Formal Verification of systems with unbounded agents

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

by

Tephilla Prince
(Roll No. CS18DP001)

Under the guidance of
Ramchandra Phawade
and
S. Sheerazuddin

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DHARWAD
June, 2025

To my mentors

Thesis Approval

The thesis entitled

Formal Verification of systems with unbounded agents

by

Tephilla Prince
(Roll No. CS18DP001)

is approved for the degree of

Doctor of Philosophy

Internal Examiner None	External Examiner Affiliation:
Supervisor Ramchandra Phawade	External supervisor S. Sheerazuddin

Chairman

Date: _____

Place: _____

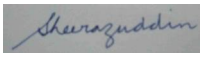
RPC Approval for Thesis Submission

This is to certify that the thesis titled **Formal Verification of systems with unbounded agents** by Tephilla Prince (CS18DP001) is approved for final submission by the Research Progress Committee.

Name: Ramchandra Phawade

Affiliation: IIT Dharwad

Role: Supervisor



Name: S. Sheerazuddin

Affiliation: NIT Calicut

Role: External supervisor

Name: Prof. Bhaskar A

Affiliation: BITS, Goa

Role: RPC Member

Name: Prof. Sreejith AV

Affiliation: IIT Goa

Role: RPC Member

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Tephilla Prince
Roll No. CS18DP001

Date:

Abstract

The complexity and size of software has increased with time. And with it, the necessity to provide mathematical guarantees to prove the behaviour of the software with respect to its requirements has increased. Empirical studies show that the sooner the errors are identified and rectified in the software development lifecycle, the lesser the error costs. If there are errors in the design of the software, they propagate to errors in the implementation which become tedious to resolve. Being able to verify the specifications of these systems in the design stage of the software development (such as using bounded model checking) is an important activity that can be done using formal techniques. The characteristics of the software play a role in the choice of the formal technique to verify a software. Client server systems are one of the largest programming paradigms. Crypto exchanges with an unbounded number of investors, multiplayer games where the number of players are not known apriori and services with an unbounded customer base are instances of unbounded client server systems.

We focus on client server systems with single server and unboundedly many clients, where the number of clients are not known apriori and there can be unbounded concurrent interactions between the clients and server. The major challenges are to identify the suitable model to abstract the behaviour of the systems, to identify suitable logics to specify the properties and to identify formal techniques for verifying the properties on the model. First, we model unbounded client-server systems as unbounded Petri nets and their temporal properties are expressed in Linear Temporal Logic (LTL) with integer arithmetic. In this system, there is unboundedness in two dimensions, namely, the number of clients as well as in the client server interactions. Hence, we propose two dimensional bounded model checking (2D-BMC) for exploiting the unboundeness in these two dimensions and utilize SMT solvers for verification.

Second, when the unbounded client-server systems have distinguishable clients, unbounded Petri nets are no longer suitable to model them. Hence, we explore suitable concurrent models where the tokens are distinguished such as ν-nets. We also require a suitable logic,

which is richer than LTL, to represent their properties naturally, such as a variant of First Order logic. The 2D-BMC algorithm is used to automatically verify the system with these extensions and implemented as a prototype.

Third, clients have their own internal structure. We model each client as a Petri net which interacts with the server Petri net. The interaction of unbounded client server systems can be represented as client Petri nets nested within the server Petri net. This is a type of higher order Petri nets, called Elementary Object Systems (EOS). We also explore the possibility of losing information in the EOS at various nesting levels and their effects on the reachability and coverability problems and chart their decidability status. We built a tool to verify properties such as safety, deadlock and reachability properties using LTL on the standard PN and EOS, as well as their lossy counterparts.

This gives us a suite of formal verification tools for client server systems with unbounded clients using various concurrent models and various suitable logics to represent their properties.

Contents

List of Tables

List of Figures

Glossary

Chapter 1

Introduction

In today's software ecosystem, it has become imperative to give mathematical guarantees for the behaviour of a software with respect to its requirements. Formal verification is the procedure to ensure the correctness of a system with respect to a desired property by checking whether a mathematical model of the system satisfies a specification. There are various *formal* techniques to verify a software system. The applicability and usage of a formal technique depends on various factors such as, the type of software, the stage of the software development life cycle where verfication is performed, the resources available for verification. For instance, in the design stage, it maybe possible to verify properties on the abstraction of a system, such as model checking (cf. p8, [13]). In the implementation stage, a more dynamic monitoring of the system behaviour and verification might be required such as runtime verification [18], where one works directly with the system (as opposed to an abstraction of it) and on select execution traces of the system (as opposed to all execution traces as in model checking), or depending on the resources, one might want to perform a check and provide correctness guarantees before deployment of the software such as static analysis [32]. The characteristics of the software system play a huge role in the selection of the formal technique that is adopted, such as if the interactions occur one after the other (sequential behaviour), if the interactions occur in parallel (concurrent behaviour), or if the interactions occur in parallel and can be made to occur one after the other (serializable), or if the system properties can be parameterized. We take the perspective of analyzing suitable ways to formally verify a particular software system.

In this work, we focus on unbounded client server systems where there is unbounded concurrency in the interaction between the clients and the servers as well as unboundedness in the number of clients (the number of clients is not known apriori). A first challenge is to identify

the suitable representation of unbounded client server systems, we explore well known models in literature by varying their expressivity such as Petri nets, ν-nets (Petri nets with names), and Elementary Object Systems (Petri nets with nesting). Another challenge is to identify the suitable way to specify the properties of the unbounded client server systems, we work with various temporal logics such as Linear Temporal Logic, Linear Temporal Logic with integer arithmetic, a variant of First Order Logic with monodic restriction. Now that the system can be represented using a suitable formal model and the property can be specified using an appropriate logic, the consequent challenge is to identify the formal technique by which to verify the system against its specifications. According to studies by NASA and the Systems Sciences Institute at IBM [41, 55], the error costs associated with identifying mismatches between requirements and the implemented software at a later stage of the software development lifecycle grow exponentially as the software goes through its development lifecycle. Therefore, we would like to use model checking to verify the unbounded client server systems at the design stage, in order to give formal guarantees early on, and avoid the high error cost. There are also other formal techniques in the design stage, such as equivalence checking, where one verifies whether two different mathematical models of a design represent the same behaviour. However, this is not interesting in our application.

Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model. This is a technique that can be used in the design stage of the software development lifecycle. Model checking is a verification technique that has the following features. The software (to be verified) is abstracted into a suitable mathematical *model* and the various behaviours of the *model* are identified as *states*. Those *states* that are possible from the initial setup of the model are called *reachable states*. As stated earlier, the requirements to be verified are formulated using *temporal logic*, it is temporal, to account for the time based properties. The reachable states of the system are traversed to verify the temporal logic properties. If the property fails, we obtain a sequence of states, known as the counterexample. Hence, it is automatic. Thus, model checking can be summarized as an algorithmic technique for checking temporal properties of systems. The model checking problem is also described as: Given a system P and a specification α on the runs of the system, decide whether system P satisfies specification α. This consists of checking that all runs of P constitute models for α. It suffices to show that no run of P is a model for $\neg\alpha$, which is the same as checking that

the intersection of the language accepted by P and the language defined by $\neg\alpha$ is empty. For compex systems with millions of states, model checking can quickly run into the state space explosion problem [35]. If there was a system with n components and each component contained m states, the composition of those components, would contain m^n states, thereby exponentially incrementing the number of states. This issue is tackled, by performing model checking on *bounded* (finite) runs of the system, and restricting the *state space*. This technique is called *bounded* model checking, which is illustrated in Fig. 1.3. In order to describe the behaviours of infinite state systems, such as unbounded client server systems, it is useful to think of automata that accept languages of infinite words, which we shall explore in the upcoming section.

1.1 Infinite State Systems

In this section, we explore automata that accept languages of infinite words such as Büchi automata [101]. Later on, we shall describe Petri nets, which can directly represent behaviours of the infinite state systems with concurrency (See Sec. 2.1)[1]. The term ω- regular languages, describes the class of languages over infinite words, which is accepted by finite state machines. An infinite word $\alpha \in \Sigma^\omega$, is an infinite sequence of symbols from the alphabet Σ. The infinite word can be represented as a function $\alpha : \mathbb{N}_0 \rightarrow \Sigma$. In order to refer to the letter at the ith position, we use $\alpha(i)$. For a finite sequence $[m \dots n]$, $m, n \in \mathbb{N}_0, m \leqslant n$, we denote $\{m, \dots, n\}$. Hence, $\alpha[m \dots n]$ denotes the finite word occurring between positions m and n, $\alpha(m)\alpha(m+1)\dots\alpha(n)$. It follows that $[m \dots]$ denotes the infinite word $\alpha(m)\alpha(m+1)\dots$ starting at m. Suppose S is a set and σ is an infinite sequence of symbols over S, $\sigma : \mathbb{N}_0 \rightarrow S$. Then, $inf(\sigma)$ denotes the set of symbols in S that occur infinitely often in σ. It can be formally defined as: $inf(\sigma) = \{s \in S | \exists^\omega n \in \mathbb{N}_0 : \sigma(n) = s\}$.

Definition 1.1.1. *An automaton is defined as a triple $A = (S, \rightarrow, S_{in})$ where S is a finite set of states, $S_{in} \subseteq S$ is the set of initial states, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation. In case of deterministic automaton, $\rightarrow$ is a function from $S \times \Sigma \rightarrow S$.*

Definition 1.1.2. *A run of a Büchi automaton, is a legal sequence of states that an automaton can pass through while reading the input α. Let $A = (S, \rightarrow, S_{in})$ be an automaton and $\alpha : \mathbb{N}_0 \rightarrow \Sigma$ be an input word. A run of A on α is an infinite sequence $\rho : \mathbb{N}_0 \rightarrow S$ such that*

[1] Alternatively, one might make a product automata to accept languages of infinite words where the underlying model allows for concurrency. However, Petri nets are a more elegant representation.

$\rho(0) \in S_{in}$ *and* $\forall i \in \mathbb{N}_0, \rho(i) \xrightarrow{\alpha(i)} \rho(i+1)$. *A run of* A *on the finite word* $w = a_0 a_1 \dots a_m$ *is a sequence of states* $s_0 s_1 \dots s_{m+1}$ *such that* $s_0 \in S_{in}$ *and* $\forall i \in [0 \dots m], s_1 \xrightarrow{\alpha(i)} s_{i+1}$.

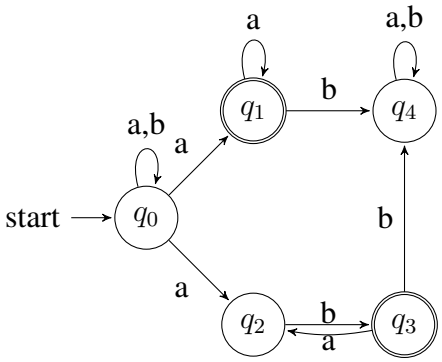


Figure 1.1: An equivalent BA

Example 1.1.3. *Consider the above figure with* $F = \{q_1, q_3\}$ *and the* ω*-word* $\alpha = abbaabababa \dots$*, some of the possible runs on* α *are given below:*

	a	b	b	a	a	b	a	b	a	b. . .
q_0	q_0	q_0	q_0	q_0	q_0	q_0	q_2	q_3	q_2	$q_3 \dots$
q_0	q_0	q_0	q_0	q_1	q_1	q_4	q_4	q_4	q_4	$q_4 \dots$
q_0	q_0	q_0	q_0	q_0	q_2	q_3	q_2	q_3	q_2	$q_3 \dots$

Acceptance Condition

Büchi automaton accepts an input if there is a run along which some subset of G occurs infinitely often [71]. Since G is a finite set, it is easy to see that there must actually be a state $g \in G$ that occurs infinitely often along σ. In other words, if we regard the state space of a Büchi automaton as a graph, an accepting run traces an infinite path that starts at some state $s \in S_{in}$, reaches a good state $g \in G$ and loops back to g infinitely often. In the infinite accepting run shown in Fig. 1.2, the move from state s to state g is the non-looping part, that we call *stem*, which ends in the *loop* at state g.

Example 1.1.4. *We illustrate bounded model checking in Figure. 1.1, where we would like to check the temporal property that the letter* a *occurs sometime in the future and reoccurs infinitely often. Now, this can occur, in a non-looping* stem *part of the run, or in the infinite* loop *of the run (cf.Fig. 1.2). Notice that the initial state* q_0 *does not accept the letter* a*. In order to*

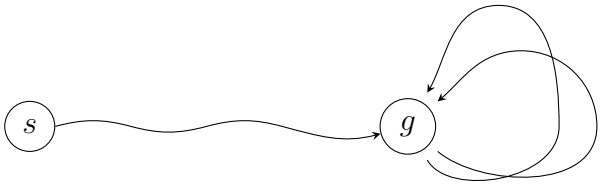


Figure 1.2: Accepting run of a Büchi automaton

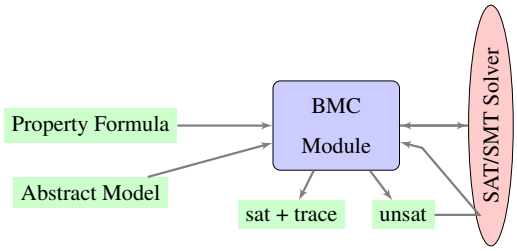


Figure 1.3: Bounded Model Checking

know, if there is some state, where a *is accepted in the future, one needs to look at the next states, apart from the initial state. Using the principle of bounded model checking, we check if* $\neg$ a *is accepted in a* stem *or in* loop *of all the runs possible in the model. The presence of an accepting run, acts as a counterexample to the property being checked. In this case, the sequence of states* $q_0, q_2, q_3, q_2, q_3, \ldots$ *is a counterexample trace. We can also view this as reaching a state, that accepts the said property.*

In Chapter. 3, we describe how to formally specify the properties in a formal manner using a logical language. One drawback is that we cannot explicitly represent concurrency in this model, hence we look at more suitable models to represent concurrency in unbounded client server systems via Petri nets, ν-nets (Petri nets with names), and Elementary Object Systems (Petri nets with nesting) in Chapter. 2. We revisit this notion of automatically performing bounded model checking, while verifying various models and properties in Sec. 5.2 and Sec. 6.2.1 using a set of specialized tools called SAT/SMT solvers. Given a set of constraints, such as in bounded model checking, SAT/SMT solvers can efficiently check if there exists a *satisfying assignment* of values to those constraints. SAT solvers work with Boolean values. SMT solvers offer various underlying theories such as bit vectors and integer arithmetic [17, 42].

1.2 Contributions

In this thesis, we focus on verification of unbounded client server systems and representing them by various suitable formal models as well as verifying the properties of the systems specified using various logics. In this system, there are unbounded interactions between the single server and unboundedly many clients that are not known apriori. The interactions are restricted to requests by the server that are responded to, by the clients. We have not worked with systems where there are interactions between the clients, nor have we worked with the multiple server multiple client systems. These extensions can be interesting future work. The major contributions of this work are the following:

1. **Verification of a Linear Temporal Logic with integer arithmetic properties on Petri nets using 2D-BMC:** Built a tool to verify Linear Temporal Logic with integer arithmetic properties on Petri nets. The tool employs SMT solvers to perform the verification queries using an extension of BMC, called the 2D-BMC algorithm. This tool competes with state of the art Petri net verification tools and is the first of its kind to use various semantics of Petri nets. (See Chapter. 5)

2. **Verification of a variant of First Order Logic on Petri nets using 2D-BMC:** Implemented a tool to verify properties of First Order Logic with monodic restriction on Petri nets with identifiers, called ν-nets and which employs SMT solvers to perform the verification queries using 2D-BMC algorithm. (See Chapter. 6)

3. **Verification of reachability of Elementary Object Systems:** The notion of reachability is to ask if a particular configuration can be reached. We implemented an open source tool to verify reachability properties on a class of higher order Petri nets with nesting, called Elementary Object Systems. (See Sec. 7.9)

4. **Charted the decidability status of Elementary Object Systems:** The notion of coverability is to ask, given a model and its configuration, is there a second reachable configuration that is smaller than the original configuration. These are important questions in Petri nets and also by extension, in Elementary Object Systems (EOS), which are essentially nets with nesting. In the interactions in the EOSs, imperfect steps are possible, which we call lossiness. Lossiness can occur at various levels of nesting in the EOSs. We

charted the decidability status of reachability and coverability problems in EOSs in the case of lossiness at various nesting levels. (See Sec. 7.8)

5. **Formulated the 2D-BMC algorithm**. We extend the standard bounded model checking algorithm to two dimensional bounded model checking, i.e, 2D-BMC that exploits the two dimensional unboundedness in the Petri nets representing the unbounded client server systems, i.e, unboundedness in the number of clients as well as unbounded concurrency. This is particularly helpful for verifying systems with true concurrency. (See Chapter. 4)

Chapter 2

Models for Unbounded Concurrency

While formally verifying infinite state systems with concurrency, it is imperative to capture their behaviour in suitable abstract models. In this chapter, we discuss the various abstractions for modeling concurrent systems that are used in the rest of the thesis.

2.1 Petri nets

Petri nets are a commonly used formalism for modeling the dynamic behaviour of systems [80, 83]. Here, we recollect the standard notions associated with nets.

Definition 2.1.1. *A* PN *is a tuple* $N = (P, T, F)$ *where* P *is a finite set of* places*,* T *is a finite set of* transitions*,* $F : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ *is the* flow function*.*

In case of weighted arcs, we also have the weight function $W : F \rightarrow \mathbb{N}_0$, where $\mathbb{N}_0$ is the set of non-negative integers. In Fig. 2.1, $P = \{p_0, p_1, p_2, p_3\}$, $T = \{t_0, t_1, t_2, t_3\}$ and F is represented graphically by the directed arcs.

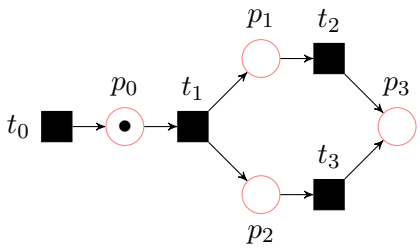


Figure 2.1: A simple Petri net with marking $\langle 1, 0, 0, 0 \rangle$

Definition 2.1.2. *The set* ${}^\bullet t = \{p \in P \mid F(p,t) > 0\}$ *is called the* set of pre-places *of* $t \in T$. *The set* $t^\bullet = \{p \in P \mid F(t,p) > 0\}$ *is called the* set of post-places *of* $t \in T$.

For instance, in Fig. 2.1, ${}^\bullet t_0 = \varnothing$, ${}^\bullet t_1 = \{p_0\}$, ${}^\bullet t_2 = \{p_1\}$, ${}^\bullet t_3 = \{p_2\}$. And $t_0{}^\bullet = \{p_0\}$, $t_1{}^\bullet = \{p_1, p_2\}$, $t_2{}^\bullet = t_3{}^\bullet = \{p_3\}$. The flow function F is also described by the pre and post condition functions.

Definition 2.1.3. *Given a PN* $N = (P, T, F)$, *the* pre-condition function $\mathtt{pre}_N$ *and the* post-condition function $\mathtt{post}_N$ *are defined as follows:*

$$\mathtt{pre}_N : T \to (P \to \mathbb{N}) \quad \mathtt{pre}_N(t)(p) = F(p,t)$$
$$\mathtt{post}_N : T \to (P \to \mathbb{N}) \quad \mathtt{post}_N(t)(p) = F(t,p)$$

The states of the PN are the distributions of tokens in the places, described by *markings*.

Definition 2.1.4. *A* marking M *of a PN* $N = (P, T, F)$ *is a function* $M : P \to \mathbb{N}_0$, *where* $\mathbb{N}_0$ *is the set of non-negative integers. A* marked PN *is a pair* $PN = (N, M_0)$ *where* N *is a PN and* M_0 *is a marking, called* initial marking.

In Fig. 2.1 the marking $\langle 1, 0, 0, 0 \rangle$, denotes the distribution of tokens in the places.

Definition 2.1.5 (Enabledness Rule)**.** *A transition* t *is enabled at marking* M *when* $\forall p \in P, M(p) \geqslant \mathtt{pre}_N(t,p)$.

The *enabledness rule* is a prerequisite for firing a transition t. On the firing of t, the successor of the current marking is obtained according to its pre and post conditions, i.e., by removing $F(p,t)$ tokens from each $p \in {}^\bullet t$ and adding $F(t,p')$ tokens to each $p' \in t^\bullet$, leaving tokens in the remaining places as they are.

Definition 2.1.6 (Firing Rule)**.** *Given a marking* M *in a Petri net, on firing an enabled transition* t, *we get the successor marking* M', *denoted by* $M \xrightarrow{t} M'$, *such that:*

$$\forall p \in P : M'(p) = M(p) - F(p,t) + F(t,p)$$

If there are several enabled transitions at a given marking, exactly one of them is non-deterministically chosen to be fired in the step.

Given the marking in Fig. 2.1 and on firing transition t_1, the marking in Fig. 2.1 is obtained.

Definition 2.1.7. *The set of markings* reachable *from marking M in a given net N, denoted by $reach_N(M)$, is the smallest set of markings such that:*

- $M \in reach_N(M)$ *and*
- *If* $M' \xrightarrow{t} M''$ *for some* $t \in T$ *and* $M' \in reach_N(M)$, *then* $M'' \in reach_N(M)$.

The reachability graph *of a PN N is the directed graph $(\mathcal{N}, E)$ where $\mathcal{N}$ is the set of markings of N, $E \subseteq \mathcal{N} \times \mathcal{N}$, and $(M, M') \in E$ iff there is some $t \in T$ such that $M \xrightarrow{t} M'$. The set of reachable markings of a marked PN (N, M_0), is $reach_N(M_0)$.*

Definition 2.1.8 (Conservative Petri net)**.** *A PN $N = (P, T, F)$ is* conservative *iff*

$$\forall p \in P \, \forall t \in T : F(p, t) = F(t, p)$$

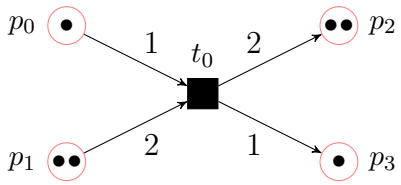


Figure 2.2: A conservative Petri net N_1

A Petri net N is said to be *conservative*, if all transitions fire token-preservingly, i.e, all transitions add exactly as many tokens to their post-places as they subtract from their pre-places. This is illustrated in Fig. 2.2.

Markings are naturally ordered by the coverability relation $\leqslant$, defined as follows.

Definition 2.1.9. *Given two markings M_1 and M_2, we say that a marking M_2 covers M_1, denoted by $M_1 \leqslant M_2$, iff $\forall p \in P, M_1(p) \leqslant M_2(p)$.*

In Petri net N_1 from Fig. 2.1 suppose the marking $M_0 = \langle 0, 0, 0, 0 \rangle$ and on firing t_0 we obtain $M_1 = \langle 1, 0, 0, 0 \rangle$, we say that M_1 covers M_0.

The *reachability problem* is as follows: given a marked PN (N, M_0) and a marking M, whether $M \in reach_N(M_0)$. For instance, in Fig. 2.2, given the initial marking, M_0 is $\langle 1, 2, 2, 1 \rangle$. We know that the marking $M_1 = \langle 0, 0, 4, 2 \rangle = reach_{N_1}(M_1)$. Hence, $M_1 \in reach_{N_1}(M_1)$.

It is known that the reachability problem for PN is *decidable* but in non-primitive recursive time [38]. The *coverability problem* is as follows: given a marked PN (N, M_0) and a marking M, whether there is a reachable marking $M' \in reach_N(M_0)$ such that $M' \leqslant M$. It is known

that also the coverability problem is decidable and in EXPSPACE [91]. Another problem on PN is *deadlock-freeness*.

Definition 2.1.10 (Dead Marking)**.** *A marking M for net N is* dead *if transition t is not enabled at M for all $t \in T$. The marked PN (N, M_0) is* deadlock-free*, if there is no dead marking in $reach_N(M_0)$.*

For instance, in Fig. 2.1, if the net is modified such that the transition t_0 is not present, it would result in a deadlock after obtaining the marking $\langle 0, 0, 0, 2 \rangle$. The *deadlock-freeness* problem is as follows: given a marked PN N, whether N is deadlock-free. It is well-known that deadlock-freeness and reachability are recursively equivalent and, thus, deadlock-freeness is decidable but in non-primitive recursive time complexity [33, 54].

2.1.1 Petri net Semantics

In this section, we look at the formal sematics of Petri nets, in particular the interleaving and concurrent semantics. The key difference in the two approaches, is in the firing condition. We recall the firing rule:

Given an enabled transition t at marking M, on firing t, we get the successor marking M', denoted by $M \xrightarrow{t} M'$, such that:

$$\forall p \in P : M'(p) = M(p) - F(p, t) + F(t, p)$$

In interleaving semantics, there is atmost one transition that is fired in an instance or step of the occurrence sequence. Given an initial marking $M_0 = \langle 0, 0, 0, 0 \rangle$, on firing t_0 twice and t_1 once, we obtain this sequence $\langle 0, 0, 0, 0 \rangle \xrightarrow{t_0} \langle 1, 0, 0, 0 \rangle \xrightarrow{t_0} \langle 2, 0, 0, 0 \rangle \xrightarrow{t_1} \langle 0, 1, 1, 0 \rangle$ as depicted in Fig. 2.3.

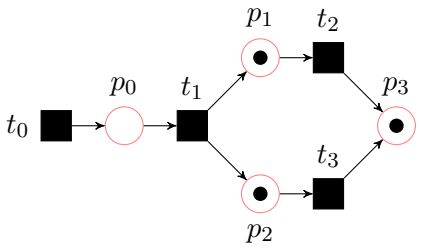


Figure 2.3: A simple Petri net N_0 with marking $\langle 0, 1, 1, 0 \rangle$

Now, both transitions t_2 and t_3 are enabled. In interleaving semantics, one may fire either of the two transitions, one after the other. However, in concurrent semantics, the following sequence is also additionally possible in a single step, resulting in Fig. 2.4.

$\langle 0, 1, 1, 0 \rangle \xrightarrow{t_2, t_3} \langle 0, 0, 0, 2 \rangle$

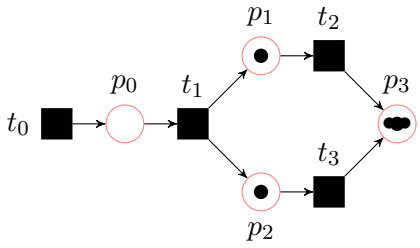


Figure 2.4: Petri net N_0 with marking $\langle 0, 0, 0, 2 \rangle$

Definition 2.1.11 (Concurrent Firing Rule)**.** *Given a Petri net N and a marking M having a set of enabled transitions $\tau \in T$, we obtain the successor marking M' on concurrent firing of a subset of transitions $\tau' \subseteq \tau$ using the following rule:*

$$\forall p \in {}^\bullet\tau' : M'(p) = M(p) - \sum_{t \in \tau'} F(p,t) + \sum_{t \in \tau'} F(t,p) \text{ if } \sum_{t \in \tau'} F(p,t) - M(p) \geqslant 0.$$

The condition $\sum_{t \in \tau'} F(p,t) - M(p) \geqslant 0$ takes care of the sufficiency of tokens at each of the pre-places of the transitions in τ' such that the transitions can be fired. This is a particularly elegant rule in the context of representing concurrent firing of transitions in a Petri net in a SMT solver.

