diagram of the basic model for processes within an operating system

B.1 Introduction to Unbounded Process Scheduler

Processes are programs in execution. Typically, a process changes its state dynamically based on internal and external inputs. When a process is created, it is in the *new* state. When it is waiting to be assigned a processor, it is in *ready* state, alternatively called as *runnable* state. When instructions are being executed in the process, it is *running*. The running process may be interrupted by external events such as waiting for keyboard input when

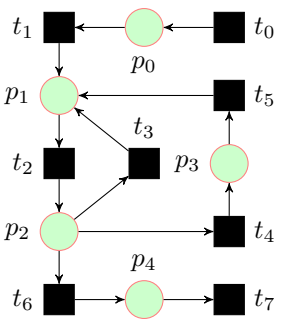

Fig. 8. Petri net for UPS

it gets *blocked*. On event completion, *blocked* process goes to *runnable*, and waits for its turn to run. A *running* process may complete its execution and *terminate*. This is usually known as the lifecycle of a process in a process scheduler given in Fig. 7.

Modeling of Unbounded Process Scheduler using Petri Nets. It is not difficult to observe that the unbounded process scheduler described above can be modeled as a Petri Net as in Fig. 8. The initial marking here is given as follows, assuming an ordering on the places $M_0 = (0, 0, 0, 0, 0)$. Here each token corresponds to a process. Transition t_0 is used to spawn new processes, while transition t_7 is used to take out the processes, terminated by transition t_6. Transition t_1 is used to load the new processes, transition t_2 is to used to run, while t_3 is used to deschedule it. Transition t_4 corresponds to blocking a process, while t_5 corresponds to interrupt. The standard representation of the Petri Net is the PNML format [15]. A Petri Net Editor such as Wolfgang Tool [30] can be used to view existing PNML files as a graphical representation as well as execute the enabled transitions and view the run of the Petri Net. Additionally, Wolfgang allows us to draw a Petri Net of our own. We drew our own Petri Net, using Wolfgang [30] for the UPS case study.

B.2 Expressing properties in $\mathcal{L}_C$ and the results of 2D-BMC

The following are the descriptions of the properties of the UPS described above, along with the results from Table. 3.

1. Consider the property of the UPS, $F(\#x)p_1(x) > p_0(x)$ which expresses that at some time, the number of runnable processes at place p_1 is greater than the number of newly created processes at place p_0. This is satisfiable at time instance $\lambda = 3$, when there is exactly one runnable process in place p_1 leading to $\kappa = 1$ and place p_0 does not contain any processes.
2. Consider the property $G(\#x)p_2(x) <= p_1(x) <= p_0(x)$ which states that it is always the case that the number of running processes is less than or equal to the number of runnable processes and the number of runnable processes is less than or equal to created processes. This is satisfiable at time instance $\lambda = 2$ and $\kappa = 1$.
3. Consider the property $G(t_0 \& t_1 \& t_7)$ which states that it is always the case that process creation, process loading into memory, and process scheduled to run occur simultaneously. This is satisfiable at $\kappa = 0,\ \lambda = 0$.
4. Consider the property $F(t_0 \& t_1 \& t_7)$ which states that at some time process creation, process loading into memory, and process scheduled to run occur simultaneously. This is unsatisfiable up to bound $k = 5$ and the external termination bound is reached. This indicates that such a scenario is not possible up to the given bound.

5. Consider the property $F(\#x <= 1)p_2(x)\&(\#x <= 3)p_1(x)\&(\#x <= 2)p_0(x)$ which states that at some time there is at most 1 running process and there are at most 3 runnable processes and there are at most 2 newly created processes in the scheduler. This is not satisfiable up to the bound $k = 5$, hence the program exits, saying that the external termination bound is reached.