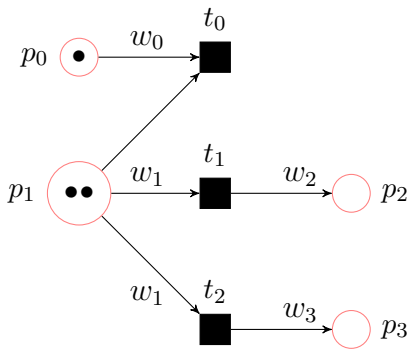


Figure 2.5: Concurrent Firing of Petri net N_1

In Fig. 2.5, $P = \{p_0, p_1, p_2, p_3\}$ and $T = \{t_0, t_1, t_2\}$. Given an initial marking $M_0 = \langle 1, 2, 0, 0 \rangle$, the set of enabled transitions w.r.t M_0, is $\tau = \{t_0, t_1, t_2\}$. Suppose $\tau' = \{t_1, t_2\}$, then

$^\bullet\tau' = \{p_1\}$ and the step $\langle 1,2,0,0\rangle \xrightarrow{t_1 \& t_2} \langle 0,0,1,1\rangle$ is allowed by the concurrent firing rule stated above. However, suppose $M_1 = \langle 1,1,0,0\rangle$, $\tau' = \{t_1, t_2\}$ and $^\bullet\tau' = \{p_1\}$, then the step $\langle 1,1,0,0\rangle \xrightarrow{t_1 \& t_2}$ is not allowed by the same rule.

2.2 Petri net Representation for an SMT Solver

The bigger picture is to verify properties of the Petri net using an SMT solver. In this section, we discuss the representation of nets such that they can be added as constraints in an SMT solver. The Petri net representation with interleaving semantics and the true concurrent semantics differ. While verifying infinite state systems, [5] employed backwards reachability for proving safety properties, and in case of concurrent programs, [6] proved liveness and termination via backwards reachability. In the unbounded Petri net setting, [3] is the last known work where unfoldings are discussed from a verification point of view. In state of the art Petri net verification tools, interleaving semantics is adopted [9]. We take an alternate perspective of this and unfold unbounded Petri nets with true concurrency using the help of SMT solvers.

We outline the variables and data structures that are necessary to describe the true concurrent semantics. We have a finite set of transitions, $t_0, \ldots, t_m$ described in the net, their names are in the list $tNames$. We have a finite set of places, $p_0, \ldots, p_l$ their names are in the list $pNames$. Arcs can be of either of two types: where the source is a transition and the target is a place or the source is a place and the target is a transition. The net formalism does not allow arcs between places and between transitions themselves. If they occur, the net description is erroneous, and we cannot move ahead with the unfolding.

We use the vector $iWTk$ to store the expressions with respect to k, to compute the incident weights of transitions. We use the vector WTk to store the expressions with respect to k, to denote the change in weights for places. We use two-dimensional matrices $Wt[m][l]$ and $iWt[m][l]$ to denote the net change in weights and the incidence weights (outgoing from places). The expressions for the same are stored in $WTVars$ and $iWTVars$ respectively.

We construct the two-dimensional weight matrix $Wt[row][col]$ of size $m \times l$ which contains the net weight of the arcs. Initially, all the matrix entries are initialized to zero.

For an arc from place p_i to transition t_j with the weight w, we have the matrix entry $Wt[i][j] = Wt[i][j] - w$. For an arc from transition t_j to place p_i with the weight w, we have the matrix entry $Wt[i][j] = Wt[i][j] + w$. Now, if there are incoming and outgoing arcs of the

same weights, then the net weight $Wt[i][j] = 0$. Notice that, by looking only at the Wt matrix, we may not distinguish between the case where there are no arcs to and from an element of the net. Hence, it is necessary to have a separate data structure for the same. We have a two-dimensional matrix $iWt[row][col]$ of size $m \times l$ which contains the incident weights to the transitions. For an arc from place p_i to transition t_j with the weight w, we have the matrix entry $iWt[i][j] = Wt[i][j] - w$. The marking of the net consists of the set tokens at each place and represents the state of the net at any instance. The initial marking of the net can be obtained from the net description and contains the number of tokens in each of the places $p_0, \ldots, p_l$ in the net. The subsequent markings may be constructed from the matrix Wt and using the transition function of the net. The transition function TF describes the behaviour of the net. The initial marking of the net is stored in $initial$. We introduce a method $printTFTruthTable$ that can aid to visualise the transition function TF which is an expression describing the function. Most utility methods and the $2 - DBMC$ algorithm are similar to that of the interleaving semantics. For experimentation, we have three different versions for constructing the Transition Function which are equivalent to each other (as verified by truth tables, experiments with 352 properties) and are a simplification of the expression using $\wedge\wedge$, $||$ instead of $\implies$ and so on. Experiments suggest that one version is slightly faster than the others. The variable $T[ti]$ denotes the ith transition being fired. Hence the expression $!T[ti]$ denotes that the i-th transition is not fired. For every pair of transitions and places, if there are no outgoing arc from the transition t_i then the expression $preCond$ containing the precondition for firing of transitions is constructed as follows:

$$
\begin{aligned}
& if(emptyOutTi)\{ && (2.1)\\
& \quad preCond = (tmp == iWt[pi][ti] \vee tmp == 0) && (2.2)\\
& \quad\quad \wedge ((T[ti] \wedge tmp == iWt[pi][ti]) && (2.3)\\
& \quad\quad \vee (!T[ti] \wedge tmp == 0)) && (2.4)\\
& cumulativeIncidentW = tmp && (2.5)\\
& emptyOutTi = !emptyOutTi\} && (2.6)\\
& \quad else\{ && (2.7)\\
& \quad preCond = preCond \wedge (tmp == iWt[pi][ti] \vee tmp == 0) && (2.8)\\
& \quad\quad \wedge ((T[ti] \wedge tmp == iWt[pi][ti]) && (2.9)\\
& \quad\quad \vee (!T[ti] \wedge tmp == 0)) && (2.10)
\end{aligned}
$$

$$cumulativeIncidentW = cumulativeIncidentW + tmp\} \tag{2.11}$$

The sum of incident weights at a transition t_i is stored in $cumulativeIncidentW$ as an expression of the $iWt[pi][ti]$ if there is an arc from the transition t_i to place p_i or it is zero. If the transition is not fired, then there are no weights to be considered.

If the expression is non-empty, the previously constructed $preCond$ are **anded** with the newly constructed expression and the cumulative incident weights are updated in the same manner.

The post condition is constructed if the weight $Wt[pi][ti]! = 0$.

$$\begin{aligned} postCond = &((tmp == Wt[pi][ti]) \vee (tmp == 0) \\ &\wedge ((T[ti] \wedge tmp == Wt[pi][ti]) \\ &\vee (!T[ti] \wedge tmp == 0)) \\ cumulativeWChange = &\ tmp \\ emptyChangePi = &!emptyChangePi \end{aligned}$$

Based on the above expressions, we construct the transition function TF

```
if (emptyTF){
        if (!emptyOutTi){
            TF = preCond & (Px[pi] + cumulativeIncidentW >= 0)
            emptyTF = false}
        if (!emptyChangePi){
            if (!emptyOutTi)
                TF = TF & postCond
                &(Py[pi] == Px[pi] + cumulativeWChange)
            else
                TF = postCond &(Py[pi] == Px[pi]
                 + cumulativeWChange)
            emptyTF = false
            }
    }
    else{
        if (!emptyOutTi)
```

```
            TF = TF &preCond
            &((Px[pi] + cumulativeIncidentW) >= 0)
        if (!emptyChangePi)
            TF = TF &postCond
             &(Py[pi] == (Px[pi] + cumulativeWChange))
    }
```

The transition function is a conjunction of the preconditions, postconditions and the change in the markings. In case of interleaving semantics, there is an additional conjunction to the transition function, a disjunction of each transition t_i, to ensure that exactly one transition is fired in a step.

2.2.1 ν-nets

In the rest of this thesis, we consider the specific setting of single server multiple client systems, with distinguishable clients. These distinguishable clients are represented using distinguishable tokens of the Petri net, i.e, tokens appended with identifiers [92]. There are an unbounded number of clients and a fresh client identifier is issued whenever a new client enters the system. When clients exit, the identifiers need to be purged. While the case study is discussed in detail in Chapter. 4, here, we outline the requirements for the formal model and arrive at a suitable representation.

As seen in the Sec. 2.1, Petri nets are suitable to model the *concurrent* behaviour of the clients and are also suitable to capture an unbounded number of clients. The places correspond to the local states of the client and server. We have a disjoint set of server places and client places. The combined interactions of the server and client processes are represented by transitions. The tokens correspond to the processes (server process, client process). Unbounded Petri nets with indistinguishable tokens are not sufficient to differentiate between processes (server process, client process). Hence we look for another model. At first glance, a candidate model is the colored Petri net (CPN) [63] which satisfies the above requirements. CPNs allow arbitrary expressions over user-defined syntax labelling the arcs, and the underlying modeling language (such as CPN Modeling Language in CPN Tools [62]) is highly expressive. However, we do not prefer the CPN, for the following engineering reasons.

First, there is a dearth of tools to automatically *unfold* unbounded colored Petri nets. Re-

call that the big picture is the automatic verification of unbounded client-server systems. Alternatively, suppose we represent the formal model as a CPN such as using CPN Tools, there are no existing tools that can automatically *unfold* an *unbounded* CPN created using CPN Tools. Existing tools can only unfold *bounded* CPNs [23, 39]. The second candidate model is a type of ν-net [93], which is a CPN defined over a system of component nets, which use a labelling function λ, to handle synchronization between multiple component nets of a larger net system.

We restrict the ν-nets to a single component, providing a simplified definition while doing away with the labelling function used in ν-nets. The client behaviour can be represented as a state machine. Similarly the server behaviour can also be represented as a state machine. In this representation, we describe the behaviour of the single server multiple client system as a single component.

We begin with some definitions that are necessary for describing the restricted ν-net. Given an arbitrary set A, we denote by $\mathcal{MS}(A)$, the set of finite multisets of A, given by the set of mappings $m : A \rightarrow \mathbb{N}$. We denote by $S(m)$ the support of m, defined as follows: $S(m) = \{a \in A \mid m(a) > 0\}$. Distinguishable tokens (identifiers) are taken from an arbitrary infinite set *Id*. To handle this, we add matching variables labeling the arcs, taken from a set Var. To handle the movement of tokens inside the ν-net, we employ a finite set of variables, Var using which we label the arcs of the net.

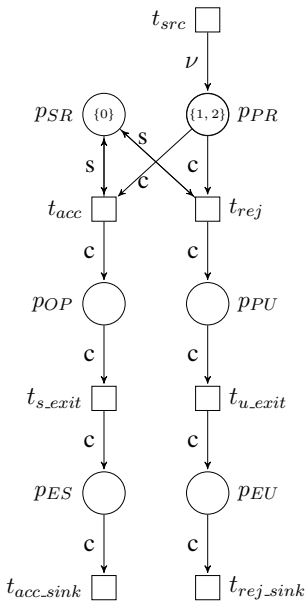


Figure 2.6: A restricted ν-net modeling APS

Definition 2.2.1. *A ν-net is a coloured Petri net $N = (P, T, F)$, where*

- *P and T are finite disjoint sets of places and transitions, respectively,*
- *F: $(P \times T) \cup (T \times P) \rightarrow \mathcal{MS}(Var)$ defines the set of arcs of the net, satisfying $\nu \notin pre(t)$ for every $t \in T$.*

For a transition t of the net, we define, $post(t) = \bigcup_{p \in P} S(F(t,p))$, $pre(t) = \bigcup_{p \in P} S(F(p,t))$ and $Var(t) = pre(t) \bigcup post(t)$.

For instance, in Fig. 2.6, $T = \{t_{acc}, t_{rej}, t_{s\_exit}, t_{u\_exit}, t_{acc\_sink}, t_{rej\_sink}\}$ and $P = \{ p_{PR}, p_{SR}, p_{OP}, p_{PU}, p_{ES}, p_{EU}\}$. $pre(t_{acc}) = \{s, c\}$ and $post(t_{acc}) = \{s, c\}$, hence $Var(t_{acc}) = \{s, c\}$.

Definition 2.2.2 (Marking)**.** *A marking of a restricted ν-net $N = (P, T, F)$ is a function $M : P \rightarrow (\mathcal{MS}(Id))$.*

In Fig. 2.6, the initial marking $M_0 = \langle \{0\}, \{1, 2\}, \varnothing, \varnothing, \varnothing, \varnothing \rangle$.

Definition 2.2.3 (Mode)**.** *We denote by $S(M)$ the set of identifiers in M. i.e, $S(M) = \bigcup_{p \in P} S(M(p))$. A mode of a transition t is a mapping $\sigma : Var(t) \rightarrow Id$, instantiating every variable in the adjacent arcs of t to some identifier.*

Let N be a restricted ν-net and M a marking of N according to Defn. 2.2.1.

Definition 2.2.4 (Enabling Rule)**.** *We say that M **enables** the transition t with mode σ whenever:*

- *If $\nu \in Var(t)$ then $\sigma(\nu) \notin S(M)$ and*
- *$\sigma(F(p,t)) \subseteq M(p)$ for all $p \in P$.*

Notice that if $\sigma(\nu) \notin S(M)$ for the enabling of transition, that causes the creation of fresh (equal) identifiers in all the places reached by arcs labelled by the special variable $\nu \in Var$ that appears only in post-condition arcs.

Definition 2.2.5 (Firing Rule)**.** *The reached marking of net N after firing of t with mode σ is denoted by $M \xrightarrow{t(\sigma)} M'$, where $\forall p \in P : M'(p) = M(p) - \sigma(F(p,t)) + \sigma(F(t,p))$.*

In Fig. 2.6, given initial marking $M = \langle\{0\}, \{1, 2\}, \varnothing, \varnothing, \varnothing, \varnothing\rangle$ and $M \xrightarrow{t_{acc}} M'$, $M' = \langle\{0\}, \{2\}, \{1\}, \varnothing, \varnothing, \varnothing\rangle$, i.e, the token $\{1\}$ has moved to the place P_{OP}. The mode is represented in the figure.

In Fig. 2.6, the transitions t_{acc}, t_{rej}, t_{s_exit}, t_{u_exit}, t_{acc_sink}, t_{rej_sink}, represent the accept, reject, exit successfully, exit unsuccessfully and the two sink transitions respectively. A transition is **identifier-preserving** if $post(t)\backslash\{\nu\} \subseteq pre(t)$. Here, all of them are *identifier-preserving* transitions, which ensures that the system with identified clients is represented correctly modeled. The firing of transition t_{src} acts as the source. The arc labelled ν ensures that a new client identifier is generated in place p_{PR}. The place p_{PR} contains a set of clients requesting for parking. In the unsuccessful scenario, the transition t_{rej} is fired when the server rejects the request, which brings the vehicle to *parking_unavailable* state represented by place p_{PU}. On firing of transition t_{u_exit}, the vehicle goes to *exited_unsuccessfully* state represented by place p_{EU}. The firing of transition t_{rej_sink} is the sink transition for the rejected parking requests. This ensures that the rejected vehicle identifier exits the system and is never reused. If the client arrives after it has exited, it is always issued a fresh identifier. Notice that there are arcs labelled s to indicate the server which has identifier 0, which is necessary for the acceptance or rejection of a parking request. The token with identifier 0 is permanently present in each marking exactly at server place p_{SR}. The ν arc ensures that new identifiers are generated, essentially giving an unbounded number of agents in the ν-net. The arcs labelled c carry the client identifiers from one client place to another. The net behaves as a standard ν-net component with autonomous transitions as described in [93]. We explored simple Petri nets, with identifiable tokens. Next, we shall explore an extension of Petri nets, suitable for modeling real world software systems.

2.2.2 Elementary object Systems

In [103, 104], they introduce the nets-within-nets paradigm, wherein, the tokens of the Petri net can be nets themselves. They are used to model the dynamic nature of token behaviour. This is synonymous with the object-oriented modeling approach introduced by Booch [94], which most software systems, including client-server systems follow.

In nets-within-nets, one can restrict the nesting of nets, to a nesting depth of two, to obtain Elementary Object Systems (EOSs). Here, the level 0 net is called the system net and the (nested) level 1 net is called the object net. In our running example, of the single server multiple clients, we represent server behaviour by the system net and the client behaviour by the object

nets that are nested within the system net. In EOSs, we have events, that enable firing of transitions in the system net or object net or at both levels, i.e system autonomous, object autonomous and synchronized events. Their formal definition is given below:

Definition 2.2.6 (EOS). *An* EOS $\mathfrak{E}$ *is a tuple* $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ *where:*

1. $\hat{N} = \langle \hat{P}, \hat{T}, \hat{F} \rangle$ *is a PN called* system net*;* $\hat{T}$ *contains a special set* $ID_{\hat{P}} = \{id_p \mid p \in \hat{P}\} \subseteq \hat{T}$ *of* idle transitions *such that, for each distinct* $p, q \in \hat{P}$*, we have* $\hat{F}(p, id_p) = \hat{F}(id_p, p) = 1$ *and* $\hat{F}(q, id_p) = \hat{F}(id_p, q) = 0$.
2. $\mathcal{N}$ *is a finite set of PNs, called* object PNs*, such that* $\blacksquare \in \mathcal{N}$ *and if* $(P_1, T_1, F_1), (P_2, T_2, F_2) \in \mathcal{N} \cup \hat{N}$,[1] *then* $P_1 \cap P_2 = \varnothing$ *and* $T_1 \cap T_2 = \varnothing$.
3. $d : \hat{P} \to \mathcal{N}$ *is called the* typing function.
4. Θ *is a finite* set of events *where each* event *is a pair* $(\hat{\tau}, \theta)$*, where* $\hat{\tau} \in \hat{T}$ *and* $\theta : \mathcal{N} \to \bigcup_{(P,T,F) \in \mathcal{N}} T^{\oplus}$*, such that* $\theta((P, T, F)) \in T^{\oplus}$ *for each* $(P, T, F) \in \mathcal{N}$ *and, if* $\hat{\tau} = id_p$*, then* $\theta(d(p)) \neq \varnothing$.

Definition 2.2.7 (Nested Markings)**.** *Let* $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ *be an EOS. The set of* nested tokens $\mathcal{T}(\mathfrak{E})$ *of* $\mathfrak{E}$ *is the set* $\bigcup_{(P,T,F) \in \mathcal{N}} (d^{-1}(P, T, F) \times P^{\oplus})$*. The set of* nested markings $\mathcal{M}(\mathbf{E})$ *of* $\mathfrak{E}$ *is* $\mathcal{T}(\mathfrak{E})^{\oplus}$*. Given* $\lambda, \rho \in \mathcal{M}(\mathbf{E})$*, we say that* λ *is a* sub-marking *of* μ *if* $\lambda \sqsubseteq \mu$.

Note that λ is a sub-marking of μ iff there is some nested marking μ' such that $\mu = \lambda + \mu'$. EOSs inherit the graphical representation of PNs with the provision that we represent nested tokens via a dashed line from the system net place to an instance of the object net where the internal marking is represented in the standard PN way. However, if the nested token is $\langle p, \varepsilon \rangle$ for a system net place p of type $\blacksquare$, we represent it with a black-token $\blacksquare$ on p. If a place p hosts $n > 2$ black-tokens, then we represent them by writing n on p. Each event $\langle \hat{\tau}, \theta \rangle$ is depicted by labeling $\hat{\tau}$ by $\langle \theta \rangle$ (possibly omitting double curly brackets). If there are several events involving $\hat{\tau}$, then $\hat{\tau}$ has several labels.

Example 1. Fig. 2.7 depicts the Petri net which *(1)* counts the parity of the a's and b's in the input word (firing sequence) *(2)* on seeing an a, the parity of the alphabet changes and the token moves to the place with a_O *(3)* on seeing an b, the parity of the alphabet changes and the token moves to the place with b_O. The marking $M_0 = \langle 1, 0, 0, 0 \rangle$ depicts the empty word, with exactly a token in place $a_E b_E$. For instance, given the firing sequences $\sigma = abba$, $M_0 \xrightarrow{\sigma} M_0$. Fig. 2.8 depicts the equivalent EOS which counts the parity of the a's and b's in the input word.

[1] This way, the system net and the object nets are pairwise distinct.

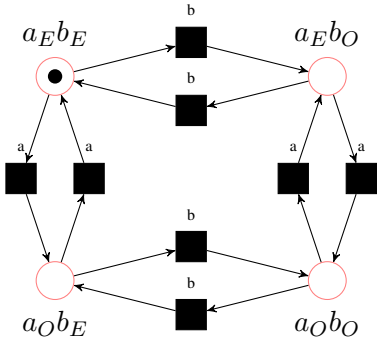


Figure 2.7: Petri net N_2 counting the parity of a and b in a word

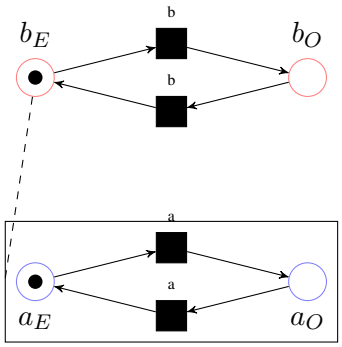


Figure 2.8: EOS counting the parity of a and b in a word

We adopt the technique in Example. 1, to the running example, single server multiple client systems.

Example 2. Fig. 2.9 depicts the client behaviour as a Petri net, where the places CR, OP, EU and ES represent *generate client request*, *occupy parking lot*, *exit unsuccessfully* and *exit successfully* respectively. Fig. 2.10 depicts the server behaviour as a Petri net where the places SR, ACR, RR represent *server ready*, *accept client request*, *reject request* respectively. The client behaviour can be modeled as an object net and the system places can model the server places, and we can obtain an EOS to model the combined behaviour of the single server multiple client system, where the clients are not identified, but an unbounded number of them mayh be generated. This is shown in Fig. 2.11. Notice that on firing the source transition, there are an unbounded number of object nets that can be geberated in the EOS. The server places are all of the same type, which enable to movement of the object tokens until exit.

Notice that the types of the system places are *minimal*, as there is exactly one type of object net, to depict the client type. Technically, $\mathcal{N} = \{\texttt{client}, \blacksquare\}$ (even if $\blacksquare$ is unused), $d(\texttt{SR}) = d(\texttt{ACR}) = d(\texttt{RR}) = \texttt{client}$, and Θ synchronizes `accept`,

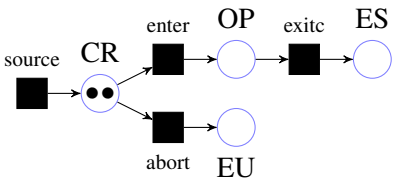


Figure 2.9: Petri net N_3 depicting client behaviour

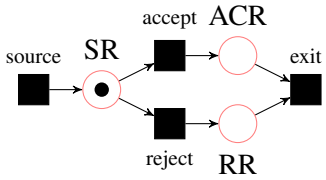


Figure 2.10: Petri net N_3 depicting server behaviour

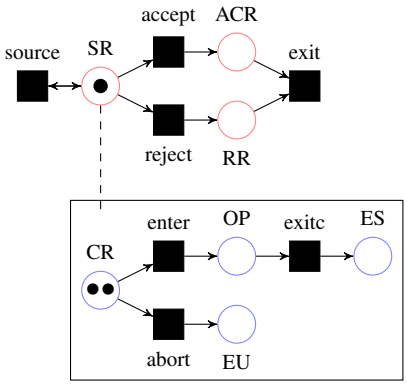


Figure 2.11: EOS modeling the single server (unbounded) multiple client system

`reject` and `exit` in $\hat{N}$ with `enter`, `abort` and `exit` in `client`. Formally, $\Theta = \{\langle \texttt{accept}, \{\{\texttt{enter}\}\}\rangle, \langle \texttt{reject}, \{\{\texttt{abort}\}\}\rangle, \langle \texttt{exit}, \{\{\texttt{exitc}\}\}\rangle\}$.

The marking $\mu = \langle \texttt{client}, \{\{\texttt{client1}, \texttt{client2}\}\}\rangle$ represents a server instance at `SR`, with two client requests, waiting to be serviced.

When firing an event $\langle \tau, \theta \rangle$, nested tokens in the system net are consumed according to the preconditions of τ in the standard PN way. At the same time, for each object net N, the inner tokens are merged so as to obtain a PN marking $\mu(N)$ for N (possibly empty). Then, transitions in $\theta(N)$ are fired in the standard PN way obtaining markings $\mu'(N)$. Next, nested markings with empty inner markings are produced in the system net according to the postconditions of τ. Finally, the markings $\mu'(N)$ are non-deterministically distributed among the empty nested tokens, according to the typing function. To be fired, the event must be enabled at both the system and at the object net level. This is captured by the enabledness condition, which makes

use of projection operators at the system (Π^1) and at the object net level (Π^2_N for each $N \in \mathcal{N}$).

Definition 2.2.8 (Projection Operators)**.** *Let $\mathfrak{E}$ be an EOS $\langle \hat{N}, \mathcal{N}, d, \Theta \rangle$. The* projection operators Π^1 *maps each nested marking $\mu = \sum_{i \in I} \langle \hat{p}_i, M_i \rangle$ for $\mathbf{E}$ to the PN marking $\sum_{i \in I} \hat{p}_i$ for $\hat{N}$. Given an object net $N \in \mathcal{N}$, the* projection operators Π^2_N *maps each nested marking $\mu = \sum_{i \in I} \langle \hat{p}_i, M_i \rangle$ for $\mathbf{E}$ to the PN marking $\sum_{j \in J} M_j$ for N where $J = \{i \in I \mid d(\hat{p}_i) = N\}$.*

To define the enabledness condition, we need the following notation. We set $\mathtt{pre}_N(\theta(N)) = \sum_{i \in I} \mathtt{pre}_N(t_i)$ where $(t_i)_{i \in I}$ is an enumeration of $\theta(N)$ counting multiplicities. We analogously set $\mathtt{post}_N(\theta(N)) = \sum_{i \in I} \mathtt{post}_N(t_i)$.

Definition 2.2.9 (Enabledness Condition)**.** *Let $\mathfrak{E}$ be an EOS $\langle \hat{N}, \mathcal{N}, d, \Theta \rangle$. Given an event $e = \langle \hat{\tau}, \theta \rangle \in \Theta$ and two markings $\lambda, \rho \in \mathcal{M}(\mathfrak{E})$, the* enabledness condition $\Phi(\langle \hat{\tau}, \theta \rangle, \lambda, \rho)$ *holds iff*

$$\Pi^1(\lambda) = \mathtt{pre}_{\hat{N}}(\hat{\tau}) \ \wedge \Pi^1(\rho) = \mathtt{post}_{\hat{N}}(\hat{\tau}) \ \wedge \forall N \in \mathcal{N},\ \Pi^2_N(\lambda) \geqslant \mathtt{pre}_N(\theta(N)) \ \wedge$$
$$\forall N \in \mathcal{N},\ \Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathtt{pre}_N(\theta(N)) + \mathtt{post}_N(\theta(N))$$

The event e is enabled with mode (λ, ρ) on a marking μ *iff $\Phi(e, \lambda, \rho)$ holds and $\lambda \sqsubseteq \mu$. Its firing results in the step $\mu \xrightarrow{(e,\lambda,\rho)} \mu - \lambda + \rho$.*

Notice that in Fig. 2.11, the transitions `accept` and `reject` are both enabled due to two client object nets in place SR and the object net places `enter`, `abort` are enabled as well. Depending on the fired transition in the system net, the corresponding synchronizing move is taken in the object net as well, in the synchronizing event.

Definition 2.2.10 (Reachability in EOS)**.** *The reachability problem for EOSs is defined in the usual way, i.e., whether there is a run (sequence of event firings) from an initial marking μ_0 to a target marking μ_f. $\mu_1 = \langle \mathtt{client}, \{\{\mathtt{client1}, \mathtt{client2}\}\} \rangle$ is a reachable marking.*

Definition 2.2.11 (Coverability in EOS)**.** *The coverability definition is standard, but with respect to the order $\preceq$ (in [65]), the component wise ordering relation. For instance, $\mu_1 = \langle \mathtt{client}, \{\{\mathtt{client1}, \mathtt{client2}\}\} \rangle$ and $\mu_2 = \langle \mathtt{client}, \{\{\mathtt{client1}, \mathtt{client2}, \mathtt{client3}\}\} \rangle$. Then, $\mu_1 \preceq \mu_2$.*

Conclusion

We make use of the above formal models for concurrency in the rest of this thesis. In order to specify their properties for verfying using bounded model checking, we need to explore how to

represent the properties in a manner that is correct as well as suitably written into formulas that can be fed to SAT/SMT solvers for verification.

Chapter 3

Logics for Unbounded Concurrency

Reactive systems are a family of systems where there is continuous interaction between the system and its environment. Autonomous vehicles that steer according to traffic and obstacles, operating systems that handle scheduling of processes based on resource utilization, robots that navigate their terrain using various inputs are examples of reactive systems. A reactive program such as an autonomous navigation system can be viewed as an abstract function from an input domain to an output domain whose behaviour consists of a transformation from initial states to final states. Typically, they do not terminate. In the context of verification of infinite state reactive programs, in order to specify the properties of programs, the properties need to be expressed in the form of a logic formula. Owing to their non-terminating nature, we need a mechanism for talking about the way the system evolves along potentially infinite computations. In this chapter, we discuss the various logics used in the rest of the thesis.

3.1 Introduction to Linear Temporal Logic

Temporal logic has become a well-established formalism for writing specifications. Many varieties of temporal logic have been defined in the past, we focus on linear time temporal logic (LTL). There is a close connection between models of LTL formulas and languages of infinite words—the models of an LTL formula constitute an ω-regular language over an appropriate alphabet. As a result, the satisfiability problem for LTL reduces to checking for emptiness of ω-regular languages [107]. Later, in [106], the connection between LTL and ω-regular languages has been extended to model checking. Unlike the satisfiability problem, which asks if a given formula α has a model, the model-checking problem is one of verification: the task is to verify

whether a given finite-state program P satisfies a specification α. This consists of checking that all runs of P constitute models for α. Since finite-state reactive programs can be represented quite naturally as Büchi automata, model checking also reduces to a problem in automata theory. It suffices to show that no run of P is a model for $\neg\alpha$, which is the same as checking that the intersection of the language accepted by P and the language defined by $\neg\alpha$ is empty. We shall explore the model checking algorithm in detail in Chapter. 4. First, we shall see how to express the properties in LTL.

3.1.1 Specifying properties

Linear temporal logic is very useful in specifying system properties. We consider atomic properties $p_1, p_2, \ldots$ of the system (or program) to be verified. Suppose p_1 and p_1 are atomic propositions in the context of a reactive system. We can formulate specifications of this (generic) system in LTL as follows:

Guarantee:	Sometime p_1 becomes true
Safety:	Always p_1 is true
Periodicity :	Initially, p_1 is true and p_1 is true precisely at every third moment
Obligation :	Sometime p_1 is true but p_2 is never true.
Recurrence :	Again and again, p_1 is true
Request-Response:	Always when p_1 is true, p_2 will be true sometime later
Until :	Always when p_1 is true, sometime later p_1 will be true again and in the meantime p_2 is always true
Fairness :	If p_1 is true again and again, then so is p_2.

We reformulate these specifications by using the temporal operators. Their syntax and semantics are explained shortly afterwards.

- Gp for *always (from now onwards) p is true*
- Fp for *eventually (sometime, including present) p is true*
- Xp for *p is true next time*
- $p_1 U p_2$ for p_1 *is true until eventually* p_2 *is true*

Now, the above specifications are rewritten using LTL as follows:

Fp_1	Sometime p_1 becomes true
Gp_1	Always p_1 is true. (See Figure 3.1).
$(p_1 \wedge X\neg p_1 \wedge XX\neg p_1 \wedge G(p_1 \iff XXXp_1))$	Initially, p_1 is true and p_1 is true precisely at every third moment.
$Fp_1 \wedge \neg Fp_2$	Sometime p_1 is true but p_2 is never true. (See Figure 3.2).
GFp_1	Again and again, p_1 is true
$G(Fp_1 \rightarrow XFp_2)$	Always when p_1 is true, p_2 will be true sometime later.
$G(p_1 \rightarrow X(p_2Up_2))$	Always when p_1 is true, sometime later p_1 will be true again and in the meantime p_2 is always true.
$GFp_1 \rightarrow GFp_2$	If p_1 is true again and again, then so is p_2. (See Figure 3.3).

The LTL formulas can be translated into Büchi automata.

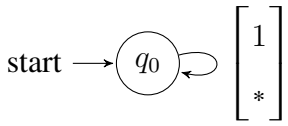


Figure 3.1: Büchi automaton for Gp_1

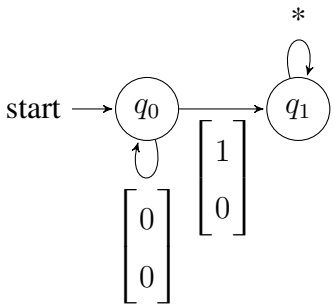


Figure 3.2: Büchi automaton for Fp_1

Next, we discuss the syntax of LTL.

3.1.2 Syntax

The LTL formulas over atomic boolean propositions $V = \{p_1, \ldots, p_n\}$ are inductively defined as follows:

- p_i is a LTL formula.

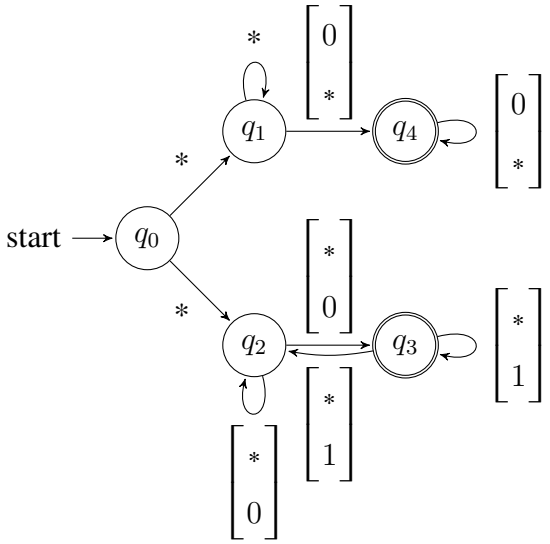


Figure 3.3: Büchi automaton for $GFp_1 \rightarrow GFp_2$

- If ϕ, ψ are LTL formulas, then so are $\neg\phi, \phi \vee \psi, \phi \wedge \psi, \phi \rightarrow \psi$
- If ϕ, ψ are LTL formulas, $X\psi, F\phi, G\phi, \phi U\psi$ where X is the next operator, F denotes sometime, and G denotes globally.

The system contains a set of states S, a set of initial states $I \subseteq S$ and transition relation $T \subseteq S \times S$. The interpretation of propositional variables may change over time but is determined by the current state of the system. This is denoted by the labelling function $L : S \rightarrow P(V)$, where S is the set of states. A propositional variable p is true in a system state s iff $p \in L(s)$. We fix one Kripke structure $K = (S, I, T, L)$ over the variables V . Given the state s is a vector containing all variables V in the current state and s' is a vector of their primed copies in the successor state, the transition relation is written as $T(s, s')$ which is interpreted as $T(s, s')$ holds iff there is a transition from $s \rightarrow s'$.

3.1.3 Semantics

The semantics of LTL are defined along paths of the model. A path π is an infinite sequence of states $\pi = (s_0, s_1, s_2, \dots)$ where $s_i \rightarrow s_{i+1}$. A path π is initialized if its first state $\pi(0) = s_0$ is an initial state. An LTL formula f holds along a path π written as $\pi \models f$ iff

- $\pi \models p$ iff $p \in L(\pi(0))$
- Similarly $\pi \models \neg p$ iff $p \notin L(\pi(0))$

- $\pi \models g \vee h$ iff $\pi \models g$ or $\pi \models h$
- $\pi \models g \wedge h$ iff $\pi \models g$ and $\pi \models h$
- $\pi \models Fg$ iff $\exists j \in \mathbb{N} : \pi^j \models g$
- $\pi \models Gg$ iff $\forall j \in \mathbb{N} : \pi^j \models g$
- $\pi \models Xg$ iff $\pi^1 \models g$

In this context, the model checking problem is to determine whether kripke structure $K \models f$ holds. A formula f has a witness in K iff there is an initialized path π where $\pi \models f$. Hence, $K \models f$ iff $\neg f$ doesnt have a witness in K. Therefore, we can reduce the model checking problem to the search for witnesses using negation and translation into Negation Normal Form, where the negations are pushed to the variables as follows:

- $\neg(g \wedge h) \equiv (\neg g) \vee (\neg h)$
- $\neg Fg \equiv G\neg g$
- $\neg Gg \equiv F\neg g$
- $\neg X \equiv X\neg g$

3.1.4 Bounded Semantics of LTL

It is observed that some infinite paths can be represented by a finite prefix with a loop. An infinite path π is a (k, l) lasso, iff $\pi(k + 1 + j) = \pi(l + j), \forall j \in \mathbb{N}$. This is represented in Figure 3.4. The path can be represented as $\pi = \pi_{stem}.\pi_{loop}^{\omega}$. In case of the finite system K, the search for witness can be restricted to lassos.

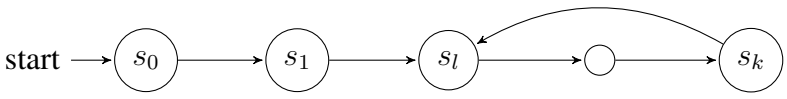


Figure 3.4: (k, l) lasso with l=2, k=4

We can rewrite the semantics given above as follows:

- $\pi^i \models p$ iff $p \in L(\pi(i))$
- Similarly $\pi^i \models \neg p$ iff $p \notin L(\pi(i))$

- $\pi^i \models g \vee h$ iff $\pi^i \models g$ or $\pi^i \models h$
- $\pi^i \models g \wedge h$ iff $\pi^i \models g$ and $\pi^i \models h$
- $\pi^i \models Fg$ iff $\exists j \in \mathbb{N} : \pi^{i+j} \models g$
- $\pi^i \models Gg$ iff $\forall j \in \mathbb{N} : \pi^{i+j} \models g$
- $\pi^i \models Xg$ iff $\pi^{i+j} \models g$

In order to obtain bounded semantics, we need to observe only the first $k+1$ states, and let $i \in 0 \dots k$. If π is (k, l) lasso, then $\pi(k+1+j) = \pi(l+j), \forall j \in \mathbb{N}$. Hence, we have the following:

- $\pi^i \models Fg$ iff $\exists j \in \min(i, l), \dots, k : \pi^j \models g$
- $\pi^i \models Gg$ iff $\forall j \in \min(i, l), \dots, k : \pi^{i+j} \models g$
- $\pi^i \models Xg$ iff $\pi^{i+l} \models g$ if $i < k$
- $\pi^i \models Xg$ iff $\pi^l \models g$ if $i = k$

Suppose our property to be tested is a safety property Gp, we need a witness for $F\neg p$. If p doesnt hold in some initial state s in system K, $k = 0$ is sufficient.

If π is not a (k, l) lasso, for any l, and we do not want to examine suffix beyond bound k, then we cannot draw conclusions about $\pi^k \models Xg$ nor about $\pi^k \models Gg$ for $i < k$.

3.1.5 Encoding LTL Semantics

This encoding [22] of the LTL witness problem to SMT uses bounded semantics. The encoding can be implemented as a recursive procedure that takes an LTL formula f, a fixed bound k, the loop start l, and the position i as parameters, where $l, i \in [0, \dots k]$. The output of this procedure is ${}_l[f]^i_k$. In case of a (k, l) loop in the model.

${}_l[p]^i_k \equiv p_i$ ${}_l[\neg p]^i_k \equiv \neg p_i$

${}_l[g \vee h]^i_k \equiv {}_l[g]^i_k \vee {}_l[h]^i_k$ ${}_l[g \wedge h]^i_k \equiv {}_l[g]^i_k \wedge {}_l[h]^i_k$

${}_l[Fg]^i_k \equiv \bigvee^k_{\min}(l, i)\ {}_l[g]^i_k$ ${}_l[Gg]^i_k \equiv \bigwedge^k_{\min}(l, i)\ {}_l[g]^i_k$

${}_l[Xg]^i_k \equiv {}_l[g]^i_k$ with $j = i+1$ if $i < k$ else $j = l$

In the absence of loops, we use the following encoding:

$[Fg]^i_k \equiv \bigvee^k_{j=i} [g]^i_k \quad [Gg]^i_k \equiv \bot \quad [Xg]^i_k \equiv [g]^{i+1}_k \text{if} i < k \text{ else} \bot \text{ if } j = k$

The full encoding is given as: $[f]_k \equiv [f]_k^0 \vee \bigvee_{l=0}^{k} \wedge_l [f]_k^0$.

Using the encoding, we can find a witness for a particular bound k. In case the formula is satisfiable, we are certain to have found a witness. If the formula is unsatisfiable, we can increment k and look for a witness of greater length. This leads to the natural question, of when we should stop increasing k, when no witness has been found yet. We discuss the algorithm in greater detail in the later chapters.

3.2 Counting LTL

Linear Temporal Logic (LTL) [87, 106] is a natural choice to describe the temporal properties of such systems. We are interested in Logic LTL_{LIA} as it is easy to specify temporal properties as well as invariants in it. With respect to Fig. 2.6, consider the invariant, where there is exactly one token in either place p_{SR} or p_{SB} or p_{RR}. This is easily expressed in LTL_{LIA} as $G(\#p_{SR} + \#p_{SB} + \#p_{RR} = 1)$.

Logic LTL_{LIA} is an extension of LTL with a few differences. In the case of LTL, atomic formulas are propositional constants which have no further structure. In LTL_{LIA}, there are two types of atomic formulas: (1) P_s - describing basic server properties, which are propositional constants and can be used to describe the transitions of the net (2) $\beta \in \Delta$, the set of client formulas, which is formally given by:

$$\alpha, \hat{\alpha} ::= \#p \mid \mathfrak{c} \mid (\mathfrak{c} * \#p) \mid (\#p * \mathfrak{c}) \mid (\alpha + \hat{\alpha}) \mid (\alpha - \hat{\alpha})$$
$$\beta ::= (\alpha < \hat{\alpha}) \mid (\alpha > \hat{\alpha}) \mid (\alpha \leqslant \hat{\alpha}) \mid (\alpha \geqslant \hat{\alpha}) \mid (\alpha = \hat{\alpha})$$

where $p \in P_c$, the set of client propositions and $\mathfrak{c} \in \mathbb{Z}^+$.
The set of server formulas Ψ are defined as follows:

$$\psi ::= q \in P_s \mid \beta \in \Delta \mid \neg\psi \mid \psi \vee \psi' \mid \psi \wedge \psi' \mid X\psi \mid F\psi \mid G\psi \mid \psi U \psi'$$

where $\psi, \psi' \in \Psi$. Modalities X, F, G and U are the usual modal operators: Next, Eventually, Globally and Until, respectively.

Unbounded Semantics

The logic LTL_{LIA} is interpreted over model sequences. Let CN be a countable set of client names that can be assigned to the processes in the system. Formally, a model is a sequence

$\varrho = m_0, m_1, \ldots$, where for all $i \geqslant 0$ m_i is a triple (ν_i, V_i, ξ_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. Further, for all $i \geqslant 0$, $V_{i+1} \subseteq V_i$ or $V_i \subseteq V_{i+1}$ which indicates that the clients can enter or exit the system dynamically.
3. $\xi_i : V_i \rightarrow 2^{P_c}$ gives the properties satisfied by each live agent at the ith instant.

The evaluation of the set of terms α at time instance i is derived from m_i and given by the following function: $||\alpha||_i : \alpha \times \mathbb{N} \rightarrow \mathbb{N}$. This is defined inductively as follows:

Base case: The term $\#p$ evaluates to a non-negative integer, $||\#p||_i$ denoting the number of clients satisfying property p at instance i where $||\#p||_i = \{a \in V_i \mid p \in \xi_i(a)\}$. The evaluation of $\mathfrak{c}$ at instance i, $||\mathfrak{c}||_i = \mathfrak{c}$.

Inductive case: The terms $(\mathfrak{c}\#p)$ (and $(\#p\mathfrak{c})$) evaluates to $\mathfrak{c}||\#p||_i$ (and $||\#p||_i\mathfrak{c}$ respectively) the multiplication of an integer denoted by $\#p$ with a constant $\mathfrak{c}$. The terms $(\alpha + \hat{\alpha})$ and $(\alpha - \hat{\alpha})$ evaluate to addition and subtraction of integers $||\alpha||_i, ||\hat{\alpha}||_i$.

The truth of a formula at an instant in the model is given by the $\models$ relation defined by induction over the structure of ψ as follows:

1. $\varrho, i \models (\alpha < \hat{\alpha})$ iff $||\alpha||_i < ||\hat{\alpha}||_i$.
2. $\varrho, i \models (\alpha > \hat{\alpha})$ iff $||\alpha||_i > ||\hat{\alpha}||_i$.
3. $\varrho, i \models (\alpha \leqslant \hat{\alpha})$ iff $||\alpha||_i \leqslant ||\hat{\alpha}||_i$.
4. $\varrho, i \models (\alpha \geqslant \hat{\alpha})$ iff $||\alpha||_i \geqslant ||\hat{\alpha}||_i$.
5. $\varrho, i \models (\alpha = \hat{\alpha})$ iff $||\alpha||_i = ||\hat{\alpha}||_i$.
6. $\varrho, i \models q$ 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.
7. $\varrho, i \models \neg\psi$ iff $\varrho, i \not\models \psi$.
8. $\varrho, i \models \psi \vee \psi'$ iff $\varrho, i \models \psi$ or $\varrho, i \models \psi'$.
9. $\varrho, i \models \psi \wedge \psi'$ iff $\varrho, i \models \psi$ and $\varrho, i \models \psi'$.
10. $\varrho, i \models X\psi$ iff $\varrho, i + 1 \models \psi$.

11. $\varrho, i \models F\psi$ iff $\exists j \geqslant i$, $\varrho, j \models \psi$.

12. $\varrho, i \models G\psi$ iff $\forall j \geqslant i$, $\varrho, j \models \psi$.

13. $\varrho, i \models \psi U \psi'$ iff $\exists j \geqslant i$, $\varrho, j \models \psi'$ and for all $i \leqslant j' < j : \varrho, j' \models \psi$.

Bounded Semantics

We provide the bounded semantics [21] of LTL_{LIA} in order to arrive at the SMT encoding in section 5.2 which is necessary for BMC. We use $\models^{\langle \lambda, \kappa \rangle}$ as an restriction of $\models$ over bounded runs of length λ. The bound k is divided into two parts: $k = \kappa + \lambda$, where κ is a bound on the number of clients and λ is a bound on the execution steps of the net. Since the runs are bounded, there are at most κ agents in the system, namely CN = $\{0, \ldots, \kappa - 1\}$.

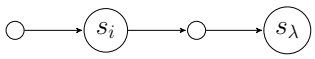


Figure 3.5: Bounded loop-free path of length λ

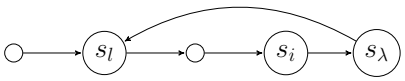


Figure 3.6: Bounded path with (λ, l) - loop

First, we describe the bounded semantics without loop where $0 \leqslant i \leqslant \lambda$:

1. $\varrho, i \models^{\langle \lambda, \kappa \rangle} (\alpha < \hat{\alpha})$ iff $||\alpha||_i < ||\hat{\alpha}||_i$.

2. $\varrho, i \models^{\langle \lambda, \kappa \rangle} (\alpha > \hat{\alpha})$ iff $||\alpha||_i > ||\hat{\alpha}||_i$.

3. $\varrho, i \models^{\langle \lambda, \kappa \rangle} (\alpha \leqslant \hat{\alpha})$ iff $||\alpha||_i \leqslant ||\hat{\alpha}||_i$.

4. $\varrho, i \models^{\langle \lambda, \kappa \rangle} (\alpha \geqslant \hat{\alpha})$ iff $||\alpha||_i \geqslant ||\hat{\alpha}||_i$.

5. $\varrho, i \models^{\langle \lambda, \kappa \rangle} (\alpha = \hat{\alpha})$ iff $||\alpha||_i = ||\hat{\alpha}||_i$.

6. $\varrho, i \models^{\langle \lambda, \kappa \rangle} q$ iff $q \in \nu_i$.

7. $\varrho, i \models^{\langle \lambda, \kappa \rangle} \neg\psi$ iff $\varrho, i \not\models^{\langle \lambda, \kappa \rangle} \psi$.

8. $\varrho, i \models^{\langle \lambda, \kappa \rangle} \psi \vee \psi'$ iff $\varrho, i \models^{\langle \lambda, \kappa \rangle} \psi$ or $\varrho, i \models^{\langle \lambda, \kappa \rangle} \psi'$.

9. $\varrho, i \models^{\langle\lambda,\kappa\rangle} \psi \wedge \psi'$ iff $\varrho, i \models^{\langle\lambda,\kappa\rangle} \psi$ and $\varrho, i \models^{\langle\lambda,\kappa\rangle} \psi'$.

10. $\varrho, i \models^{\langle\lambda,\kappa\rangle} X\psi$ iff $\begin{cases} \varrho, i+1 \models^{\langle\lambda,\kappa\rangle} \psi & \text{if } (i < \lambda) \\ \varrho, i \not\models^{\langle\lambda,\kappa\rangle} X\psi & \text{otherwise} \end{cases}$

    When instance i is less than the bound λ, the formula ψ is evaluated at instance $i+1$ else, the formula is unsatisfiable.

11. $\varrho, i \models^{\langle\lambda,\kappa\rangle} F\psi$ iff $\exists j : i \leqslant j \leqslant \lambda$, $\varrho, j \models^{\langle\lambda,\kappa\rangle} \psi$.

    This formula is satisfiable if there is some instance j such that $i \leqslant j \leqslant \lambda$ at which the property ψ holds.

12. $\varrho, i \not\models^{\langle\lambda,\kappa\rangle} G\psi$.

    In the absence of a loop in the bounded run, the above formula is always unsatisfiable.

13. $\varrho, i \models^{\langle\lambda,\kappa\rangle} \psi U \psi'$ iff $\exists j : i \leqslant j \leqslant \lambda$, $\varrho, j \models^{\langle\lambda,\kappa\rangle} \psi'$ and for all $j' : i \leqslant j' < j : \varrho, j' \models^{\langle\lambda,\kappa\rangle} \psi$.

    This formula is satisfied when formula ψ' is satisfied at some instance j and for all instances less than j, formula ψ is satisfied.

Second, we describe the bounded semantics with (λ, l)-loop [21] where $0 \leqslant i \leqslant \lambda$ and $0 \leqslant l \leqslant \lambda$ as in Fig. 3.6. Semantics and explanations are given only where they differ from the corresponding case without loop:

10. $\varrho, i \models^{\langle\lambda,\kappa\rangle} X\psi$ iff $\begin{cases} \varrho, i+1 \models^{\langle\lambda,\kappa\rangle} \psi & \text{if } (i < \lambda) \\ \varrho, l \models^{\langle\lambda,\kappa\rangle} \psi & \text{otherwise} \end{cases}$

    When instance i is less than the bound λ, the formula ψ is evaluated at instance $i+1$ else, ψ is evaluated at instance l, which is the next instance of λ.

11. $\varrho, i \models^{\langle\lambda,\kappa\rangle} F\psi$ iff $\exists j : min(l, i) \leqslant j \leqslant \lambda$, $\varrho, j \models^{\langle\lambda,\kappa\rangle} \psi$.

    This formula is satisfiable if there is some instance j such that $min(l, i) \leqslant j \leqslant \lambda$, where the formula ψ is satisfied.

12. $\varrho, i \models^{\langle\lambda,\kappa\rangle} G\psi$ iff $\forall j : min(l, i) \leqslant j \leqslant \lambda$, $\varrho, j \models^{\langle\lambda,\kappa\rangle} \psi$.

This formula is satisfiable if for all instances j such that $min(l, i) \leqslant j \leqslant \lambda$, the formula ψ is satisfied in each instance.

13. $\varrho, i \models^{\langle\lambda,\kappa\rangle} \psi U \psi'$ iff
$$\begin{cases} \exists j : i \leqslant j \leqslant \lambda, \varrho, j \models^{\langle\lambda,\kappa\rangle} \psi' \text{ and} & \text{if } (i \leqslant l) \\ \forall j' : i \leqslant j' < j : \varrho, j' \models^{\langle\lambda,\kappa\rangle} \psi & \\ \exists j : i \leqslant j \leqslant \lambda, \varrho, j \models^{\langle\lambda,\kappa\rangle} \psi' \text{ and} & \text{if } (i > l) \\ \forall j' : i \leqslant j' < j : \varrho, j' \models^{\langle\lambda,\kappa\rangle} \psi & \\ \text{or} & \\ \exists j : l \leqslant j < i, \varrho, j \models^{\langle\lambda,\kappa\rangle} \psi' \text{ and} & \\ \forall j' : l \leqslant j' < j : \varrho, j' \models^{\langle\lambda,\kappa\rangle} \psi & \end{cases}$$

Consider the two cases:

- If $(i \leqslant l)$, the current instance i is less than or equal to the loop instance l, this formula is satisfied when formula ψ' is satisfied at some instance j such that $i \leqslant j \leqslant \lambda$ and for all instances between i and j, formula ψ is satisfied.
- If $(i > l)$, the formula may be satisfied in either of the two intervals between i to λ or between l to i. Hence, the formula is satisfied if either of the following are satisfied: formula ψ' is satisfied at some instance j such that $i \leqslant j \leqslant \lambda$ and for all instances less than j, formula ψ is satisfied or, formula ψ' is satisfied at some instance j such that $l \leqslant j < i$ and for all instances between l and j, formula ψ is satisfied.

3.3 First Order Logic with Monodic restriction

The monodic logic FOTL_1 is a syntactic subclass of MFOTL [60] with two restrictions : it is monadic and all formulas have quantifier rank 1. We choose this logic FOTL_1 as it is natural for expressing properties of unbounded client-server systems where the number of clients is both unbounded and dynamic. A *monodic* formula is a well-formed formula with at most one free variable in the scope of a temporal modality. In this section, we describe the syntax and semantics of FOTL_1 with suitable examples. To give a flavour of FOTL_1 and its expressibility, we enumerate some properties of APS that are not easily expressible in Linear Temporal Logic (LTL). Let P_s be the set of atomic propositions of the server and P_c be the set of client predicates.

In the APS running example, they are defined as follows:

$$\begin{aligned} P_c = \{&parking\_requested(PR),\ occupy\_parking\_lot(OP),\\ &parking\_unavailable(PU), exit\_successfully(ES),\\ &exited\_unsuccessfully(EU)\}\\ P_s = \{&server\_ready(SR)\} \end{aligned}$$

1. When a vehicle requests a parking space, it is always the case that for every vehicle, it eventually exits the system, either successfully after being granted a parking space, or unsuccessfully, when its request is denied.

$$\psi_1 = \mathbf{G}_s(\forall x)\Big(parking\_requested(x) \Rightarrow \mathbf{F}_c\left(exit\_successfully(x)\ \vee\ exit\_unsuccessfully(x)\right)\Big)$$

2. It is always the case that if the client occupies a parking lot, it will eventually exit the parking lot.

$$\psi_2 = \mathbf{G}_s(\forall x)\Big(occupy\_parking\_lot(x) \Rightarrow \mathbf{F}_c(exit\_successfully(x))\Big)$$

3. There may be clients whose requests are rejected.

$$\psi_3 = \mathbf{G}_s(\exists x)\big(parking\_requested(x) \wedge \mathbf{F}_c(exit\_unsuccessfully(x))\big)$$

4. There may be clients who have requested for parking and who wait in the parking unavailable state until they are able to exit the system.

$$\psi_4 = \mathbf{G}_s(\exists x)\Big(parking\_requested(x) \wedge \mathbf{F}_c\big(parking\_unavailable(x)\ \mathbf{U}_c\ exit\_unsuccessfully(x)\big)\Big)$$

It can be observed that there are no free variables in the scope of $\mathbf{G_s}$ and exactly one free variable in the scope of the client modalities. It is also possible to construct FOTL_1 specifications with propositions from P_s and server transitions. The ease of expressibility of the client and server behaviour is the key motivation behind the logic FOTL_1 which is formally described in the subsequent section.

3.3.1 Syntax

The set of *client formulae*, Δ, is the *boolean and temporal modal closure* of atomic client formulae P_c:

$$\alpha, \beta \in \Delta ::= p(x), p \in P_c \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha \wedge \beta \mid \mathbf{X}_c\alpha \mid \mathbf{F}_c\alpha \mid \mathbf{G}_c\alpha \mid \alpha\, \mathbf{U}_c\, \beta$$

The set of *server formulae*, Ψ, is the *boolean and temporal modal closure* of $\Phi = \{(\exists x)\alpha, (\forall x)\alpha \mid \alpha \in \Delta\}$ and atomic server formulae P_s:

$$\Psi ::= q \in P_s \mid \neg\psi \mid \phi \in \Phi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \mathbf{X_s}\psi \mid \mathbf{F_s}\psi \mid \mathbf{G_s}\psi \mid \psi_1\, \mathbf{U_s}\, \psi_2$$

where $\psi, \psi_1, \psi_2 \in \Psi$. It can be observed that in $\mathsf{FOTL_1}$, the quantifier depth is at most one and quantifier alternation is not allowed. The syntax allows us to specify only monodic formulas. Every variable in the server formulas is bounded; $\mathsf{FOTL_1}$ allows for only client formulas to be quantified and we do not have quantifiers over server formulae.

3.3.2 Semantics

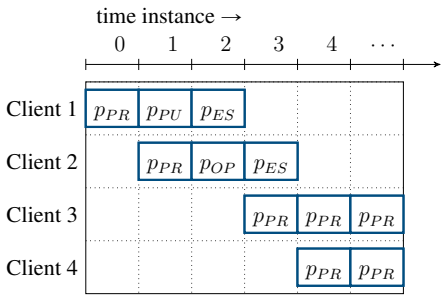


Figure 3.7: Snapshot of the running example (APS) depicting live windows

We consider the *unbounded client-server systems* where all clients are of the same type and they may enter and exit the system dynamically. At any point in time, the number of clients is *bounded*, but their cardinality is *unknown* and *dynamic*. We refer to the clients that are present in the system at any point in time as *live agents* (clients). The *live window* of a particular client begins when it enters the system and ends when the client exits the system. Hence, if there are several *live agents*, their *live windows* would overlap each other. This is interesting as it allows us to reason about the *live clients* which satisfy particular properties simultaneously. We illustrate these concepts with respect to the running example APS before formally defining them.

Example 3.3.1. *Fig. 3.7 depicts the snapshot of the system with* 4 *distinguishable clients, with overlapping* live windows*, with the bound* 5*. The* x *axis denotes the time instance. The clients are along the* y *axis. While the system is* unbounded*, there are a finite number of clients at an instant, as shown in this figure. Each row shows the local state of that client. For each instance, the local state of the client is in the cell i.e, client* 1 *is at state* p_{PR} *at instance* 0*. For client* 1*, the* left boundary*, when it enters the system is at instance* 0 *and its* right boundary *is at instance* 2*, when it exits the system . This corresponds to the client requesting parking and getting rejected. There may be multiple clients in the same local state (client* 3 *and client* 4 *are in state* p_{PR} *at instance* 4*). There may be clients which are live at the bound* 5 *and have not exited the system, such as clients* 3 *and* 4*. This is an interesting case, where the bound (in the snapshot) is equal to the current right boundary for the client.*

At the outset, we define the following objects:

- Let CN be a countable set of client names. The client enters the system and gets a unique identifier (name) from CN. When the client exits the system, the identifier gets discarded and is never reused.

- Let $CS = (Q, \Sigma, \delta, I, F)$ be a finite state machine describing the behaviour of a client. Let $L : Q \rightarrow 2^{P_c}$ be the definition (labelling function) of each client state $q \in Q$ in terms of a subset of properties P_c true in that state. The states in F are sink states with no outgoing arcs.

  Example: In the running example (APS), the clients have exactly one initial state $I = parking\_requested\ (PR)$ and two final states namely,

  $F = \{exit\_successfully\ (ES), exit\_unsuccessfully\ (EU)\}$.

- Let $\mathfrak{Z} : CN \times \mathbb{N}_0 \rightarrow Q$ be a partial mapping describing the local state of each client $a \in CN$ at an instance $i \in \mathbb{N}_0$. For instance, $\mathfrak{Z}(a, i) \in q$, means that the local state of each client a at instance i is state q, where $q \in Q$.

Formally, a model is a triple $M = (\nu, V, \xi)$ where

1. ν gives the local behaviour of the server as follows:

   $\nu = \nu_0\nu_1\nu_2 \ldots$, where for all $0 \leqslant i$, $\nu_i \subseteq P_s$,

2. V gives the set of live agents (clients) at each instance.

   $V = V_0V_1V_2\ldots$, where for all $0 \leqslant i$, V_i is a finite subset of CN, gives the set of live agents at the ith instance.

   For every $0 \leqslant i$, V_i and V_{i+1} satisfy the following properties:

   (a) if $V_i \subseteq V_{i+1}$ then for every $a \in V_{i+1} - V_i$ such that $\mathfrak{Z}(a, i+1) \in I$.

   (b) if $V_{i+1} \subseteq V_i$ then for every $a \in V_i - V_{i+1}$ such that $\mathfrak{Z}(a, i) \in F$.

   V may satisfy the following interesting property. For any $a \in CN$ and $i \in \mathbb{N}_0$ if $a \in V_i$ and if there exists $j > i$ such that $a \notin V_j$ then we may define the left and right boundaries of the live window for a, denoted by $left(a)$ and $right(a)$ where $left(a) \leqslant i \leqslant right(a)$. If no such j exists then there is no right boundary for the live window of that client.

3. $\xi = \xi_0\xi_1\xi_2\cdots$, where for all $0 \leqslant i$, $\xi_i : V_i \to 2^{P_c}$ gives the properties satisfied by each live agent at ith instance, in other words, the corresponding states of live agents. In terms of $\mathfrak{Z}$, $L(\mathfrak{Z}(a, i)) = \xi_i(a)$. Alternatively, ξ_i can be given as $\xi_i : V_i \times P_c \to \{\top, \bot\}$.

3.4 SMT Encoding for the Logic

In this section, we describe the SMT encoding for $\mathsf{FOTL_1}$ which will enable us to implement a bounded model checker tool for restricted ν-nets using $\mathsf{FOTL_1}$ specifications. Let $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$ be the SMT encoding of λ-bounded runs of the net $\mathcal{M}$ containing at most κ agents. The SMT encoding $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$ can be given based on the definition of the restricted ν-net. Let ψ be a property written in the logic language $\mathsf{FOTL_1}$ which is being model-checked in bounded runs of $\mathcal{M}$. For a given $\langle\lambda, \kappa\rangle$ and i (ψ is asserted at i, $0 \leqslant i \leqslant \lambda$), we define two encoding functions $[\psi]^i_{\langle\lambda,\kappa\rangle}$ and ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$. The formula $[\psi]^i_{\langle\lambda,\kappa\rangle}$ denotes the SMT encoding of ψ, where the bounded run of length λ and at most κ agents in the system does not contain a loop. The formula ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$ denotes the SMT encoding of ψ, where the bounded run of length λ and at most κ agents contains a loop which is asserted at i.

We add the following formulas about the dead agents (agents that are not alive) to the system specification:

- If a client exits the parking lot, then it becomes dead in the next state.

$$\delta_1 = \bigwedge_{0\leqslant i\leqslant\lambda-1}\left(\bigwedge_{0\leqslant \mathrm{j}\leqslant\kappa-1} exit\_successfully(\mathrm{j}, i) \Rightarrow (dead(\mathrm{j}, i+1))\right)$$

$$\delta_2 = \bigwedge_{0 \leqslant i \leqslant \lambda-1} \left(\bigwedge_{0 \leqslant \mathrm{j} \leqslant \kappa-1} exit\_unsuccessfully(\mathrm{j}, i) \Rightarrow (dead(\mathrm{j}, i+1)) \right)$$

- If a client is dead then it remains dead.

$$\delta_3 = \bigwedge_{0 \leqslant i \leqslant \lambda-1} \left(\bigwedge_{0 \leqslant \mathrm{j} \leqslant \kappa-1} dead(\mathrm{j}) \Rightarrow (dead(\mathrm{j}, i+1)) \right)$$

In case of a different case study, where we consider any other net, instead of APS, we can suitably replace $exit\_successfully(\mathrm{j})$, $exit\_unsuccessfully(\mathrm{j})$ by the relevant termination condition(s).

The SMT encoding is derived from the bounded semantics in section 3.3. The following encodings are extensions of similar mappings defined in [22]. First, we inductively define $[\psi]^i_{\langle\lambda,\kappa\rangle}$ 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. $[(\exists x)\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{0 \leqslant \mathrm{j} \leqslant \kappa-1} [\alpha[\mathrm{j}/x]]^i_{\langle\lambda,\kappa\rangle}$
4. $[(\forall x)\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge_{0 \leqslant \mathrm{j} \leqslant \kappa-1} [\alpha[\mathrm{j}/x]]^i_{\langle\lambda,\kappa\rangle}$
5. $[\psi_1 \vee \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\psi_1]^i_{\langle\lambda,\kappa\rangle} \vee [\psi_2]^i_{\langle\lambda,\kappa\rangle}$
6. $[\psi_1 \wedge \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv [\psi_1]^i_{\langle\lambda,\kappa\rangle} \wedge [\psi_2]^i_{\langle\lambda,\kappa\rangle}$
7. $[\mathbf{X_s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} [\psi]^{i+1}_{\langle\lambda,\kappa\rangle} & \text{if } (i < \lambda) \\ False & \text{otherwise} \end{cases}$
8. $[\mathbf{F_s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{i \leqslant j \leqslant \lambda} [\psi]^j_{\langle\lambda,\kappa\rangle}$
9. $[\mathbf{G_s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv False$
10. $[\psi_1 \mathbf{U_s} \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{i \leqslant j \leqslant \lambda} ([\psi_2]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge_{i \leqslant j' < j} [\psi_1]^{j'}_{\langle\lambda,\kappa\rangle})$
11. $[p(\mathrm{j})]^i_{\langle\lambda,\kappa\rangle} \equiv p(\mathrm{j}, i)$
12. $[\neg\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \neg\, [\alpha]^i_{\langle\lambda,\kappa\rangle}$
13. $[\alpha \vee \beta]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} \vee [\beta]^i_{\langle\lambda,\kappa\rangle}$
14. $[\alpha \wedge \beta]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} \wedge [\beta]^i_{\langle\lambda,\kappa\rangle}$

15. $[\mathbf{X}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} \big(dead(\mathrm{j}, i+1) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, i+1) \Rightarrow [\alpha]^{i+1}_{\langle\lambda,\kappa\rangle}\big) & \text{if } (i < \lambda) \\ False & \text{if } (i = \lambda) \end{cases}$

If the client is dead at the instance $i+1$, the original formula evaluates to false. If the client is live, the standard bounded semantics apply. Similar semantics are given in equations $16-18$ based on the client liveness at instance j.

16. $[\mathbf{F}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{i \leqslant j \leqslant \lambda} \Big(\big(dead(\mathrm{j}, j) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, j) \Rightarrow [\alpha]^j_{\langle\lambda,\kappa\rangle}\big) \Big)$

17. $[\mathbf{G}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge_{i \leqslant j \leqslant \lambda} \Big(\big(dead(\mathrm{j}, j) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, j) \Rightarrow [\alpha]^j_{\langle\lambda,\kappa\rangle}\big) \Big)$

18. $[\alpha\mathbf{U}_c\beta]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{i \leqslant j \leqslant \lambda} \Big(\big(dead(\mathrm{j}, j) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, j) \Rightarrow prop\_encode_{\alpha\mathbf{U}_c\beta}\big) \Big)$

where $prop\_encode_{\alpha\mathbf{U}_c\beta} = \bigwedge_{i \leqslant j \leqslant \lambda} [\beta]^j_{\langle\lambda,\kappa\rangle} \wedge \bigvee_{i \leqslant j' < j} [\alpha]^{j'}_{\langle\lambda,\kappa\rangle}$

Second, we inductively define ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$, for the loop case 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. ${}_l[(\exists x)\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{0 \leqslant \mathrm{j} \leqslant \kappa-1} {}_l[\alpha[\mathrm{j}/x]]^i_{\langle\lambda,\kappa\rangle}$
4. ${}_l[(\forall x)\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge_{0 \leqslant \mathrm{j} \leqslant \kappa-1} {}_l[\alpha[\mathrm{j}/x]]^i_{\langle\lambda,\kappa\rangle}$
5. ${}_l[\psi_1 \vee \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv {}_l[\psi_1]^i_{\langle\lambda,\kappa\rangle} \vee {}_l[\psi_2]^i_{\langle\lambda,\kappa\rangle}$
6. ${}_l[\psi_1 \wedge \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv {}_l[\psi_1]^i_{\langle\lambda,\kappa\rangle} \wedge {}_l[\psi_2]^i_{\langle\lambda,\kappa\rangle}$
7. ${}_l[\mathbf{X}_\mathbf{s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} {}_l[\psi]^{i+1}_{\langle\lambda,\kappa\rangle} & \text{if } (i < \lambda) \\ {}_l[\psi]^l_{\langle\lambda,\kappa\rangle} & \text{if } (i = \lambda) \end{cases}$
8. ${}_l[\mathbf{F}_\mathbf{s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{min(l,i) \leqslant j \leqslant \lambda} {}_l[\psi]^j_{\langle\lambda,\kappa\rangle}$
9. ${}_l[\mathbf{G}_\mathbf{s}\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge_{min(l,i) \leqslant j \leqslant \lambda} {}_l[\psi]^j_{\langle\lambda,\kappa\rangle}$

10. $${}_l[\psi_1 \mathbf{U_s} \psi_2]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} \bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\psi_2]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\psi_1]^{j'}_{\langle\lambda,\kappa\rangle}) & \text{if } (i \leqslant l) \\ \Big(\big(\bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\psi_2]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\psi_1]^{j'}_{\langle\lambda,\kappa\rangle}) \big) & \\ \qquad \text{or} & \text{if } (i > l) \\ \big(\bigvee\limits_{l \leqslant j < i} ({}_l[\psi_2]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{l \leqslant j' < j} {}_l[\psi_1]^{j'}_{\langle\lambda,\kappa\rangle}) \big) \Big) & \end{cases}$$

11. $${}_l[p(\mathrm{j})]^i_{\langle\lambda,\kappa\rangle} \equiv p(\mathrm{j}, i)$$

12. $${}_l[\neg\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \neg\, {}_l[\alpha]^i_{\langle\lambda,\kappa\rangle}$$

13. $${}_l[\alpha \vee \beta]^i_{\langle\lambda,\kappa\rangle} \equiv {}_l[\alpha]^i_{\langle\lambda,\kappa\rangle} \vee {}_l[\beta]^i_{\langle\lambda,\kappa\rangle}$$

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

15. $${}_l[\mathbf{X}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} \big(dead(\mathrm{j}, i+1) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, i+1) \Rightarrow {}_l[\alpha]^{i+1}_{\langle\lambda,\kappa\rangle}\big) & \text{if } (i < \lambda) \\ \big(dead(\mathrm{j}, l) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, l) \Rightarrow {}_l[\alpha]^{l}_{\langle\lambda,\kappa\rangle}\big) & \text{if } (i = \lambda) \end{cases}$$

If the client is dead at the next instance ($i+1$ or l, respectively), the original formula evaluates to false. If the client is live, the standard bounded semantics apply. Similar semantics are given in equations $16-18$ based on the client liveness at instance j.

16. $${}_l[\mathbf{F}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee_{min(l,i) \leqslant j \leqslant \lambda} \Big(\big(dead(\mathrm{j}, j) \Rightarrow False\big) \wedge \big(\neg dead(\mathrm{j}, j) \Rightarrow {}_l[\alpha]^j_{\langle\lambda,\kappa\rangle}\big) \Big)$$

17. $${}_l[\mathbf{G}_c\alpha]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge_{min(l,i) \leqslant j \leqslant \lambda} \Big((dead(\mathrm{j}, j) \Rightarrow False) \wedge (\neg dead(\mathrm{j}, j) \Rightarrow {}_l[\alpha]^j_{\langle\lambda,\kappa\rangle}) \Big)$$

18. $${}_l[\alpha \mathbf{U_c} \beta]^i_{\langle\lambda,\kappa\rangle} \equiv$$
$$\bigvee_{min(l,i) \leqslant j \leqslant \lambda} \Big(\big(dead(\mathrm{j}, j) \;\Rightarrow\; False\big) \wedge \big(\neg dead(\mathrm{j}, j) \;\Rightarrow\; loop\_prop\_encode_{\alpha \mathbf{U_c} \beta}\big) \Big)$$

$$loop\_prop\_encode_{\alpha \mathbf{U_c} \beta} = \begin{cases} \bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\beta]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\alpha]^{j'}_{\langle\lambda,\kappa\rangle}) & \text{if } (i \leqslant \lambda) \\ \Big(\big(\bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\beta]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\alpha]^{j'}_{\langle\lambda,\kappa\rangle}) \big) & \\ \qquad \text{or} & \text{if } (i > \lambda) \\ \big(\bigvee\limits_{l \leqslant j < i} ({}_l[\beta]^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{l \leqslant j' < j} {}_l[\alpha]^{j'}_{\langle\lambda,\kappa\rangle}) \big) \Big) & \end{cases}$$

Conclusion

The above encoding is utilized in the verification tool that utilizes SMT solvers, which is described in Sec. 6.2.1. Our larger goal is to verify unbounded client server systems. We shall explore the challenges associated with it, the existing literature on verification of these systems and the gap that we aim to fill with our work in the upcoming chapter.

Chapter 4

Bounded Model Checking

Client server systems are a computing paradigm where work loads are distributed by the service providers called server, to the service requesters called clients, or alternatively, the clients request resources from a server. There are variations to this model, where there are single servers and multiple clients, multiple servers and clients and communication among the various entities, passive servers and communicating clients [34]. For instance, stock markets and cryptocurrency exchanges with an unbounded number of investors, multiplayer games where the players are not known apriori can be viewed as client server systems with unbounded agents which we refer to as unbounded client server systems (UCS). The larger goal is to formally verify properties on unbounded client-server systems with concurrency. The central questions that we ask and address in this work are the following:

1. How do we formally verify UCS?
   (a) How do we formally model UCS?
2. How do we express properties of client-server systems naturally?
   (a) Do existing logics suffice?
   (b) What additional properties are interesting in the context of unbounded client-server systems?
3. Are there formal verification tools for UCS? Can we add to the repertoire?
   (a) How are existing tools different, unique? Is another tool necessary?

How do we formally verify UCS? Let us address the first question- formal verification of unbounded client-server systems. In order to formally verify a system, we need to first abstract the system into a formal model and then, describe its properties as logic specifications. UCS can be viewed as a distributed system which permits concurrent servicing of requests. In literature, Petri nets are a well-studied formal model for representing concurrent and distributed systems. The big picture is to employ Petri nets and their extensions for modeling unbounded client-server systems where there is true concurrency in the interaction between the clients and the servers as well as unboundedness in the number of clients. In Sec. 2, we discussed the various suitable formal models - simple PNs, ν-nets and EOSs. To represent the properties of the systems, we employ temporal logics (See Sec. 3).

In order to verify the properties of the unbounded client-server systems (modeled as PNs), we discuss a formal verification technique called model checking. Traditional symbolic model checking employs an exhaustive method to analyse all possible executions of a formal model, but suffers from state space explosion. Bounded Model Checking (BMC) with satisfiability solving, is an alternate approach to address the state space explosion problem. In BMC, a symbolic execution of a transition system and the negation of a property to be verified (translated into a logic formula) is fed to a satisfiability solver. If the formula is satisfiable, a counterexample is found and the property does not hold. If the formula is unsatisfiable up to a pre-defined bound, say k, it is concluded that the property holds upto bound k. It is easy to see that this approach is not complete, however, it is widely adopted in practice for verification of safety properties (where bounds are computable). We explore the applicability of BMC to verify UCS and discuss the algorithm consequently in Sec. 4.1.

How do we express properties of client-server systems naturally? What additional properties are interesting in the context of unbounded client-server systems? Typically, properties such as safety (a client request is never stuck indefinitely without being accepted or rejected), reachability (a client request is eventually serviced) and deadlock-freeness (a server or client action are always taken at each instance) are relevant for study. And these properties are expressible in a temporal logic such as LTL as in Sec. 3.1.

Second, we could additionally make use of interger arithmetic to specify invariants about the system in a counting logic where we have propositions that can count at the atomic level. Assuming arbitrary processes a, b, an interesting invariant would be: The total number of clients

undergoing process a and number of clients undergoing process b does not exceed n. This essentially gives an upper bound and can be described in the counting logic LTL_{LIA} described in Sec. 3.2.

Third, in the context of *unbounded* client-server systems, where clients dynamically enter the system and exit them, it is interesting to reason about the set of clients that are live (i.e, currently interacting with the server) at that instant. If we are unable to model live clients, we would need additional mechanisms to keep track of the clients that have exit the system, which can quickly grow in number, since it is an unbounded system. Since these live clients are distinguishable (via client identifiers) we could reason easily about the live clients and rephrase the following properties over sets of live clients: safety (is it always the case that any set of live clients is never stuck indefinitely without being accepted or rejected) reachability (is it always the case that all live client requests are eventually serviced) and deadlock-freeness (is it always the case that a server or live client action are always taken at each instance)? These properties are expressible naturally, in a fragment of first order logic FOTL_1 described in Sec. 3.3.

Are there formal verification tools for UCS? Although unbounded client-server systems are abundant in the software ecosystem, to the best of our knowledge, there are no specialized tools to verify unbounded client-server systems. However, [97] are recent works in the context of verifying unbounded systems. In [97], they employ SMT solvers to verify unbounded concurrent data structures where there is unboundedness in the list size as well as the number of threads that may access the list. They abstract the system to a bounded model and perform standard bounded model checking on it. We reckon that by employing a technique such as two-dimensional bounded model checking, proposed in Sec. 4.1, it may be possible to verify the system, while allowing concurrent behaviours (which we discuss later in this section). This may result in the solver converging to a satisfying assignment quicker (which we demonstrate in our experiments).

With regards to verification of unbounded nets, [3, 4] studied the model checking of unbounded PNs with backwards reachability, where a set of safety properties were verified. While modeling unbounded client-server systems as unbounded PNs (cf.Sec2.1), we identify the gap in modeling the behaviour of this system with true concurrency and explore ways to verify properties beyond safety. There is no known work on verifying properties of unbounded PNs over truly concurrent semantics. The existing PN (and unbounded PN) verification tools employ

interleaving semantics [3,4,6,9]. We fill this gap while focussing on unbounded client-server systems. It is to be noted that our approach is in contrast to parametric model checking on various software systems [102]. We do not parameterize the dimensions, rather allow them to *unfold* simultaneously.

The novelty of our work is twofold; first, to give a succinct representation of unbounded client-server systems, second, to study true concurrent behaviour and consequently to formally verify unbounded client-server systems in practice over various suitable logics and providing tools for them, while taking advantage of SMT solvers.

4.1 Bounded Model Checking Algorithms

In the subsequent section, we describe BMC algorithms with respect to verification of LTL properties on Petri nets. Notice that in Algorithm. 1 and Algorithm. 2, the temporal property α can be replaced with properties in any logic from Sec. 3 and the model can be replaced by any formal model. Where the unfolded configuration of the Petri net is mentioned (cf. Line. 11 in Algorithm. 1 and cf. Line.14 in Algorithm. 2), the Petri net may be unfolded using either interleaving or true concurrent semantics. By unfolding, we refer to the process of obtaining the subsequent configurations from the initial marking of the net.

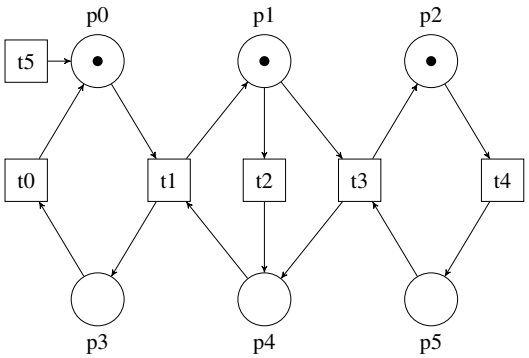


Figure 4.1: An unbounded net

Example 3. Given an unbounded PN Fig. 4.1, one possible sequence of transition firings with interleaving semantics, where exactly one enabled transition is fired at instant is shown in Fig. 4.2. The below are the other possible firing sequences:

$$t_2 \rightarrow t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow t_4$$

$$t_4 \rightarrow t_2 \rightarrow t_1 \rightarrow t_2$$

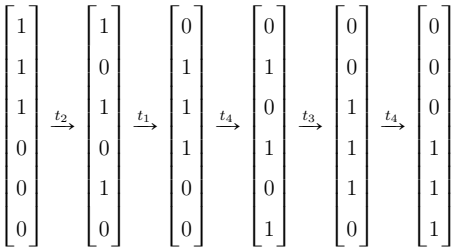


Figure 4.2: Sequence of firings via interleaving semantics

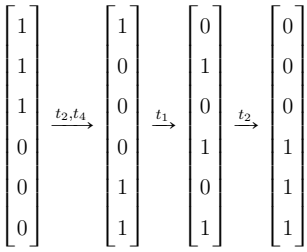


Figure 4.3: Sequence of firings via true concurrent semantics

However, with respect to Fig. 4.1 and true concurrent semantics, we obtain the much shorter firing sequence Fig. 4.3.

In the verification tools discussed in Sec. 5 and Sec. 6, we default to using true concurrent semantics, while also allowing interleaving semantics.

We illustrate the usage of the algorithms via an example.

Example 4. Consider the Fig. 4.1 with the following temporal properties.

1. First consider the LTL reachability property. $F(p_0 = 0 \& p_1 = 0 \& p_2 = 0 \& p_3 = 1 \& p_4 = 1 \& p_5 = 1)$. It is beneficial to use Algorithm. 1 since it is a reachability property and with 5 steps with interleaved unfolding (or 3 steps in case of concurrent unfolding), the marking can be reached. It is not beneficial to use Algorithm. 2 here, since it will not converge to a solution faster.

2. Second, consider the LTL_{LIA} property, where

   $F((\#x)p_3(x) > p_0(x) \wedge p_4(x) > p_1(x) \wedge p_5(x) > p_2(x))$. This property requires searching by varying both the number of tokens as well as the time instance, hence Algorithm. 2 is suitable here.

Algorithm 1 Bounded Model Checking for Petri nets

Input: Initial Marking $M_{I(P)}$, temporal property α in NNF and bound.

Output: SAT with counterexample or UNSAT

Require: $bound \geqslant 0$

1: $index \leftarrow 0$
2: **if** $index == 0$ **then**
3: AddToSolver($M_{I(P)}$)
4: AddToSolver(α)
5: **if** SOLVER.CHECK()==SAT **then**
6: Display trace and exit
7: **end if**
8: **else if** SOLVER.CHECK()==UNSAT **then**
9: $index \leftarrow 1$
10: **while** $index \leqslant bound$ **do**
11: AddToSolver($M_{index(P)}$)
12: AddToSolver(α)
13: **if** SOLVER.CHECK()==SAT **then**
14: Display trace and exit
15: **else if** SOLVER.CHECK()==UNSAT **then**
16: $index \leftarrow index + 1$
17: **end if**
18: **end while**
19: **end if**

4.1.1 Related Work

In symbolic model checking [77] the state transition system is not represented explicitly but as propositional formulas. This prevents what is known as "state-space explosion" problem in program verification, as it slashes the size of representation by one degree of exponent. Originally binary decision diagrams (BDDs) were used to symbolically represent systems, and operations on system states corresponded to BDD operations. In practice, BDDs can handle systems defined using hundreds of variables, but often blow up in space. The technique called Bounded Model Checking (BMC) was an attempt to replace BDDs with SAT (and now SMT) in sym-

Algorithm 2 Two Dimensional Bounded Model Checking for Petri nets

Input: Initial Marking $M_{I(P)}$, temporal property α in NNF and bound.

Output: SAT with counterexample or UNSAT

Require: $bound \geqslant 0$

```
1:  index ← 0
2:  if index == 0 then
3:      AddToSolver(M_{I(P)})
4:      AddToSolver(α)
5:      if SOLVER.CHECK()==SAT then
6:          Display trace and exit
7:      end if
8:  else if SOLVER.CHECK()==UNSAT then
9:      index ← 1
10:     while index ⩽ bound do
11:         for λ ← 1 to bound do
12:             for κ ← 1 to bound do
13:                 AddToSolver(M_{index(P)})
14:                 AddToSolver(UnfoldPN(λ,κ))
15:                 AddToSolver(α)
16:                 if SOLVER.CHECK()==SAT then
17:                     Display trace and exit
18:                 else if SOLVER.CHECK()==UNSAT then
19:                     index ← index + 1
20:                 end if
21:             end for
22:         end for
23:     end while
24: end if
```

bolic model checking. However SAT lacks the possibility to eliminate variables, which is a key operation in BDD-based model checking. In BMC, the solution is to focus on falsification and, at least in a first approximation, drop completeness. It has been found that SAT-based model checking, at least for falsification, scales much better [98].

Autonomous Parking System (APS), which 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. APS is an instance of an infinite (parameterized) family of finite state systems. Such systems are parameterized because the number is known only at run time. Such a family can usually be seen as one single infinite-state system. Checking reachability in parameterized systems is, in general, undecidable [11].

The verification problem for a family of similar state-transition systems is easy to formulate: Given a family $M = \{M_i\}_{i=1}^{\infty}$ of systems M_i and a temporal formula α, verify that for every i, M_i satisfies α. This is known as the Parameterized Model-checking Problem (PMCP). As mentioned above, PMCP is undecidable, in general. However, for *specific* families the problem may be solvable.

In one of the earliest works on PMCP, Browne et al. [27] consider the problem of verifying a family of token rings, that is, the family of rings of size 2, size 3, size 4 and so on. In order to verify the entire family, they establish a *bisimulation* relation between a two-process ring and an n-process token ring for any $n \geqslant 2$. It follows that the two-process ring and the n-process ring satisfy exactly the same temporal formulae.

In [88], Pnueli *et al.* introduce the $(0, 1, \infty)$-counter abstraction method by which a parameterized system of unbounded size is abstracted into a finite-state system. As each process in the parameterized system is finite-state, the abstract variables are limited counters which count, for each local state s of a process, $\kappa(s)$–the number of processes which currently are in local state s. The counters are saturated at 2, which means that $\kappa(s) = 2$ whenever 2 *or more* processes are at state s. The emphasis is on the derivation of an *adequate and sound* set of fairness requirements that enable proofs of liveness properties of the abstract system, from which we can safely conclude a corresponding liveness property of the original parameterized system.

Recently, BMC has been successfully used for verifying safety properties of a special kind of parameterized systems, namely, fault-tolerant distributed algorithms (FTDA). In [67], Konnov et. al., demonstrate that reachability properties of FTDAs can be verified by BMC based techniques. In order to ensure completeness, an upper bound on the distance between states is required. So, they show that the diameters of accelerated counter systems of FTDAs have a quadratic upper bound in the number of local transitions.

Note that APS can not be modelled as FTDAs, which are bit more specialized. In FTDAs the firing of a transition depends not only on the current states of a fixed number of processes,

but also on the total number of processes [14]. This is quite unlike Petri nets which we have used to model APS.

One of the earliest work on BMC for Petri nets was [56]. In this paper, the author applies BMC to 1-safe Petri nets by translating the bounded reachability problem for 1-safe Petri nets into constraint Boolean circuit satisfiability. The BCSAT constraint Boolean satisfiability checker is used to check whether the generated constraint circuit is satisfiable, thereby solving the bounded reachability problem of 1-safe Petri nets. BMC technique was later extended to verify properties of Timed Automata [12] and Timed Petri nets [78].

BMC using SMT Solvers has been an active area of research [89]. In particular, while applying BMC in the Petri net setting, techniques such as net reductions, and structural reductions [100] are applied. Recently, in [8, 9] the generalized reachability of Petri net is encoded 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. Our work attempts to bridge this gap while preserving the original Petri net structure. In Section 5.2 we provide the SMT encoding of the counting logic LTL_{LIA} which distinguishes our work. In [95] a similar counting linear temporal logic for specifying properties of multi-robot applications is proposed. In contrast, LTL_{LIA} uniquely supports specifying properties of client-server systems.

A recent related work on SMT-based verification of safety properties of parameterized multi-agent system (PMAS) using infinite-state model checking was done in [48]. A similar work is [19], where the authors describe parameterised verification of an unbounded number of data-aware multi-agent systems whose properties are specified using a branching temporal logic. In [70] parameterised verification of multi-agent systems is explored and a model checker MCMAS-P implementing parameterised model checking is also described. In these papers, they perform parameterised model checking, whereas in our case, we do not know the number of agents beforehand. In [74] they consider dynamic multi-agent systems (HD-MAS) where the number of agents evolve which is the closest formal model to ours. Our logic can easily be extended to express properties of multi-agent systems which is part of future work.

Conclusion

Now that we have charted out the boundary of what has been verified with respect to these unbounded client server systems, we shall utilize the logics from Chapter. 3 to specify properties

of these systems on Petri nets and build a verification tool in the upcoming chapter.

Chapter 5

LTL Tool over Petri nets

In this chapter, we elaborate on the first contribution of the thesis - bounded model checking of unbounded Petri nets with concurrent semantics. First, we discuss bounded model checking for various properties on Petri nets. Second, we provide the SMT encoding for unbounded Petri nets and their concurrent unfolding. Third, we describe the model checking tool for verifying temporal properties (expressed in LTL with linear integer arithmetic) for unbounded Petri nets and discuss the experimental results.

5.1 Bounded Model Checking for unbounded PNs

The intuition behind BMC and its extension, $2D$-BMC is to represent a counterexample-trace of bounded length symbolically and check the resulting formula with a SAT/SMT solver. If the formula is satisfiable and thus the path feasible, a satisfying assignment returned by the solver can be translated into a concrete counterexample trace that shows that the property is violated. Otherwise, the bound is increased and the process repeated. Complete extensions to this approach allow to stop this process at one point, with the conclusion that the property cannot be violated, hopefully before the available resources are exhausted. The details of $2D$-BMC are discussed in detail in Sec. 4.1. In this section, we focus on performing $2D$-BMC on unbounded Petri nets, taking advantage of true concurrency in the nets and representing the properties of the nets in a temporal logic LTL_{LIA}. Linear Temporal Logic (LTL) [87, 106] is a natural choice to describe temporal properties of unbounded systems. While LTL is useful to us, we cannot express any counting properties, hence, we propose using the counting logic language LTL_{LIA} an extension of LTL with a few differences (cf. Sec. 3.2). 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. The syntax of the logic is explained with respect to client-server systems, but is also applicable in other infinite state systems such as Fig. 2.6. In LTL_{LIA}, 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 $(\#x \leqslant \mathfrak{c})\alpha$ 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 $(\#x)\alpha \leqslant \beta$ and $(\#x)\alpha > \beta$.
Formally, the set of client formulas Δ is given by:

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

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 $q(x)$. One way to express this is as follows: $\neg(((\#x)p(x) > q(x)) \vee ((\#x)q(x) > p(x)))$. Another way would be $(\#x)p(x) <= q(x) \wedge \neg((\#x)q(x) > p(x))$.
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. For the semantics and encoding of this logic we refer the reader to Sec. 3.2.

5.2 Encoding

Now that we have discussed the logic, in the subsequent section, we encode the bounded model checking problem for nets on LTL_{LIA} using various semantics.

5.2.1 Encoding the bounded model checking problem for nets on LTL_{LIA}

First, we describe the notation that we have used. Let $[\mathcal{M}]$ be the SMT encoding of the Petri net model of the system $\mathcal{M}$. Let ϕ be the LTL_{LIA} 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 the system $\mathcal{M}$ against ψ for the bound $k = \lambda + \kappa$ (where $k \geqslant 0$) 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 the number of clients. The propositional formula $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$ encodes the runs of $\mathcal{M}$ of bound $k = \lambda + \kappa$. The formulae $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$, for $0 \leqslant \lambda, \kappa \leqslant k$ are defined later in this section.

The formulas ${}_lL_{\langle\lambda,\kappa\rangle}$ $(0 \leqslant l \leqslant \lambda)$ and $L_{\langle\lambda,\kappa\rangle}$ are loop conditions that are mentioned in the encoding above. For any $0 \leqslant l \leqslant \lambda$, ${}_lL_{\langle\lambda,\kappa\rangle} = \mathcal{T}(s_\lambda, s_l)$ where s_l is the lth state in the run of $\mathcal{M}$ and s_λ is the λth state. Here, the 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: $L_{\langle\lambda,\kappa\rangle} = \bigvee_{0\leqslant l\leqslant\lambda} {}_lL_{\langle\lambda,\kappa\rangle}$. 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 SMT 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 SMT 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 $\mathcal{M}$. The formula $[\psi]^i_{\langle\lambda,\kappa\rangle}$ denotes the SMT encoding of ψ where the bounded run does not have a loop. The formula ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$ denotes the SMT encoding of ψ where the bounded run contains a loop for any i. These encodings are extensions of similar mappings defined in [22] and are given in Section 3.2.

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 $t_0, t_1, \ldots, t_{n_t}$ and $p_0, p_1, \ldots, p_{n_p}$ to represent transitions and places respectively. We use the i-th copy of the transition variable for instance i as follows $t_{0i}, \ldots, t_{n_t i}$, place variables $p_{0j}, \ldots, p_{n_p i}$, where $0 \leqslant i \leqslant \lambda$, in $[\mathcal{M}]_{\langle\lambda,\kappa\rangle}$. For any $\kappa \geqslant 0$, we define

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

Inductively, for any $\lambda > 0$,

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

Now that we have formally described the notations for encoding of the system $\mathcal{M}$, we derive the SMT encoding of LTL_{LIA} for $\mathcal{M}$ from the bounded semantics described earlier.

In this section, we describe the SMT encoding, which is necessary for implementing a bounded model checker tool for Petri nets using LTL_{LIA} specifications. We need to introduce counter variables for each place p in the input Petri net in order to define the SMT encodings of the property ψ. These extra variables are as follows: $\{c_p^i \mid i \geqslant 0,$ and p is a place in the Petri net$\}$. Now we describe the SMT encodings of ψ.

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

1. $[\#p]^i_{\langle\lambda,\kappa\rangle} \equiv c_p^i$
2. $[\mathfrak{c}]^i_{\langle\lambda,\kappa\rangle} \equiv \mathfrak{c}$
3. $[\mathfrak{c} * \#p]^i_{\langle\lambda,\kappa\rangle} \equiv \mathfrak{c} \times c_p^i$
4. $[\#p * \mathfrak{c}]^i_{\langle\lambda,\kappa\rangle} \equiv c_p^i \times \mathfrak{c}$
5. $[\alpha + \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} + [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
6. $[\alpha - \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} - [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
7. $[\alpha < \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} < [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
8. $[\alpha > \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} > [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
9. $[\alpha \leqslant \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} \leqslant [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
10. $[\alpha \geqslant \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} \geqslant [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
11. $[\alpha = \hat{\alpha}]^i_{\langle\lambda,\kappa\rangle} \equiv [\alpha]^i_{\langle\lambda,\kappa\rangle} = [\hat{\alpha}]^i_{\langle\lambda,\kappa\rangle}$
12. $[q]^i_{\langle\lambda,\kappa\rangle} \equiv q_i$

13. $[\neg q]^i_{\langle\lambda,\kappa\rangle} \equiv \neg q_i$

14. $[\psi \vee \psi']^i_{\langle\lambda,\kappa\rangle} \equiv [\psi]^i_{\langle\lambda,\kappa\rangle} \vee [\psi']^i_{\langle\lambda,\kappa\rangle}$

15. $[\psi \wedge \psi']^i_{\langle\lambda,\kappa\rangle} \equiv [\psi]^i_{\langle\lambda,\kappa\rangle} \wedge [\psi']^i_{\langle\lambda,\kappa\rangle}$

16. $[X\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} [\psi]^{i+1}_{\langle\lambda,\kappa\rangle} & \text{if } i < \lambda \\ \text{False} & \text{otherwise} \end{cases}$

17. $[F\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee\limits_{i \leqslant j \leqslant \lambda} [\psi]^j_{\langle\lambda,\kappa\rangle}$

18. $[G\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \text{False}$

19. $[\psi U \psi']^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee\limits_{i \leqslant j \leqslant \lambda} ([\psi']^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} [\psi]^{j'}_{\langle\lambda,\kappa\rangle})$

We define ${}_l[\psi]^i_{\langle\lambda,\kappa\rangle}$ inductively. To conserve space, encoding is given only where they differ from the corresponding case without loop.

16. ${}_l[X\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} {}_l[\psi]^{i+1}_{\langle\lambda,\kappa\rangle} & \text{if } (i < \lambda) \\ {}_l[\psi]^{l}_{\langle\lambda,\kappa\rangle} & \text{if } (i = \lambda) \end{cases}$

17. ${}_l[F\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigvee\limits_{min(l,i) \leqslant j \leqslant \lambda} {}_l[\psi]^j_{\langle\lambda,\kappa\rangle}$

18. ${}_l[G\psi]^i_{\langle\lambda,\kappa\rangle} \equiv \bigwedge\limits_{min(l,i) \leqslant j \leqslant \lambda} {}_l[\psi]^j_{\langle\lambda,\kappa\rangle}$

19. ${}_l[\psi \mathbf{U} \psi']^i_{\langle\lambda,\kappa\rangle} \equiv \begin{cases} \bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\psi']^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\psi]^{j'}_{\langle\lambda,\kappa\rangle}) & \text{if } (i \leqslant l) \\ \Big(\big(\bigvee\limits_{i \leqslant j \leqslant \lambda} ({}_l[\psi']^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{i \leqslant j' < j} {}_l[\psi]^{j'}_{\langle\lambda,\kappa\rangle}) \big) & \\ \qquad \text{or} & \text{if } (i > l) \\ \big(\bigvee\limits_{l \leqslant j < i} ({}_l[\psi']^j_{\langle\lambda,\kappa\rangle} \wedge \bigwedge\limits_{l \leqslant j' < j} {}_l[\psi]^{j'}_{\langle\lambda,\kappa\rangle}) \big) \Big) & \end{cases}$

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

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

The variable with respect to time instances is denoted as λ. Hence, it is also a positive integer. In order to determine $[M]_{\langle\lambda,\kappa\rangle}$, we can make use of the previously encoded $[M]_{\langle\lambda-1,\kappa\rangle}$ and unfold the transition relation by one length, and **and** it with the place constraints as done previously (cf. Fig 5.1). Hence we get the resultant relation:

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

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

$[(\#x > k)p(x)]^i_{\langle\lambda,\kappa\rangle} \equiv c_p > \kappa$. This encoding is similar for the loop-free formula as well.

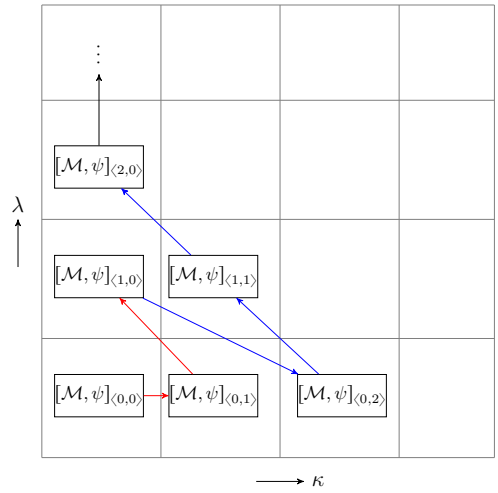


Figure 5.1: Unfolding of the encoded formula $[\mathcal{M},\psi]_{(\lambda,\kappa)}$ with respect to λ (execution steps) and κ (number of tokens)

Unfolding the encoded formula

The unfolding of the formula $[\mathcal{M},\psi]_{\langle\lambda,\kappa\rangle}$ for each k with respect to λ (execution steps) and κ (number of tokens), upto $k = 2$ is depicted pictorially in Fig. 5.1. Initially, when $k = \lambda + \kappa = 0$, $k = 0$ (bound), $\lambda = 0$ (time instance), $\kappa = 0$ (number of clients), the formula $[\mathcal{M},\psi]_{\langle 0,0\rangle}$ 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 $\langle\lambda,\kappa\rangle = \langle 1,0\rangle$. First, κ is incremented and the formula $[\mathcal{M},\psi]_{\langle 0,1\rangle}$ is evaluated. If this found to be unsatisfiable, λ is incremented and the formula

$[\mathcal{M}, \psi]_{\langle 1,0 \rangle}$ is evaluated. In each micro-step of the unfolding, either λ or κ are incremented, and the resulting formula is verified. The Fig. 5.1 represents one among many possible ways of exploring the state space in the system.

5.3 DCModelchecker : A tool to verify Unbounded PNs

DCModelChecker 2.0 is an easy-to-use tool for performing two-dimensional bounded model checking of Petri nets using the logic LTL_{LIA}. DCModelChecker 1.0 [86] can verify counting properties with interleaving semantics. We incrementally extended DCModelChecker 1.0, adding support for true concurrent semantics to the tool, to obtain DCModelChecker 2.0 [85] which supports both. In the rest of this chapter, when we mention tool, we refer to DCModelChecker 2.0. We give the architecture and the overview of the tool in Section 5.3.1. We describe the workflow in Section 5.3.2, and in Section 5.4, we report the experiments. The experiments are easily reproducible by directly executing the scripts in our artifact [85].

5.3.1 Architecture

The general architecture of the tool is shown in Fig. 5.2. The tool has two primary inputs- system description in standard PNML format and the property to be tested expressed in LTL_{LIA}. The system description using Petri nets in PNML format can be obtained from the vast collection of industrial and academic benchmarks available at MCC [68] or created using a Petri net Editor [20, 109]. We make use of both types of benchmarks, for comparative testing. While several Petri net verification tools perform bounded model checking, ours is unique in the model checking strategy, displaying the counterexample and useful in particular for verifying temporal properties and invariants of unbounded client-server systems.

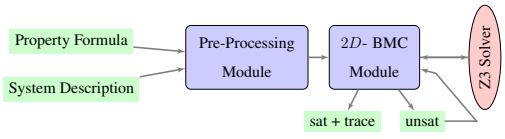


Figure 5.2: DCModelChecker 2.0 architecture

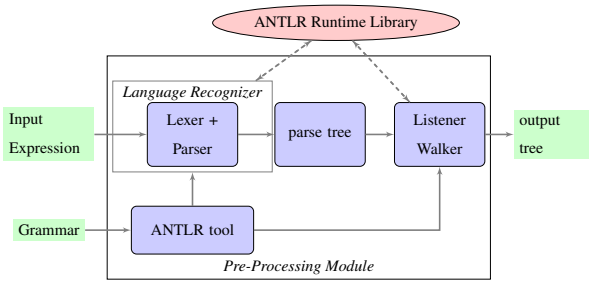


Figure 5.3: Pre-Processing the model and formula using ANTLR

First, the two inputs are fed to the pre-processing module and consequently to the DC-ModelChecker 2.0 tool. The objective of the pre-processing module is to read and validate the two inputs. We make use of ANTLR [82] 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. 5.3. 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 required data structures. We have hand coded the grammar for the PNML format of both types and the grammar of LTL_{LIA}, to be used by ANTLR. This is explained in detail in Sec. 5.3.3.

The 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 [42], to solve the encoded formula and give us a result of unsatisfiable, or satisfiable with a counterexample trace. If unsatisfiable, the tool can increment the bound and look further, until the external termination bound is hit, according to the BMC algorithm. We chose Z3, for its wide industrial applications, developer community support, and ease of use. The detailed workflow is discussed in the subsequent section.

5.3.2 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 $k = 0$ 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 configuration 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. 5.1. We have depicted the workflow only up to $k = 1$ in Fig. 5.5. In practice, for the property ϕ, we search until we have considered all micro-steps for the termination bound k. As expected, for the given model and bound 100 (i.e, when parameter k reaches 100) it is observed that a counterexample is not found, hence we terminate. This means that the property holds for all traces of the model up to the length 100.

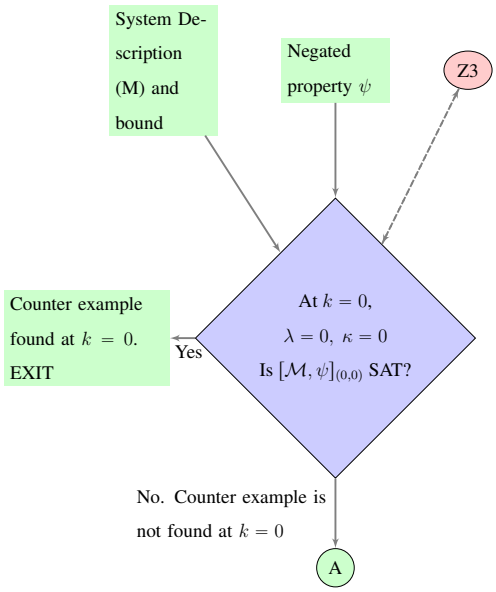


Figure 5.4: Workflow for $k = 0$

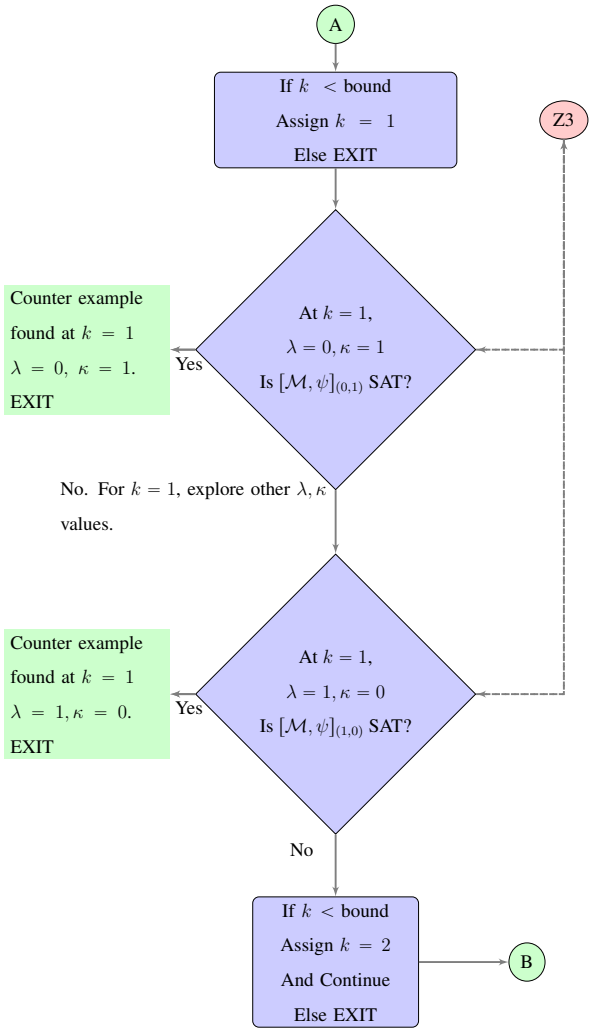


Figure 5.5: Workflow for $k = 1$

5.3.3 Pre-processing Petri nets

In order to build a verification tool that uses Petri nets, we require a standard way of representation. While, various Petri net verification tools exist, recently, the most widely accepted standard is the Petri net Markup Language (PNML) [58]. PNML is based on the Extensible Markup Language, or XML for short, which lends itself to interoperability between tools, while ensuring readability. We employ the ANTLR [82] tool for parsing the PNML input. This tool was chosen, since it will also enable us to pre-process the various extensions to Petri nets (cf. Chapter. 2) and the various logic specifications(cf. Chapter. 3). This module is also available as an open source tool [90].

The tools discussed and built as part of this research work are compatible with any Petri net verification/analysis tools that use the PNML format. Additionally, one may also use existing graphical tools to visualise the Petri nets, limited only by the size of the Petri net [109]. This

decision was made early in the work, to enable reuse of the prototypes and modules built in the tool chain by other research groups and to ensure reproducibility of results.[1] We list down the actual **lexer** and **parser** grammars that were used to pre-process the nets in the following section.

Lexer rules:

First, we name the following set of rules, such that we can refer to in the parsing phase:

```
lexer grammar PNMLLexer;
```

Consider the following rule that recognizes the escape sequences for tab spaces, carriage return and new line respectively and skips over them when they occur in the input file (PNML file).

```
S : [ \t\r\n]+ -> skip ;
```

The PNML format is a type of markup file which contains tags. We have the following rules to identify the open, close and comment tags; the digits and text:

```
COMMENT : '<!--' .*? '-->' -> skip;
SPECIAL_OPEN: '<?' ->
    pushMode(INSIDE) ;
OPEN : '<' -> pushMode(INSIDE) ;
DIGIT : [0-9]+ ;
```

```
TEXT : ~[<&]+ ; //16 bit char except
    < and &
mode INSIDE;
CLOSE : '>' -> popMode ;
SPECIAL_CLOSE: '?>' -> popMode ;
SLASH_CLOSE : '/>' -> popMode ;
```

We have rules to tokenize the keywords and special characters used in the PNML file:

```
SLASH : '/' ;
EQUALS : '=' ;
QUOTE : '"' ;
SQUOTE : '\'' ;
PLACE : 'place' ;
TRANSITION : 'transition' ;
ARC : 'arc';
INITIAL : 'initialMarking' ;
```

```
INSCRIPTION : 'inscription' ;
TEXTTAG : 'text' ;
USCORE : '_' ;
SOURCE : 'source' ;
TARGET : 'target' ;
ID : 'id' ;
STRING : '"' ~[<"]* '"'
         | '\'' ~[<']* '\'';
```

[1] It is noteworthy, that when this research was submitted to Petri net tool venues, the tool was awarded the reproducibility badge, however as the tool paper itself was not accepted, it is not being displayed alongside.

We have a special set of rules for the name tag. The name consists of an alphabet and may be succeded by any combination of digits, alphabets and special symbols (hyphen, underscore, dot), as allowed by the Petri net graphical analysis tool that we used [109].

```
Name : NameStartChar NameChar* ;
NameChar : NameStartChar
         | '-' | '_' | '.' | DIGIT
         | '\u00B7'
         | '\u0300'..'\u036F'
         | '\u203F'..'\u2040';
NameStartChar : [:a-zA-Z]
         | '\u2070'..'\u218F'
         | '\u2C00'..'\u2FEF'
         | '\u3001'..'\uD7FF'
         | '\uF900'..'\uFDCF'
         | '\uFDF0'..'\uFFFD';
```

Parser rules:

Now that we have the lexer rules that can identify and tokenize the input, in this section, we write the the parser rules which will be used by ANTLR to construct the parse tree from the valid input and throw errors if any.

First, we set the specific vocabulary of the parser as **PNMLLexer** (the set of lexer rules we defined above):

```
parser grammar PNMLParser;
options {
   tokenVocab = PNMLLexer;
}
```

A valid PNML file contains a header followed by valid elements:

```
doc: header element;
```

An element may be either an open tag followed by a close tag or an empty tag, each containing its name and possibly containing a set of attributes. There may be open tags for exactly one of the predefined keywords such as place, transition etc. Notice that it is not possible to define just a close tag using these set of rules. If such an input exists, our tool parses the PNML file and invalidates it by throwing a suitable error.

```
header: '<?' Name attribute* '?>';
element:
     '<' Name attribute* '>' (
      place
      | transition
      | arc
      | element
      | textTagDigit
      | textTag
      | TEXT
      )* '<' '/' Name '>'
      | '<' Name attribute* '/>';
```

The place tag contains a mandatory identifier attribute and may contain details to represent the initial marking or special text for parsing.

```
place: '<' 'place' 'id' '=' STRING '>' (initial | element | TEXT)* '<' '/'
   'place' '>';
```

The rule for parsing the initial marking:

```
initial:
   '<' INITIAL '>' (textTagDigit | textTag | element | TEXT)* '<' '/'
      INITIAL '>';
```

The rule for parsing transitions, with a mandatory identifier.

```
transition:
   '<' 'transition' 'id' '=' STRING '>' (element | TEXT)* '<' '/'
      'transition' '>';
```

The rule to recognize arcs from a place/transition to transition/place along with their identifiers.

```
arc:
   '<' 'arc' 'id' '=' STRING source target '>' (
      inscription
      | element
      | TEXT
   )* '<' '/' 'arc' '>'
   | '<' 'arc' 'id' '=' STRING source target '/>';
source: 'source' '=' STRING;
target: 'target' '=' STRING;
```

Some additional rules to parse the text and specify the type of net for validation:

```
1 inscription:
2    '<' INSCRIPTION '>' (textTagDigit | textTag | element | TEXT)* '<' '/'
        INSCRIPTION '>';
3 textTagDigit: '<' TEXTTAG '>' DIGIT '<' '/' TEXTTAG '>';
4 textTag: '<' TEXTTAG '>' TEXT '<' '/' TEXTTAG '>';
5 attribute: ('id' | Name) '=' STRING;
```

Similarly, the rules are written to validate the logic LTL_{LIA} as well and are available in our artifact. As illustrated, in Fig. 5.2, given a valid system description in a PNML file and a valid specification which are validated by the pre-processing module, the $2D$-BMC algorithm verifies the specification using Z3 and produces a counterexample trace or times out. In the next section, we look at the experiments conducted on our bounded model checking tool.

5.4 Benchmarks and Comparisons

In this section, we detail the experimental evaluation of DCModelChecker 2.0. The goals of our evaluation are to demonstrate: (1) the correctness of results returned by DCModelChecker 2.0 in comparison with the state-of-the-art ITS-Tools [99] (2) the comparison of DCModelChecker 2.0 against DCModelChecker 1.0 on similar FireabilityCardinality properties which are not verifiable by tools in the MCC (3) verification of Unbounded Petri nets in comparison with other tools (4) the strength of DCModelChecker 2.0 is in additionally verifying invariants specified in LTL_{LIA} which are not verifiable by tools in the MCC nor DCModelChecker 1.0. To replicate the experiments and plots, a collection of simple shell and python scripts are publicly available in our artifact [85].

Benchmarks: In the first set of experiments described in Section 5.4.1 we consider benchmark models from the Model Checking Contest [68](MCC) and translated the LTLFireability properties into LTL_{LIA}. In our second set of experiments in Section 5.4.2, we make use of the MCC benchmark models and write the properties containing both fireability and cardinality constraints for those models, which are not expressible in the language of MCC. In the third set of experiments, we make use of synthetic unbounded Petri nets [9]. The fourth experiment demonstrates verification of invariants using the unbounded client-server system benchmark (APS) using DCModelChecker 2.0.

		ITS Tools	
		True	False
DCModelChecker 2.0	SAT Property is False	A Erroneous output by DCModelChecker $\mid A \mid = 9$	B Both tools agree on the property being false $\mid B \mid = 87$
	UNSAT Property is True	C $\mid C \mid = 72$	D $\mid D \mid = 179$

Table 5.1: Categorization of LTLFireability results

5.4.1 Experiment 1: Computing the Tool Confidence and Verification of LTLFireability properties - DCModelChecker 2.0 vs the state-of-the-art tool

To compute the Tool Confidence, 22 benchmarks from the MCC were verified using DCModelChecker 2.0 and compared against the state-of-the-art model checking tool ITS-Tools [99]. ITS-Tools has 100 % tool confidence [2]. The comparative experiments were performed with a timeout of 3600 seconds.

Evaluation Criteria: ITS-Tools returns True when the property holds true and False when the property does not hold. DCModelChecker 2.0 however, returns a satisfying counterexample to the negation of the property when the property does not hold. When DCModelChecker 2.0 returns UNSAT, it means that the property holds up to the given bound, and that the property may become false at a greater bound. Hence, for comparison, we may categorize the results of both tools as shown in Table 5.1.

The instances in category A are those false positive properties where DCModelChecker 2.0 returned a counterexample, whereas ITS-Tools returned false. The instances in category B are those where both tools agreed that the property was false, this is particularly significant since it gives the correctness of DCModelChecker 2.0. The instances in category C are those where DCModelChecker 2.0 returns UNSAT and ITS-Tools returns true. The instances in category D are those where DCModelChecker 2.0 returns UNSAT and ITS-Tools returns false. Those instances in categories C and D may become false at a greater bound, hence we do not conclu-

sively take them into account. We consider only the properties that fall in category A and B for computing the confidence ratio. This is similar to the MCC tool confidence computation [1] which excludes those values that are not computed by the tool. The Confidence of a tool is a value $C_{tool} \in [0, 1]$, where 1 denotes that the tool returns the correct result that is agreed upon by atleast 3 participating tools and 0 denotes that the tool never returns the commonly agreed result (otherwise referred to as trusted values). In the LTL Formulas examination category, ITS-Tools has a tool confidence of 100%, $C_{ITS-Tools} = 1$. Hence, it is sufficient to compare DCModelChecker 2. 0 against ITS-Tools to check for the correctness of the results. The C_{tool} is computed as follows:

$$C_{tool} = \frac{\mid V_{tool} \mid}{\mid V \mid} \tag{5.1}$$

where $\mid V \mid$ is the number of trusted values, which we obtain from ITS-Tools. $\mid V_{tool} \mid$ is the number of results obtained by DCModelChecker 2.0 within the set of trusted values, such that $V \subseteq V_{tool}$. For Experiment 1, comparing LTL Fireability properties of DCModelChecker and ITS-Tools, we obtained the following tool confidence $C_{DCModelChecker} = \frac{|A|}{|A+B|} = 0.90$. The underlying $2D$-BMC strategy is not complete.

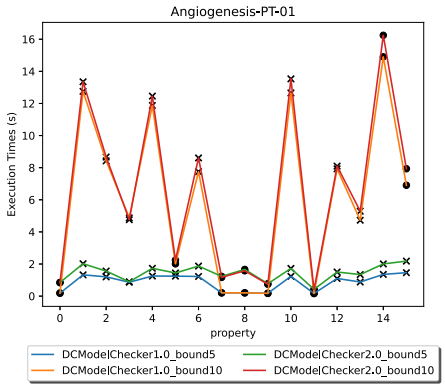


Figure 5.6: Verification of FireabilityCardinality Properties with bound 5 and 10

5.4.2 Experiment 2: Verification of FireabilityCardinality properties DCModelChecker 1.0 vs DCModelChecker 2.0

In this set of experiments, DCModelChecker 1.0 and DCModelChecker 2.0 are compared on similar properties. Here, 20 benchmarks from the Model Checking Contest [68] have been considered and their properties expressed in $\mathcal{L}_C$ [84] for verification by DCModelChecker 1.0 and

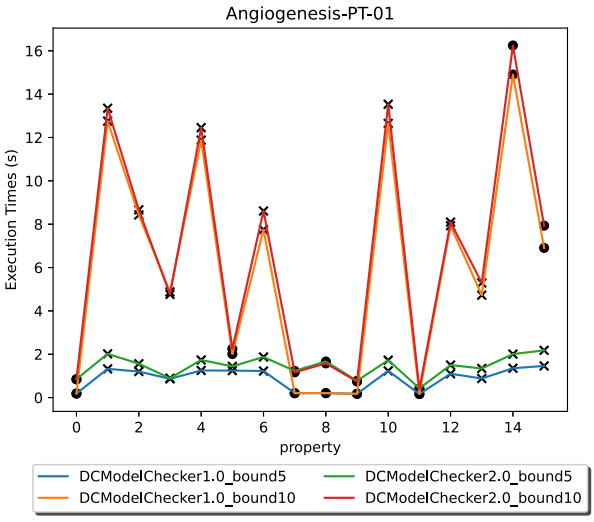
their equivalent LTL_{LIA} properties for verification by DCModelChecker 2.0 respectively. The logic LTL_{LIA} is strictly more expressive than logic $\mathcal{L}_C$. It is to be noted that the languages $\mathcal{L}_C$ and LTL_{LIA} are not equivalent to the logic language used in the MCC. Hence, we have given 16 synthetic properties called FireabilityCardinality properties, derived from the LTLFireability and LTLCardinality properties from the MCC.

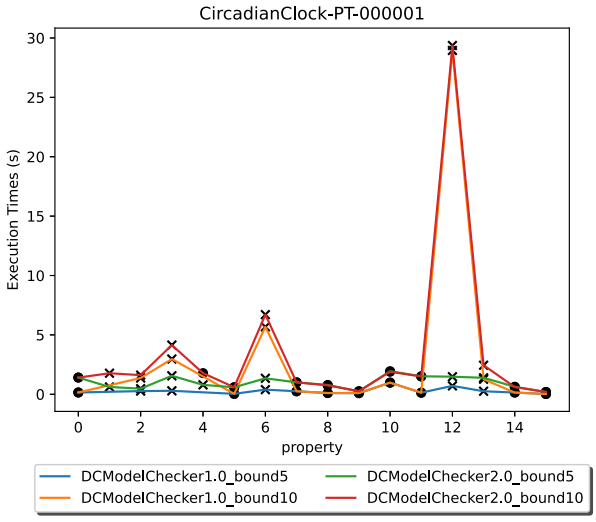
We introduced the set of FireabilityCardinality properties as a combination of the LTLFireability and LTLCardinality properties. For instance, $G(F(t_0) \wedge F(\#p_1 > 999)$, which is a conjunction of the temporal property describing the firing of transition t_0 and the counting property with place p_1.

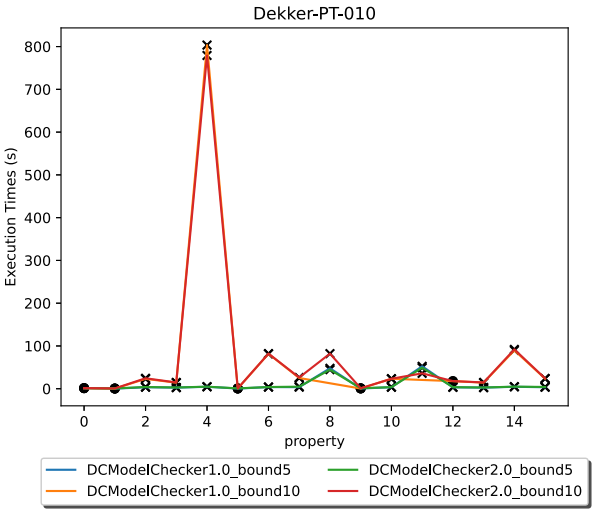
For each of these 20 benchmarks, DCModelChecker 2.0 was pitted against DCModelChecker 1.0 to verify a set of 16 FireabilityCardinality properties for each model, totalling 320 properties and 1280 runs with a timeout of 3600 seconds each. The execution times (seconds) are plotted as separate line plots for bounds 5 and 10. As a sample, 16 properties were verified for the model Angiogenesis-PT-01 as shown in Fig 5.6. Each instance represents a FireabilityCardinality property. If the property is false and the tool returns a counterexample trace, it is depicted by a $\bullet$ and the symbol $\times$ denotes that the tool returned unsatisfiable (indicating that the property may be false at a greater bound). In Fig 5.6, we can observe that for bound 5 (blue line), out of the 16 properties, 5 properties were false. On increasing the bound to 10, we obtain 3 new counterexample traces (corresponding to properties 5, 14 and 15) as more search space is explored by the tool, which also takes more time. This trend is observed in all models in our experiment, when the bound is increased, new counterexamples may be found by tool. The execution times of DCModelChecker 2.0 and DCModelChecker 1.0 are comparable. It is important to note that for each property both tools agree on the SAT/ UNSAT answers returned by them.

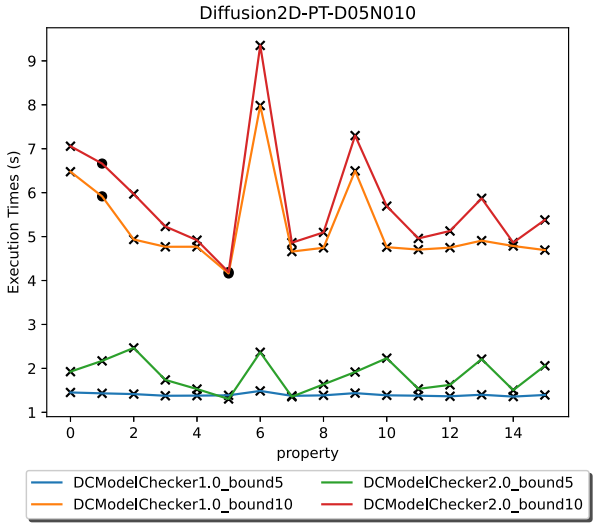
5.4.3 Experiment 3: Verification of Unbounded Petri nets

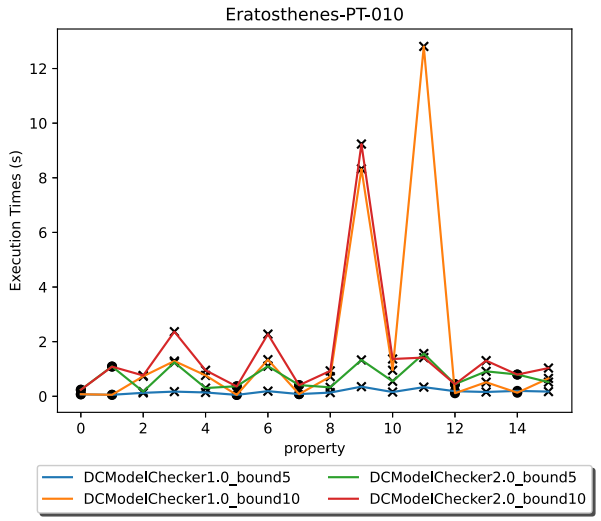
To quote Amat et al, "It is difficult to find benchmarks for unbounded Petri nets" [9]. The strength of DCModelChecker 2.0 lies in verifying unbounded Petri nets. Since, MCC benchmarks are bounded [9], we make use of the unbounded Petri nets given by [81] and verify LTLFireability properties in comparison with DCModelChecker 1.0 [86], ITS-Tools [99] and Tapaal [40]. The results of these experiments are in Table 5.2, and [86]. Each property that was verified is false. In each case it was observed that DCModelChecker 2.0 and DCModelChecker

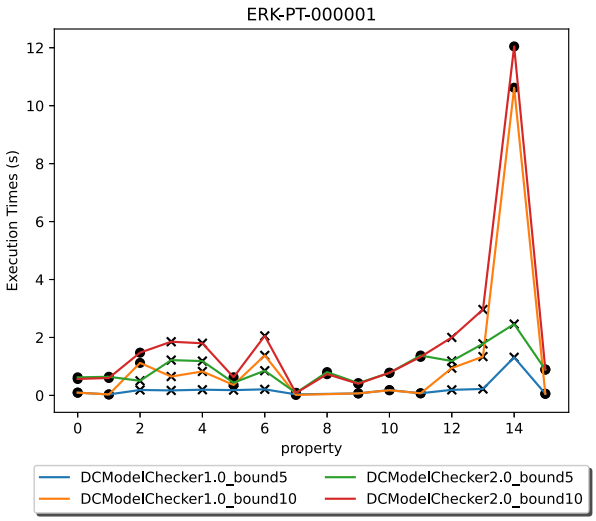

Angiogenesis-PT-01
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

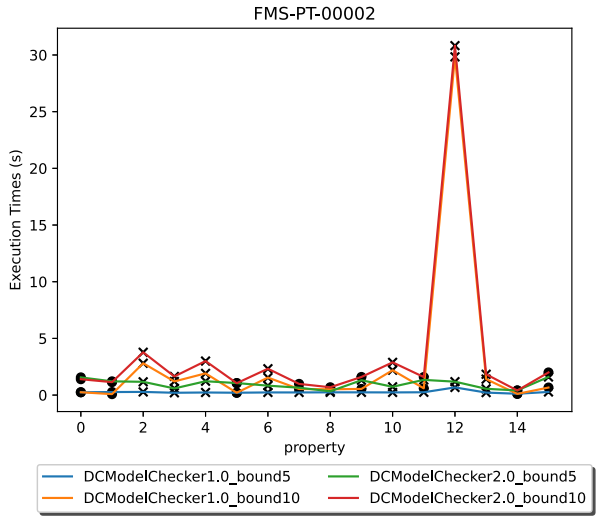

CircadianClock-PT-000001
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

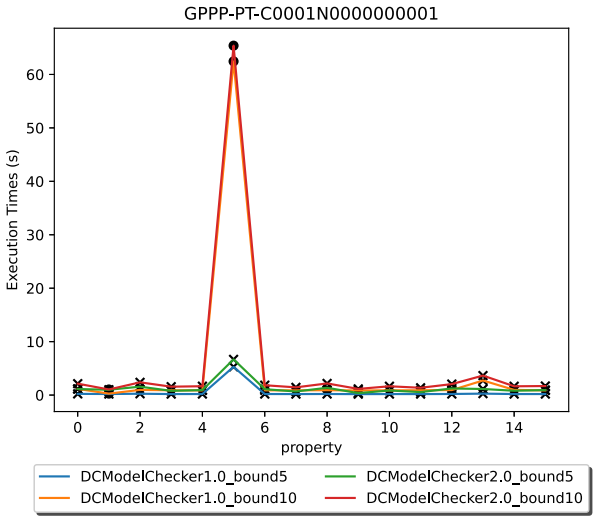

Dekker-PT-010
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

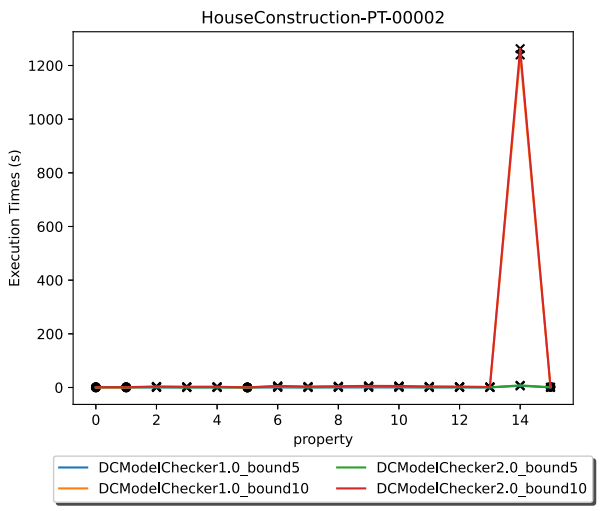

Diffusion2D-PT-D05N010
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

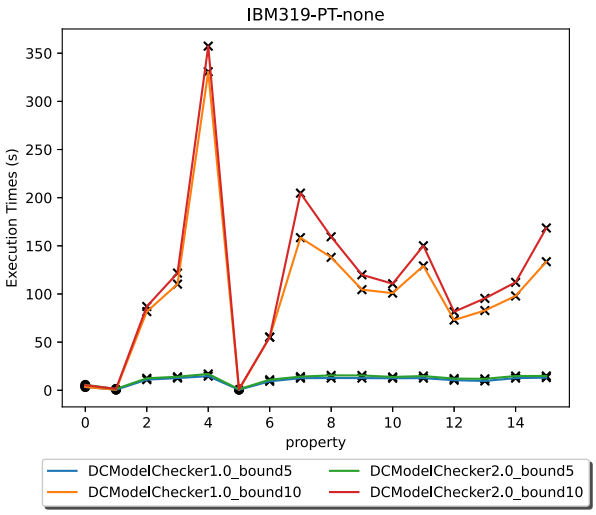

Eratosthenes-PT-010
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

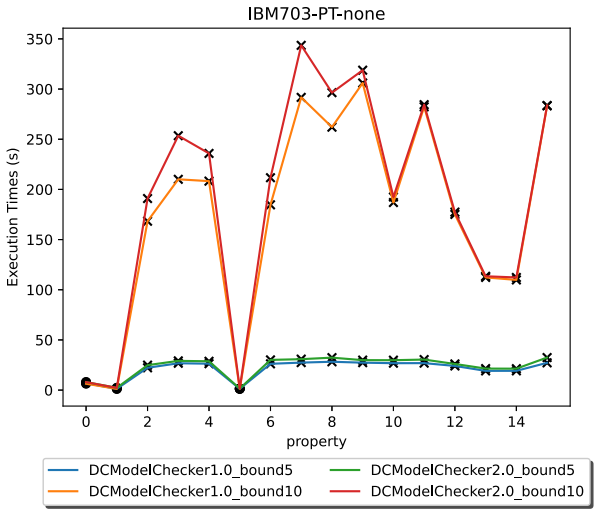

ERK-PT-000001
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

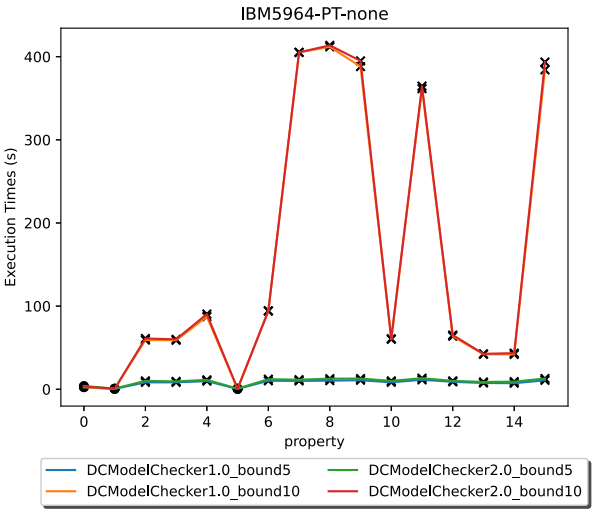

FMS-PT-00002
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

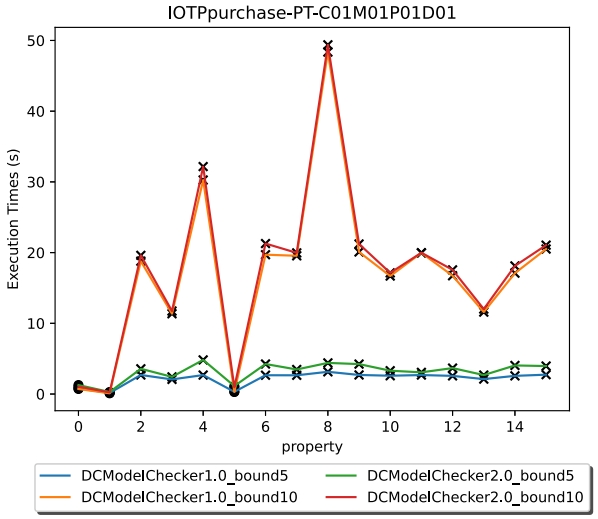

GPPP-PT-C0001N0000000001
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

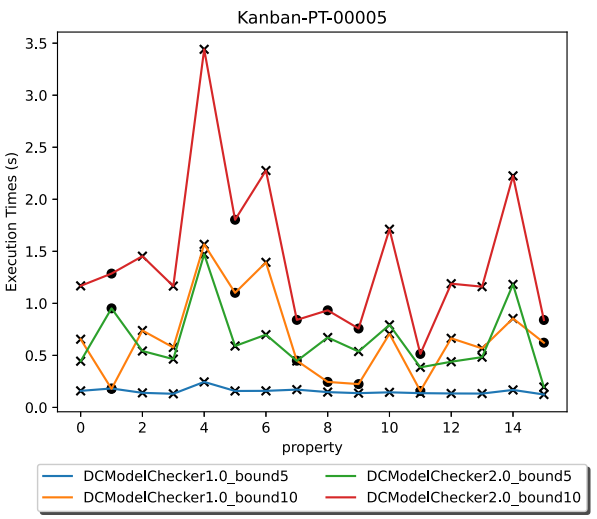

HouseConstruction-PT-00002
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

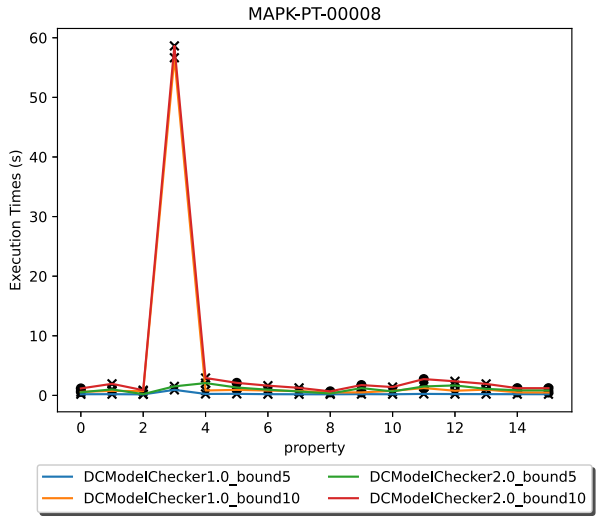

IBM319-PT-none
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

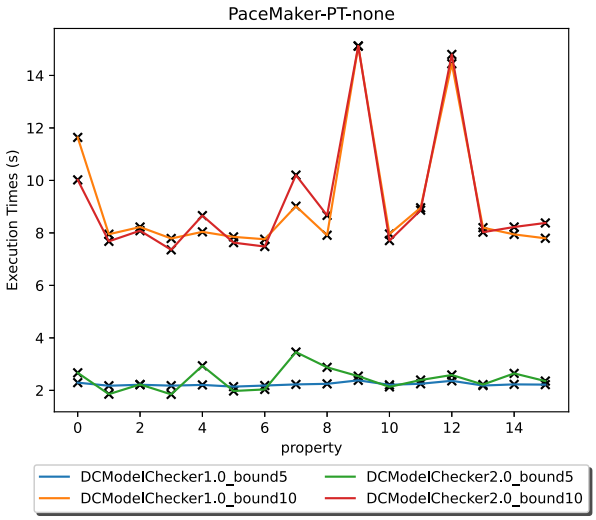

IBM703-PT-none
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

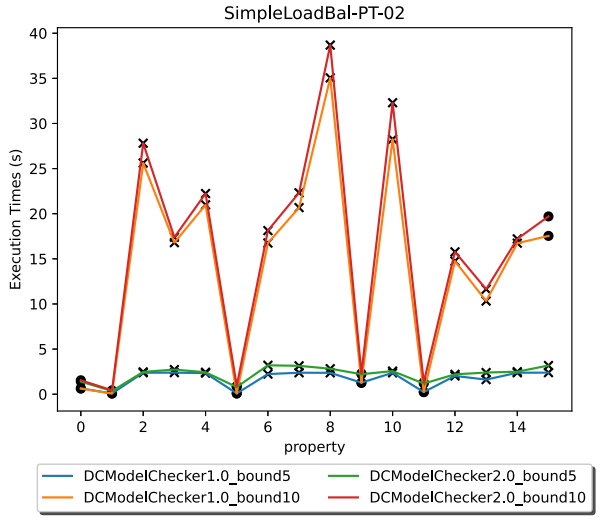

IBM5964-PT-none
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

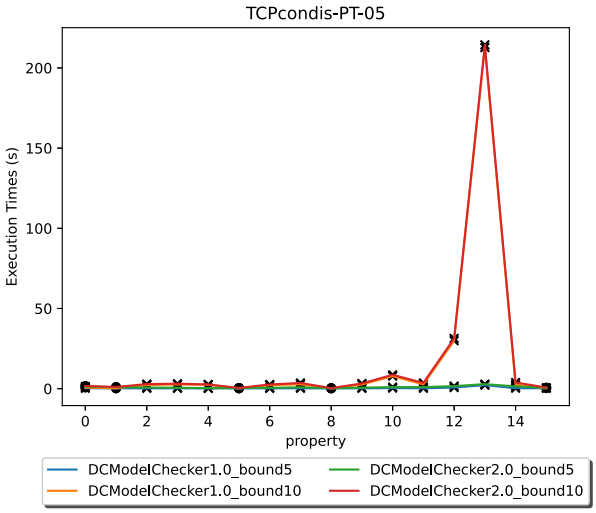

IOTPpurchase-PT-C01M01P01D01
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

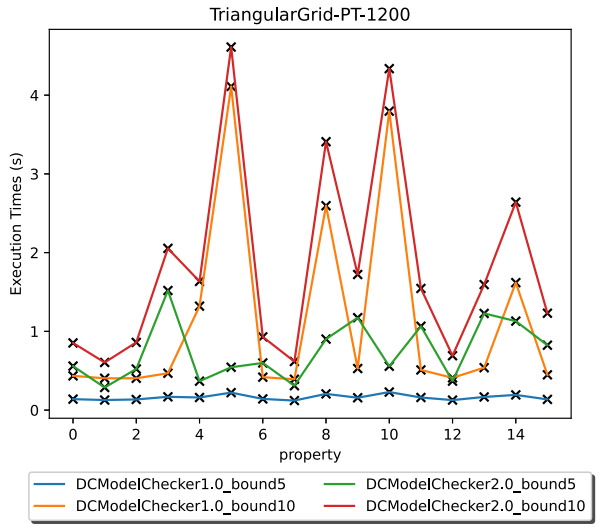

Kanban-PT-00005
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

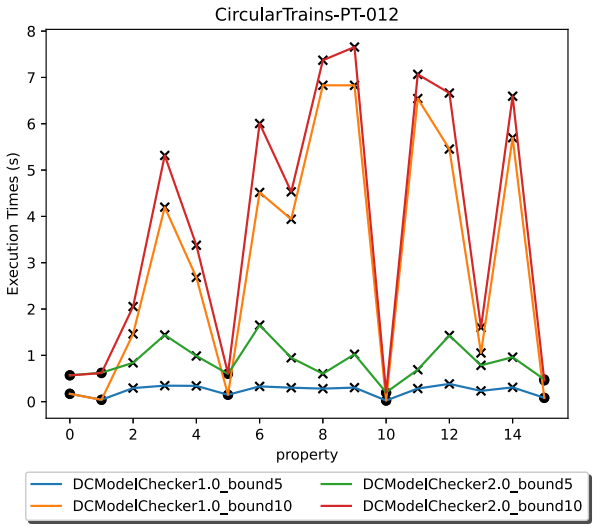

MAPK-PT-00008
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10


PaceMaker-PT-none
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10


SimpleLoadBal-PT-02
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10


TCPcondis-PT-05
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10


TriangularGrid-PT-1200
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10


CircularTrains-PT-012
Execution Times (s)
property
DCModelChecker1.0_bound5
DCModelChecker1.0_bound10
DCModelChecker2.0_bound5
DCModelChecker2.0_bound10

Model	Execution Times(ms)			
	DCModelChecker 1.0	DCModelChecker 2.0	ITS-Tools	Tapaal
Parity	**0.010**	0.40	0.933	0.02
PGCD	**0.019**	0.98	1.487	0.07
Process	0.033	0.85	1.201	**1e-05**
CryptoMiner	0.036	0.64	0.957	**7e-06**
Murphy	0.031	0.83	1.161	**8e-06**

Table 5.2: Results of Experiment 3

Model	**Property**
Parity	$\neg(t_0 U t_1)$
PGCD	$\neg GF(t_0 U t_1)$
Process	$\neg F(t_0 U t_1)$
CryptoMiner	$\neg F(\text{OB } U \text{ GH})$
Murphy	$\neg F(t_1 U t_4)$

Table 5.3: List of properties verified on unbounded nets

1.0 returned that the property is false and additionally gave a counterexample trace, whereas ITS-Tools and Tapaal returned that the property is false. In Section 5.4.3, we verified LTLFireability properties of unbounded Petri nets which were all found to be false. The properties are described in Table 5.3. The detailed output is available in the artifact [86].

5.4.4 Experiment 4: Verification of invariants for unbounded client-server systems using DCModelChecker 2.0

We have additionally verified client-server properties specified in the LTL_{LIA} language on our synthetic benchmark APS, against LTL_{LIA} specifications. These cannot be verified using DCModelChecker 1.0 [86]. The specifications are available in our artifact [85]. Our experiments demonstrate that DCModelChecker 2.0 has an advantage over DCModelChecker 1.0 in being able to additionally verify invariants. And it can be added to the arsenal of BMC tools for Petri nets.

5.4.5 Related Petri net verification tools

There are several tools such as KREACH [45], Petrinizer [47], QCOVER [24], ICOVER [53] verifying specific classes of properties like reachability and coverability of Petri nets. Recently, in [8, 9] the generalized reachability of Petri net is encoded into a BMC problem and solved using SMT solvers. However, BMC for unbounded Petri nets with true concurrency and LTL properties while leveraging the power of SMT Solvers remained unexplored until now. DC-ModelChecker bridges this gap.

The Model Checking Contest (MCC) [68] has attracted many tools for formal verification of concurrent systems that use a portfolio approach. In particular, we looked at ITS-Tools [99] which implements structural reduction techniques [100] and uses a layer of SMT when solving LTL. Moreover, it does not perform BMC. While ITS-Tools is much faster on bounded nets (MCC benchmarks) it gives only binary answers (sat or unsat). From our experiments, on a subset of unbounded nets, while Tapaal is faster, it also gives binary answers. On the contrary, our tool preserves the original Petri net structure and additionally gives a counterexample trace which is useful in bug finding in systems.

Conclusion

In this chapter, we discussed the first of its kind verification tool for checking properties of unbounded client server systems using $2D$-BMC. Next, we shall look at an extension of the Petri net, called ν-net and a different and richer logic, a variant of First Order logic, to express a different set of specifications of unbounded client server systems in the upcoming chapter.

Chapter 6

First Order LTL Tool with Monodic restriction

6.1 FO LTL over Petri nets with names

Formal verification of Petri nets is well studied. In practice, many communication protocols, services and applications are client-server systems. These systems can be modeled as Petri nets and verified using existing tools. However, these tools are not specifically suited for unbounded client-server systems as they do not allow the user to explicitly specify client and server properties as well as their unboundedness. Traditional logics such as LTL or CTL are not suitable for expressing these properties either. It is necessary to find suitable logics to express properties of unbounded client-server systems where the number of clients is not known a priori and the clients are distinguishable. To specify the properties of such systems, we consider a one variable fragment of Monadic First Order Temporal Logic, called FOTL_1 which is described in detail in Section 3.3. We propose a restriction on the ν-net to model the unbounded client server system.

6.1.1 Modeling an Autonomous Parking System

We consider as the running example, the Autonomous Parking System (APS) that manages parking lots and solves the search for parking lots by vehicles in crowded spaces through communication between the system (server) and the vehicle (client). This system has been successfully implemented by the industry [29, 108]. This is a type of single server multiple client system, where the clients are unbounded. The state diagrams for the server and client are given

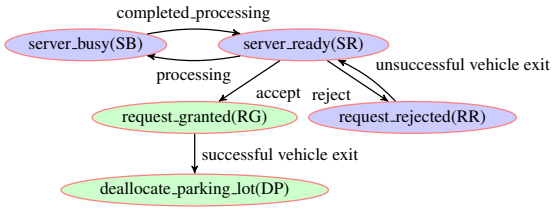


Figure 6.1: State diagram of server

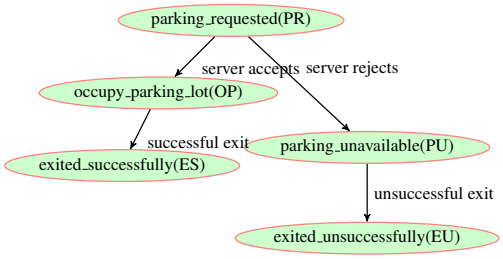


Figure 6.2: State diagram of client

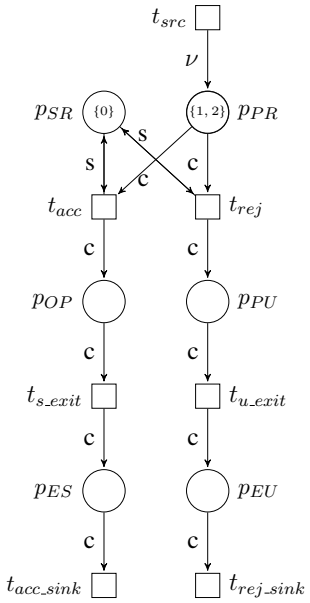


Figure 2.6: A restricted ν-net modeling APS

in Fig. 6.1 and Fig. 6.2 respectively. The combined interactions between clients and the server are modelled as a Petri net in Fig. 2.6. In this Petri net there are two types of places- client places and server places that model the client states and server states respectively. There is at most one token in the server place, and there can be an unbounded number of tokens in the client places. The number of tokens in the client places exactly corresponds to the number of clients in that state at any point in time.

Initially, the system is in the state *server_ready (SR)*, which is ready to service the requests. To keep the system sufficiently occupied, we assume a steady inflow of requests for parking, When a client inquires about parking space, the client is in the *parking_requested (PR)* state. The server 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, etc. We assume two disjoint workflows for each scenario. First, if the server accepts the request, the server is in

request_granted (RG) state and simultaneously, the client goes to *occupy_parking_lot (OP)* state. At some point, the client gives up its allocated parking space, is in *exit_parking_lot_successfully* and simultaneously the server is in *deallocate_parking_lot (DP)* state. This marks the successful exit of the client from the system. Second, if the server rejects the request, the client is in *parking_unavailable (PU)* state and the server is in *request_rejected (RR)* state. The only option is for the client to exit. At any point, the server can either accept or reject the request. After granting the request, the server can go to *server_busy (SB)* state. Theoretically, this description allows for an unbounded number of client requests to be processed by the server, albeit there may be limitations on the availability of 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 server and clients described above can be interleaved and modelled as a single ν-net as in Fig. 2.6.

In this ν-net, the places p_{PR}, p_{OP} and p_{ES} corresponds to the client states *parking_requested*, *occupy_parking_lot* and *exit_parking_lot*, the places p_{SR}, p_{SB}, p_{RG} and p_{DP} correspond to the server states *server_ready*, *server_busy*, *request_granted* and *deallocate_parking_lot*. The transitions in the net correspond to the transitions in the state diagrams of the server and client. The transitions t_{acc}, t_{exit}, t_{acc_sink}, represent the accept, exit and the sink transition respectively. The firing of transition t_{src} acts as the source, where an unbounded number of vehicle requests can be spawned. The firing of transition t_{cp} represents the server completing the processing of a request and returning to its ready state. In the unsuccessful scenario, the transition t_{rej} is fired when the server rejects the request, which brings the vehicle to *parking_unavailable* state represented by place p_{PU}, and the server goes to *request_rejected* state represented by place p_{RR}. The firing of transition t_{ve} ensures that the rejected vehicle exit is processed by the server, and it is ready to take on more vehicle requests. This is a type of single server multiple client system, where the clients are unbounded. The state diagrams for the server and client are given in Fig. 6.1 and Fig. 6.2, respectively. The combined interactions between clients and the server are modeled as a Petri net in Fig. 2.6.

To give a flavour of FOTL_1 and its expressibility, we enumerate some properties of APS that are not easily expressible in Linear Temporal Logic (LTL). Let P_s be the set of atomic propositions of the server and P_c be the set of client predicates. In the APS running example,

they are defined as follows:

$$P_c = \{parking\_requested\ (PR),\ occupy\_parking\_lot\ (OP),\ parking\_unavailable\ (PU),$$
$$exit\_successfully\ (ES),\ exit\_unsuccessfully\ (EU)\}$$
$$P_s = \{server\_ready\ (SR),\ server\_busy\ (SB), request\_granted\ (RG),$$
$$request\_rejected\ (RR),\ deallocate\_parking\_lot\ (DP)\}$$

1. When a vehicle requests a parking space, it is always the case that for every vehicle, it eventually exits the system, either successfully after being granted a parking space, or unsuccessfully, when its request is denied.

$$\psi_1 = \mathbf{G}_s(\forall x)\Big(parking\_requested(x) \Rightarrow \mathbf{F}_c\ (exit\_successfully(x)\ \vee\ exit\_unsuccessfully(x))\Big)$$

2. It is always the case that if the client occupies a parking lot, it will eventually exit the parking lot.

$$\psi_2 = \mathbf{G}_s(\forall x)\Big(occupy\_parking\_lot(x) \Rightarrow \mathbf{F}_c(exit\_successfully(x))\Big)$$

3. There may be clients whose requests are rejected.

$$\psi_3 = \mathbf{G}_s(\exists x)\big(parking\_requested(x) \wedge \mathbf{F}_c(exit\_unsuccessfully(x))\big)$$

4. There may be clients whose requests are rejected who wait in parking unavailable state until they are able to exit the system.

$$\psi_4 = \mathbf{G}_s(\exists x)\Big(parking\_requested(x) \wedge \mathbf{F}_c\big(parking\_unavailable(x)\ \mathbf{U}_c\ exit\_unsuccessfully(x)\big)\Big)$$

6.2 Tool and Implementation

The verification tool follows the same underlying architecture as DCModelChecker 2.0, where we employ Z3 to perform SMT queries using Linear Integer Arithmetic theory. We make use of ANTLR to pre-process the ν-net, as previously discussed. We verify the FOTL_1 properties of the APS case study and report them here. Since this is a first of its kind, there are no similar tools to compare the results of the model checker. However, we can see from the liveness windows that the counterexample or UNSAT obtained from the tool are justified.

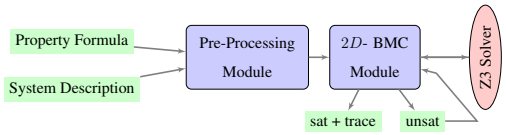


Figure 5.2: DCModelChecker 3.0 architecture

6.2.1 Satisfiability relations

Given a valid model $M = (\nu, V, \xi)$, the satisfiability relations for the logic, $\models$ and $\models_x$ can be defined, via induction over the structure of the server formulae $\psi \in \Psi$, and client formulae $\alpha \in \Delta$, respectively. We selectively describe some of the important relations that provide insight into the others.

1. $M, i \models q$ iff $q \in \nu_i$.

   The atomic server formula $q \in P_s$ is satisfied in the model M at instant i if and only if q is a member of the local state of the server ν_i at instant i, where $i \in \mathbb{N}_0$.

3. $M, i \models (\exists x)\alpha$ iff $\exists a \in CN$, $a \in V_i$ and $M, [x \mapsto a], i \models_x \alpha$.

   The client formula α is satisfied at some instant i if and only if α is satisfied in any of the *live* clients in V_i i.e, for some client $a \in CN$ and the client name $a \in V_i$, where $i \in \mathbb{N}_0$.

8. $M, i \models \mathbf{F_s}\psi$ iff $\exists j \geqslant i$, $M, j \models \psi$.

   The server formula $\mathbf{F_s}\psi$ is satisfied at some instant i if and only if there is some instant $j \geqslant i$, where the formula ψ holds.

16. $M, [x \mapsto a], i \models_x \mathbf{F}_c\alpha$ iff $\exists j \geqslant i$, $a \in V_j$, $M, [x \mapsto a], j \models_x \alpha$.

    The client formula $\mathbf{F}_c\alpha$ is satisfied at some instant i if and only if there is some instant $j \geqslant i$, where the formula α holds in any of the *live* clients in V_j.

6.2.2 Observations

We list down some observations about the logic FOTL_1 which supplement our understanding of the language.

Definition 6.2.1. *Ghost Interval Given a temporal property of a client, whose right boundary is λ' such that the $\lambda' < \lambda$. There is a ghost interval between λ and λ' where the formula α need not be asserted.*

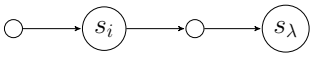


Figure 3.5: Bounded loop-free path of length λ

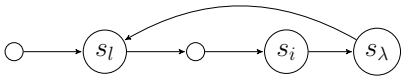


Figure 3.6: Bounded path with (λ, l) - loop

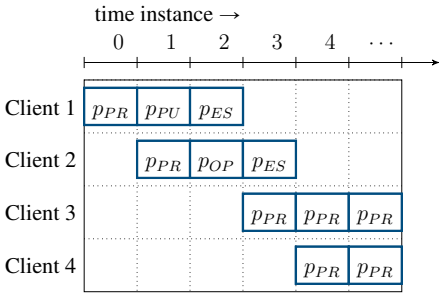


Figure 3.7: Snapshot of the running example (APS) depicting live windows

In Fig. 3.7, there is a *ghost interval* for client 1 between 1 (right boundary) and 5 (the bound λ).

Let $\psi_5 = (\forall x)\Big(\mathbf{G}_c\big(parking\_requested(x)\mathbf{U}_c\ exit\_successfully(x)\big)\Big)$. In ψ_5, $\mathbf{G}_c$ holds only till the *live window* ends for the corresponding client i.e, this will be asserted only as long as the client is *live*. Similarly, all other client temporal modalities $\mathbf{F}_c$, $\mathbf{U}_c$, and $\mathbf{X}_c$ need not be asserted beyond the *live window* of that particular client.

At any given time, there are as many client lassos as there are clients, which are then combined to construct the server lasso. For currently *active* clients the right boundary will be the execution bound λ.

We list down some of the properties that cannot be expressed in FOTL_1:

1. Consider the property, where there is at least one token in either place p_{SR} or p_{SB} or p_{RR}. From the net, $p_{SR} + p_{SB} + p_{RR} = 1$ is an invariant. These places represent the server being available, busy and the server rejecting the request.

2. The following property is not expressible in FOTL_1: It is always the case that the client request with an identifier (say) 1001 is rejected.

3. It is always the case that if the client request is accepted at time instance x, then the client exits the parking space at time instance z, $z > x$.

4. Formulas with quantifiers before server temporal modality are not permitted.

5. We do not have formulas of type, client i satisfies a formula whereas client j doesn't satisfy the same formula. We do not have equality/inequality to differentiate between the clients. We assume that all clients behave the same way and are of the same type. In the future, this model could be extended to clients of a finite set of types.

6. We cannot compare clients in the same state.

$$\psi_6 = \mathbf{G}_s((\exists x)(\exists y)parking\_requested(x) \wedge parking\_requested(y) \wedge \mathbf{F}_c(exit\_successfully(x) \wedge \mathbf{F}_c(exit\_successfully(y)))$$

7. Here, there are no free variables either in the scope of $\mathbf{G_s}$ or $\mathbf{F_s}$.

$$\text{Let } \psi_7 = \big(\mathbf{G_s}(\exists x)\ parking\_requested(x)\big) \wedge \big(\mathbf{F_s}(\exists y)\ parking\_requested(y)\big)$$

The formula ψ_7 states that it is always the case that there is a client who has requested parking and eventually, there exists a client who has also requested parking. However, we cannot compare these two clients in our logic.

The ease of expressibility of the client and server behaviour is the key motivation behind the logic FOTL_1 which is formally described in Sec. 3.3.

6.3 Bounded Semantics

In this section, we describe the bounded semantics of FOTL_1 in order to arrive at the SMT encoding, which is necessary for BMC. We use $\models^k$ as a restriction on $\models$, where k is the bound in the BMC strategy. To tackle the unbounded number of clients as well as unboundedness in the unfolding of the model, it is necessary to introduce two bounds: κ is a bound on the number of clients and λ is a bound on the execution steps of the net. The two parameters κ and λ are independent of each other. Since the runs are bounded, there are atmost κ clients in the system, namely CN = $\{0, \ldots, \kappa - 1\}$.

First, we describe the bounded semantics without loop for a subset of formulas, as shown in Fig. 3.5 where $0 \leqslant i \leqslant \lambda$, where i is the current instance on the bounded path.

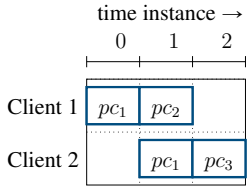


Figure 6.3: Snapshot at bound = 2

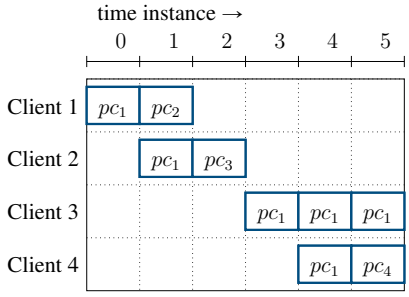


Figure 6.4: Snapshot at bound = 5

3. $M, i \models^k (\exists x)\alpha$ iff $\exists a \in CN$, $a \in V_i$ and $M, [x \mapsto a], i \models^k_x \alpha$.

   The above formula is satisfied when there is at least one live client such that α is satisfied in the model at instance i for the particular live client a. Given $\alpha = pc_1 \vee pc_2$ and the snapshot of the system with the bound 2 is in Fig. 6.3. Here, $CN = \{1, 2\}$. At instance $i = 0$, model M satisfies formula α. Hence, this formula holds true at instance 0.

4. $M, i \models^k (\forall x)\alpha$ iff $\forall a \in CN$, if $a \in V_i$ then $M, [x \mapsto a], i \models^k_x \alpha$.

   The above formula is satisfied when the α is satisfied in the model at instance i for all live clients in the set CN. Given $\alpha = pc_4$ and Fig. 6.3. The model M does not satisfy the formula for all clients, since client 2 does not satisfy α.

8. $M, i \models^k \mathbf{F_s}\psi$ iff $\exists j : i \leqslant j \leqslant \lambda$, $M, j \models^k \psi$.

   This formula is satisfiable if there is some instance j such that $i \leqslant j \leqslant \lambda$ at which the property ψ holds. Given $\psi = (\exists x)pc_4$. In Fig. 6.3, the formula is unsatisfiable. However, in Fig. 6.4, it is satisfiable at instance 5, where client 4 is in local state pc_4.

16. $M, [x \mapsto a], i \models^k_x \mathbf{F}_c\alpha$ iff $\exists j : i \leqslant j \leqslant \lambda$, $a \in V_j$ and $M, [x \mapsto a], j \models^k_x \alpha$.

This formula is satisfiable if there is some instance j such that $i \leqslant j \leqslant \lambda$ at which the property α is satisfied for the client a and the client a is a live client at instance j. Given $\alpha = pc_2$. In Fig. 6.3 and Fig. 6.4, the formula is satisfiable at instance 1 due to client 1.

Notice that in the bounded semantics equations 15-18, the semantics depend crucially on whether a is a live agent. Similarly, we can describe the bounded semantics with (λ, l) - loop similar to the (k, l) - loop in [21]. As shown in Fig. 3.6, i is the current instance on the bounded path, where the formula is evaluated, λ is the bound and l is the start position of the loop, where $0 \leqslant i \leqslant \lambda$ and $0 \leqslant l \leqslant \lambda$. Consider the following lemma that relates the bounded and unbounded semantics of logic FOTL_1.

Lemma 1. *Let f be a FOTL_1 formula and Π be a path then* $\Pi \models^i_k f \implies \Pi \models^k f$.

Proof Sketch: The proof proceeds by performing a case-by-case analysis of the client and server formulae. For instance, consider the server atomic formula $f = q$. It needs to be proved that $\Pi \models^i_k q \implies \Pi \models^i q$. Consider the left hand side.

- $\Pi \models^i_k q$

- IFF $q \in v_i$ (Using the semantics described in Section 6.3)

- IFF $\Pi \models^i q$

Lemma 1 is similar to the one in [21] and the complete proof is left to the reader based on the sketch given above.

Theorem 6.3.1. *UCSTL is decidable.*

Proof. MFOTL is decidable [60]. UCSTL is a syntactic fragemnt of MFOTL which is monodic and restricted to quantifier rank 1. □

6.4 Related Work

An $MFOTL$ formula φ is monodic if every subformula has at most one free variable, in the scope of ψ's. Decidability is shown by encoding models of a monodic sentence φ in structures called quasimodels and then expressing the statement "there exists a quasimodel satisfying a given monodic sentence" as a monadic second order sentence. In [59], they show that the

monodic fragment is also $EXPSPACE$-complete. In this work, we propose a one-variable fragment of $MFOTL$, called FOTL_1 for unbounded client-server systems. Additionally, we have provided the bounded semantics which led to the SMT encoding of this logic. The SMT encoding is desirable for implementing a bounded model checking tool where the properties are specified in this logic for unbounded client-server systems.

As part of future work, we would like to build a BMC tool that takes in FOTL_1 specifications for unbounded client-server systems based on the SMT encoding given in this work. Building verification tools for unbounded client-server systems is a non-trivial engineering pursuit. The BMC implementation using FOTL_1 specifications will join the arsenal of tools such as KREACH [45], Petrinizer [47], QCOVER [24], ICOVER [53] for nets. We can also have multiple types of clients in the unbounded client-server systems, which may model a richer set of systems. Alongside this, the complexity and decidability of model checking unbounded client-server systems using FOTL_1 specifications is yet to be explored.

Conclusion

In this chapter, we discussed a verification tool for Petri nets with identifiers and a variant of First Order Logic. In the upcoming chapter, we consider Petri nets with nesting and look at suitable temporal properties on them and build a tool to verify them as well.

Chapter 7

Verification of Elementary Object Systems

7.1 Introduction

Imperfections are natural in the context of message passing systems: imperfect communication channels may spontaneously lose, duplicate, or shuffle carried messages or even deliver new unwanted ones. When compared to their perfect counterpart, verification of imperfect systems is usually easier. For example, reachability in Communicating Finite Automata (CFA) over *perfect* (FIFO) channels is undecidable [26], but it is decidable when the channels are *lossy* [31] (messages may be non-deterministically lost and never delivered) or *unordered* [7] (the message sending and reception order may be different). The same principle generally holds even outside CFA. Specifically, two counter machines can encode Turing Machines [43, 79] and suffer from undecidable reachability. Instead, Mayr [75, 76] showed that if lossiness is applied to the counters (the natural numbers stored in the counters may non-deterministically decrease), then reachability becomes decidable; nevertheless, several other problems remain undecidable. Similar results are also available for Petri Nets (PNs): Bouajjani and Mayr [25] studied the impact of lossiness on the model checking problem of Vector Addition Systems with States (VASS, equivalent to Petri Nets) against fragments of the Modal μ-Calculus. In this case, while the EF (and EG) fragment of the UB language (which includes negation and labeled variants of the CTL next operator) is undecidable for VASS [46], it is decidable for lossy VASS.

Imperfections may be naturally interpreted as perturbations of the system configurations and, thus, verification of imperfect systems can be studied under the lens of robustness [16, 66], i.e., checking whether a property holds if at most k perturbations occur, for some given $k \leqslant |\mathbb{N}|$. Recently, Köhler-Bussmeier and Capra [66] put forward the nets-within-nets paradigm [105] to

naturally specify robustness properties in a Multi Agent System (MAS) context. Specifically, in the PN setting, *black tokens* that mark a fixed set of places are moved around by a fixed set of transitions. In contrast, in nets-within-nets, the tokens could additionally be PN objects themselves. Thus, even without further expert information on the net design, these *object tokens* naturally model agents, which might be affected by perturbations. In [66], perturbations follow a drastic *let it crash* approach, causing agent break-downs. Technically, this is achieved by enforcing a deadlock in the object token. However, nets-within-nets with less disruptive perturbations may still be suitable to model perturbed MAS, where the agents may suffer imperfections even without completely breaking down.

We focus on Elementary Object Systems (EOSs) [65], which are a simple nets-within-nets model, yet featuring most of the important ingredients. Moreover, they can be generalized to more sophisticated models, such as full-fledged Object Systems [105]. This makes EOSs an excellent candidate for our study. Our key contributions are:

1. We formally define three forms of lossiness in the EOS setting, corresponding to the nesting levels of the tokens.
2. We provide examples illustrating the relevance of these lossiness relations and formalize lossy reachability/coverability problems on them.
3. We completely chart the decidability status of these problems (see Tab. 7.1).

Standard reachability/coverability problems have been studied in [65], but only in a perfect setting, i.e., without lossiness. Preliminary results on EOS robustness were put forward in [66]. However, they do not address the problem of perturbations on reachability/coverability in a systematic way. To the best of our knowledge, ours is the first work that attempts a full formal classification of EOS reachability/coverability with lossy perturbations. As discussed above, the concept of lossiness is well established in the PN literature; the most relevant works are [25, 76, 96]. However, their approach uniformly interleaves each standard step with lossiness and, thus, does not properly address robustness, where also the number of lossy events is of interest. Moreover, when compared to PNs, EOSs offer more room for lossiness because of the nesting of tokens. This may significantly complicate the picture. Instead, we consider the full spectrum of lossiness, from one occurrence to infinitely many, at all nesting levels.

7.2 Preliminaries

7.2.1 Binary Relations

Let us fix some notation for binary relations. Given a set X, the *identity relation* id_X on X is the relation $\{(x, x) \in X^2 \mid x \in X\}$. Given a binary relation $<$ on X, we denote its *reflexive closure* $< \cup id_X$ by $\leqslant$ and its *anti-reflexive part* $< \setminus id_X$ by $\lneq$. We use the symbol $>$ to denote the relation such that, $x > y$ iff $y < x$. For example, if $<$ is transitive, then $\leqslant$ and $\geqslant$ are transitive and reflexive (i.e., quasi orders). The same applies to the symbols $\prec$ and $\succ$ and their closures. From now on, we use $\prec$ to denote arbitrary transitive relations, and $<$ (possibly with a subscript) to represent fixed transitive relations, e.g., the standard order of $\mathbb{N}$.

7.2.2 Multisets

A *multiset* $\boldsymbol{m}$ on a set D is a mapping $\boldsymbol{m} : D \to \mathbb{N}$. The *support* of $\boldsymbol{m}$ is the set $\texttt{Support}(m) = \{i \mid \boldsymbol{m}(i) > 0\}$. The multiset $\boldsymbol{m}$ is finite if its $\texttt{Support}(\boldsymbol{m})$ is finite. The family of all multisets over D is denoted by $D^{\oplus}$. We denote a finite multiset $\boldsymbol{m}$ by enumerating the elements $d \in \texttt{Support}(\boldsymbol{m})$ exactly $\boldsymbol{m}(d)$ times in between $\{\{$ and $\}\}$, where the ordering is irrelevant. For example, the finite multiset $\boldsymbol{m} : \{p, q\} \longrightarrow \mathbb{N}$ such that $\boldsymbol{m}(p) = 1$ and $\boldsymbol{m}(q) = 2$ is denoted by $\{\{p, q, q\}\}$. The empty multiset $\{\{\}\}$ (with empty support) is also denoted by $\varnothing$. On the empty domain $D = \varnothing$ the only defined multiset is $\varnothing$; to stress this out we denote this special case, i.e., the empty multiset over the empty domain, by ε. Given two multisets $\boldsymbol{m_1}$ and $\boldsymbol{m_2}$ on D, we define $\boldsymbol{m_1} + \boldsymbol{m_2}$ and $\boldsymbol{m_1} - \boldsymbol{m_2}$ on D as follows: $(\boldsymbol{m_1} + \boldsymbol{m_2})(d) = \boldsymbol{m_1}(d) + \boldsymbol{m_2}(d)$ and $(\boldsymbol{m_1} - \boldsymbol{m_2})(d) = max(\boldsymbol{m_1}(d) - \boldsymbol{m_2}(d), 0)$. Similarly, for a finite set I of indices, $\sum_{i \in I}\{\{d_i\}\}$ denotes the multiset $\boldsymbol{m}$ over $\bigcup_{i \in I}\{d_i\}$ such that $\boldsymbol{m}(d) = |\{i \in I \mid d_i = d\}|$ for each $d \in D$. With a slight abuse of notation, we omit the double brackets, i.e., $\sum_{i \in I}\{\{d_i\}\} = \sum_{i \in I} d_i$. If $I = \{1, \ldots, n\}$, then $\sum_{i \in I} d_i = \sum_{i=1}^{n} d_i$. Finally, we write $\boldsymbol{m_1} \sqsubseteq \boldsymbol{m_2}$ if, for each $d \in D$, we have $\boldsymbol{m_1}(d) \leqslant \boldsymbol{m_2}(d)$.

7.2.3 Petri Nets

We assume that the reader is familiar with standard PNs. Here we just fix the notation (see, e.g., [80]). We denote a PN N as a tuple $N = (P, T, F)$, where P is a finite place set, T is a finite transition set, and F is a flow function. Where useful, we equivalently interpret F via the

functions $\mathtt{pre}_N : T \to (P \to \mathbb{N})$ where $\mathtt{pre}_N(t)(p) = F(p,t)$ and $\mathtt{post}_N : T \to (P \to \mathbb{N})$ where $\mathtt{post}_N(t)(p) = F(t,p)$. For example, a transition $t \in T$ is enabled on a marking μ (finite multiset of places) if, for each place $p \in P$, we have $\mathtt{pre}_N(t)(p) \leqslant \mu(p)$. Its firing results in the marking μ' such that $\mu'(p) = \mu(p) - \mathtt{pre}_N(t)(p) + \mathtt{post}_N(t)(p)$, for each $p \in P$. We denote markings according to multiset notation. For example, the marking μ that places one token on place p and two on place q is denoted by $\{\{p, q, q\}\}$. The empty marking is denoted by $\varnothing$. We also work with the special *empty PN* $\blacksquare = (\varnothing, \varnothing, \varnothing)$, whose only marking is ε.

7.2.4 Elementary Object Systems

An EOS [65] is, intuitively, a PN (called system net) whose tokens carry an internal PN (called object net), taken from a finite set $\mathcal{N}$. Each place can host only one fixed type of internal PN. The EOS fires events, which synchronize a transition τ in the system net and multisets $\theta(N)$ of transitions in the object nets N consumed by τ.

Definition 1 (EOS). An *EOS* $\mathfrak{E}$ is a tuple $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ where:

1. $\hat{N} = \langle \hat{P}, \hat{T}, \hat{F} \rangle$ is a PN called *system net*; $\hat{T}$ contains a special set $ID_{\hat{P}} = \{id_p \mid p \in \hat{P}\} \subseteq \hat{T}$ of *idle transitions* such that, for each distinct $p, q \in \hat{P}$, we have $\hat{F}(p, id_p) = \hat{F}(id_p, p) = 1$ and $\hat{F}(q, id_p) = \hat{F}(id_p, q) = 0$.
2. $\mathcal{N}$ is a finite set of PNs, called *object PNs*, such that $\blacksquare \in \mathcal{N}$ and if $(P_1, T_1, F_1), (P_2, T_2, F_2) \in \mathcal{N} \cup \hat{N}$,[1] then $P_1 \cap P_2 = \varnothing$ and $T_1 \cap T_2 = \varnothing$.
3. $d : \hat{P} \to \mathcal{N}$ is called the *typing function*.
4. Θ is a finite *set of events* where each *event* is a pair $(\hat{\tau}, \theta)$, where $\hat{\tau} \in \hat{T}$ and $\theta : \mathcal{N} \to \bigcup_{(P,T,F) \in \mathcal{N}} T^{\oplus}$, such that $\theta((P,T,F)) \in T^{\oplus}$ for each $(P,T,F) \in \mathcal{N}$ and, if $\hat{\tau} = id_p$, then $\theta(d(p)) \neq \varnothing$.

Since the nets in $\mathcal{N}$ are disjoint, we can denote each event $\langle \hat{\tau}, \theta \rangle$, as a pair $\langle \hat{\tau}, M \rangle$ for a multiset M over $\bigcup_{(P,T,F) \in \mathcal{N}} T$ such that $M(t) = \theta(N)(t)$ where $N = (P,T,F) \in \mathcal{N}$ and $t \in T$. EOS tokens are nested, i.e., each token at a system place p carries a PN marking μ for the object net $d(p)$. EOS markings, also called nested markings, are multisets of nested tokens. With a slight abuse of notation, we denote markings omitting double curly brackets from multiset notation.

[1] This way, the system net and the object nets are pairwise distinct.

Definition 2 (Nested Markings)**.** Let $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ be an EOS. The set of *nested tokens* $\mathcal{T}(\mathfrak{E})$ of $\mathfrak{E}$ is the set $\bigcup_{(P,T,F)\in\mathcal{N}}(d^{-1}(P,T,F) \times P^{\oplus})$. The set of *nested markings* $\mathcal{M}(\mathbf{E})$ of $\mathfrak{E}$ is $\mathcal{T}(\mathfrak{E})^{\oplus}$. Given $\lambda, \rho \in \mathcal{M}(\mathbf{E})$, we say that λ is a *sub-marking* of μ if $\lambda \sqsubseteq \mu$.

Note that λ is a sub-marking of μ iff there is some nested marking μ' such that $\mu = \lambda + \mu'$. EOSs inherit the graphical representation of PNs with the provision that we represent nested tokens via a dashed line from the system net place to an instance of the object net where the internal marking is represented in the standard PN way. However, if the nested token is $\langle p, \varepsilon \rangle$ for a system net place p of type ■, we represent it with a black-token ■ on p. If a place p hosts $n > 2$ black-tokens, then we represent them by writing n on p. Each event $\langle \hat{\tau}, \theta \rangle$ is depicted by labeling $\hat{\tau}$ by $\langle \theta \rangle$ (possibly omitting double curly brackets). If there are several events involving $\hat{\tau}$, then $\hat{\tau}$ has several labels. Fig. 2.11 shows the client server system with unbounded number of clients represented as an EOS.

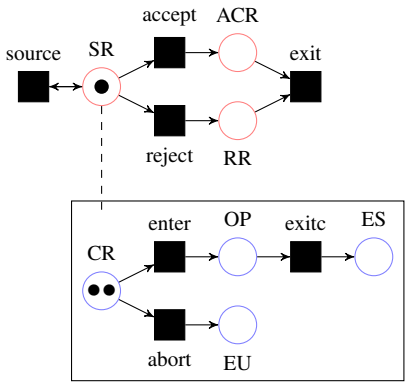


Figure 2.11: EOS modeling the single server (unbounded) multiple client system

Example 5. Fig. 7.1 depicts the system net $\hat{N}$ (the idle transitions are omitted) and object net `drone` of an EOS $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ modeling a drone that *(1)* moves between a base and a field, *(2)* has two batteries, *(3)* consumes one charge-unit per battery per movement, and *(4)* charges its batteries by multiples of two charge-units when at base. Technically, $\mathcal{N} = \{\texttt{drone}, ■\}$ (even if ■ is unused), $d(\texttt{base}) = d(\texttt{field}) = \texttt{drone}$, and Θ synchronizes `takeOff` and `land` (respectively `charge`) in $\hat{N}$ with `move` (`charge1` and `charge2`) in `drone`. Formally, $\Theta = \{\langle \texttt{takeOff}, \{\{\texttt{move}\}\} \rangle, \langle \texttt{land}, \{\{\texttt{move}\}\} \rangle, \langle \texttt{charge}, \{\{\texttt{charge1}\}\} \rangle, \langle \texttt{charge}, \{\{\texttt{charge2}\}\} \rangle\}$. The marking $\mu = \langle \texttt{drone}, \{\{\texttt{batt1}, \texttt{batt1}\}\} \rangle$ represents a single partially charged drone at `base`, with two charge units in the first battery.

When firing an event $\langle \tau, \theta \rangle$, nested tokens in the system net are consumed according to the preconditions of τ in the standard PN way. At the same time, for each object net N, the inner

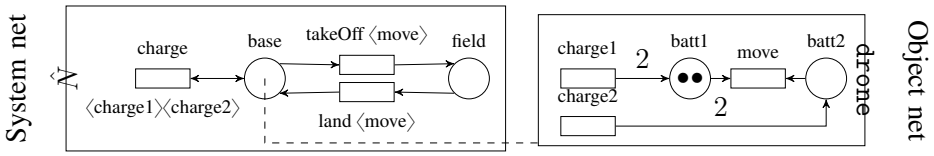


Figure 7.1: EOS in Example 5 with marking $\{\{\langle \texttt{drone}, \{\{\texttt{batt1}, \texttt{batt1}\}\}\rangle\}\}$. The idle transitions are omitted.

tokens are merged so as to obtain a PN marking $\mu(N)$ for N (possibly empty). Then, transitions in $\theta(N)$ are fired in the standard PN way obtaining markings $\mu'(N)$. Next, nested markings with empty inner markings are produced in the system net according to the postconditions of τ. Finally, the markings $\mu'(N)$ are non-deterministically distributed among the empty nested tokens, according to the typing function. To be fired, the event must be enabled at both the system and at the object net level. This is captured by the enabledness condition, which makes use of projection operators at the system (Π^1) and at the object net level (Π^2_N for each $N \in \mathcal{N}$).

Definition 3 (Projection Operators)**.** Let $\mathfrak{E}$ be an EOS $\langle \hat{N}, \mathcal{N}, d, \Theta \rangle$. The *projection operators* Π^1 maps each nested marking $\mu = \sum_{i \in I} \langle \hat{p}_i, M_i \rangle$ for $\mathbf{E}$ to the PN marking $\sum_{i \in I} \hat{p}_i$ for $\hat{N}$. Given an object net $N \in \mathcal{N}$, the *projection operators* Π^2_N maps each nested marking $\mu = \sum_{i \in I} \langle \hat{p}_i, M_i \rangle$ for $\mathbf{E}$ to the PN marking $\sum_{j \in J} M_j$ for N where $J = \{i \in I \mid d(\hat{p}_i) = N\}$.

To define the enabledness condition, we need the following notation. We set $\mathtt{pre}_N(\theta(N)) = \sum_{i \in I} \mathtt{pre}_N(t_i)$ where $(t_i)_{i \in I}$ is an enumeration of $\theta(N)$ counting multiplicities. We analogously set $\mathtt{post}_N(\theta(N)) = \sum_{i \in I} \mathtt{post}_N(t_i)$.

Definition 4 (Enabledness Condition)**.** Let $\mathfrak{E}$ be an EOS $\langle \hat{N}, \mathcal{N}, d, \Theta \rangle$. Given an event $e = \langle \hat{\tau}, \theta \rangle \in \Theta$ and two markings $\lambda, \rho \in \mathcal{M}(\mathfrak{E})$, the *enabledness condition* $\Phi(\langle \hat{\tau}, \theta \rangle, \lambda, \rho)$ holds iff

$$\Pi^1(\lambda) = \mathtt{pre}_{\hat{N}}(\hat{\tau}) \ \wedge \Pi^1(\rho) = \mathtt{post}_{\hat{N}}(\hat{\tau}) \ \wedge \forall N \in \mathcal{N},\ \Pi^2_N(\lambda) \geqslant \mathtt{pre}_N(\theta(N)) \ \wedge$$
$$\forall N \in \mathcal{N},\ \Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathtt{pre}_N(\theta(N)) + \mathtt{post}_N(\theta(N))$$

The event e is *enabled with mode* (λ, ρ) *on a marking* μ iff $\Phi(e, \lambda, \rho)$ holds and $\lambda \sqsubseteq \mu$. Its firing results in the step $\mu \xrightarrow{(e,\lambda,\rho)} \mu - \lambda + \rho$.

Example 6. In the setting of the EOS $\mathfrak{E}$ and marking μ in Ex. 5 (Fig. 7.1), the event $\langle \texttt{charge}, \{\{\texttt{charge1}\}\} \rangle$ is enabled on $\mu = \langle \mathit{base}, \{\{\texttt{batt1}, \texttt{batt1}\}\} \rangle$ with mode (λ, ρ) where $\lambda = \mu$ and $\rho = \langle \mathit{base}, \{\{\texttt{batt1}, \texttt{batt1}, \texttt{batt1}, \texttt{batt1}\}\} \rangle$. Since $\lambda = \mu$, its firing results in the

step $\mu \xrightarrow{\langle e,\lambda,\rho\rangle} \rho$. Instead, the event $\langle \texttt{charge}, \{\{\texttt{charge2}\}\}\rangle$ is enabled on μ with mode (λ, ρ') where $\rho' = \langle base, \{\{\texttt{batt1}, \texttt{batt1}, \texttt{batt2}, \texttt{batt2}\}\}\rangle$. Its firing results in the step $\mu \xrightarrow{\langle e,\lambda,\rho'\rangle} \rho'$. These are the only enabling modes for $\langle \texttt{charge}, \{\{\texttt{charge1}\}\}\rangle$ and $\langle \texttt{charge}, \{\{\texttt{charge2}\}\}\rangle$ on μ. No other event is enabled on μ, irrespective of the mode.

The reachability problem for EOSs is defined in the usual way, i.e., whether there is a run (sequence of event firings) from an initial marking μ_0 to a target marking μ_f. Also coverability definition is standard, but with respect to the order $\leqslant_f$ (denoted by $\preceq$ in [65]; see Def. 6 below) that allows one to add both *(1)* tokens in the inner markings of available nested tokens *(2)* or nested tokens with some internal marking on the system net places. It is known that both these problems are undecidable (Th. 4.3 in [65]. However, coverability is decidable on the fragment of *conservative EOSs* (cEOSs; Th. 5.2 in [65]), where, for each system net transition t, if t consumes a nested token on a place of type N, then it also produces at least one token on a place of the same type N. Nevertheless, reachability remains undecidable (Th. 5.5 in [65]).

Definition 5 (cEOS)**.** A cEOS is an EOS $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta\rangle$ with $\hat{N} = \langle \hat{P}, \hat{T}, \hat{F}\rangle$ where, for all $\hat{t} \in \hat{T}$ and $\hat{p} \in \texttt{Support}(\texttt{pre}_{\hat{N}}(\hat{t}))$, there exists $\hat{p}' \in \texttt{Support}(\texttt{post}_{\hat{N}}(\hat{t}))$ such that $d(\hat{p}) = d(\hat{p}')$.

7.3 Problem

We study reachability and coverability of EOSs (cEOSs) affected by several forms of lossiness. First, we define three lossiness relations and show their relevance. Second, we formally define the problem we study in the following sections.

7.3.1 Lossy EOSs

We study EOSs (cEOSs) affected by lossiness, where nested markings may non-deterministically lose their tokens according to a quasi order, called lossiness relation. Lossiness can occur *(1)* at the object level, if lossiness removes only tokens from the inner markings of nested tokens, *(2)* at the system level, if lossiness removes whole nested tokens only, and *(3)* at both levels (the full EOS), if both whole nested tokens and/or regular tokens from the remaining nested tokens are removed. These levels are captured, respectively, by the lossiness quasi orders $\leqslant_o$ (object-lossiness), $\leqslant_s$ (system-lossiness), and $\leqslant_f$ (full-lossiness) as defined next.

Definition 6. Given an EOS $\mathfrak{E}$ and two nested markings μ and μ' for $\mathfrak{E}$, we have *(1)* $\mu \leqslant_s \mu'$ if $\mu \sqsubseteq \mu'$ or, equivalently, there is some μ'' such that $\mu' = \mu + \mu''$, *(2)* $\mu \leqslant_o \mu'$ if we can write $\mu = \sum_{i \in I} \langle \hat{p}_i, M_i \rangle$ and $\mu' = \sum_{i \in I} \langle \hat{p}_i, M'_i \rangle$ and, for each $i \in I$, $M_i \sqsubseteq M'_i$, and *(3)* $\mu \leqslant_f \mu'$ if there is some nested marking μ'' such that $\mu \leqslant_o \mu'' \leqslant_s \mu'$.

Example 7. Consider the marking $\mu = \langle \texttt{base}, \{\{\texttt{batt1}, \texttt{batt1}\}\} \rangle$ in Ex. 5. By removing 1 or 2 charge units we obtain the markings $\mu_1 = \langle \texttt{base}, \{\{\texttt{batt1}\}\} \rangle$ and $\mu_2 = \langle \texttt{base}, \varnothing \rangle$. By adding to μ a discharged token at place `field`, we obtain the $\mu' = \langle \texttt{base}, \{\{\texttt{batt1}, \texttt{batt1}\}\} \rangle + \langle \texttt{field}, \varnothing \rangle$. By removing the drone from μ, we obtain $\mu'' = \varnothing$. We have, among the others, $\mu' \geqslant_s \mu$, $\mu \geqslant_o \mu_1 \geqslant_o \mu_2$, $\mu' \geqslant_f \mu$, $\mu' \geqslant_f \mu_2$, and $\mu' \geqslant_f \mu''$.

The relation $\leqslant_f$ coincides with $\preceq$ in [65]. Moreover, the order of $\leqslant_o$ and $\leqslant_s$ is irrelevant in $\leqslant_f$ definition (Rem. 1 below). An EOS (cEOS) suffering from object-, system-, or full-lossiness is called, respectively, object-, system-, or full-lossy EOS (cEOS) or, simply lossy EOS (cEOS).

These lossiness relations are relevant to model non-deterministic phenomena not directly captured by the EOS. For example, in the context of Ex. 5, object-lossiness results in the non-deterministic loss of tokens at places `batt1` and `batt2`, which models the partial/total discharge of drone batteries because of non-modeled drone movements within the base or the field, or because of other unexpected phenomena. In a slightly more complex EOS, with intermediate places capturing the flight from base to field and vice-versa, object-lossiness captures also the non-deterministic usage of extra charge-units because of contingencies like strong winds. Instead, system lossiness results in the loss of nested tokens modeling drones. This captures the loss of drones because of, e.g., break-downs, wrong flight paths, or seizure from higher priority processes (assuming the EOS is a module in a more complex system). Full-lossiness capture both aspects. Similar interpretations can be given each time the (nested) tokens represent resources, like charge-units or drones in Ex. 5. These scenarios are common in the literature (see, e.g., water- and fire-units in [105] and raw-resources in production plants in [30]).

Lossy EOSs are relevant also to capture partial/total internal break-downs. This happens, e.g., when the tokens model resource containers instead of resources themselves. For example, after modifying the `drone` object net into the object net `drone2` in Fig. 7.2,[2] each regular token represents a battery with bounded capacity. Its charge level is captured by its position in the object net. Consequently, object-lossiness represents the break-down of internal components,

[2] Also the events have to be modified accordingly, i.e., for each $n \in \{0, 1, 2\}$, by synchronizing `takeOff` and `landing` with `discharge n` object net transitions, and `charge` with the `charge n` object net transitions.

in this case the battery. The loss of all batteries results in drone deadlock (cf. [66]). This case is analogous to the application of system-lossiness discussed above, since drones can be seen as internal components of a higher level process (captured by the whole EOS): the loss/break-down of all drones results in system deadlock. More in general, this interpretation applies when the EOS uses conservative system and/or object nets.[3] Also these scenario are common in the literature (see, e.g., the finite control of robots [65], the internal state of fire-fighters in [105], and customers and cars in [73]).

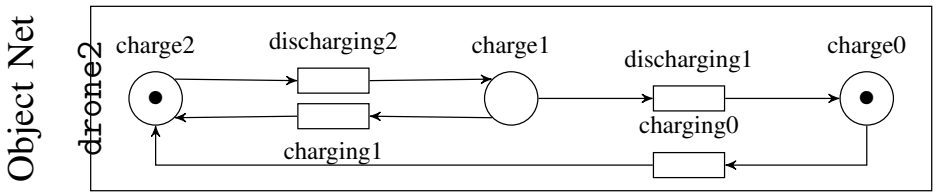


Figure 7.2: Object net `drone2` with one fully charged 2-bounded battery and a fully discharged one.

7.3.2 Lossy-reachability/coverability

We study the problem of ℓ-reachability/coverability, i.e., whether a target nested marking can be reached/covered from an initial one via a run suffering at most ℓ lossy steps, where $\ell \in \mathbb{N} \cup \{\omega\}$.[4] These problems are relevant to study EOS robustness in front of losses/break-downs.

Definition 7. Given a transition system $TS = (V, \rightarrow)$, a transitive relation $\prec$ on V, and an $\ell \in \mathbb{N} \cup \{\omega\}$, a $(\prec, \ell)$-*run* in TS is a run whose steps are labeled either by $\rightarrow$, called *standard steps*, or by $\succ$, called $\prec$-*lossy* or *lossy steps*, and at most ℓ steps are lossy. The set of $(\prec, \ell)$-runs from μ_0 is denoted by $Runs_\ell(\prec, \mu_0)$. A $(\prec, \ell)$-run is called ℓ'-*strong* if it contains exactly ℓ' lossy steps.

The definition also applies to reflexive or anti-reflexive transitive relations, i.e., we can also talk about $(\preceq, \ell)$-runs and $(\precnsim, \ell)$-runs. We denote a labeled step from μ to μ' by $\mu \rightsquigarrow \mu'$. To stress that the step is labeled by $\rightarrow$ or by $\prec$, we denote it by $\mu \rightarrow \mu'$ or by $\mu \succ \mu'$, respectively. Whenever we have a lossy run σ from μ to μ', we write $\mu \rightsquigarrow^\sigma \mu'$. The $(\prec, \ell)$-reachability/coverability problems ask whether a target can be reached/covered under $\prec$ from an initial configuration with at most ℓ $\prec$-lossy steps.

[3] A PN is conservative when each transition consumes and produces the same number of tokens.

[4] Recall that ω is the first limit ordinal, whose cardinality is $|\mathbb{N}|$, i.e., the same as $\mathbb{N}$.

Definition 8 ($(\prec, \ell)$-reachability/coverability for EOSs (cEOSs))**.** Let $\ell \in \mathbb{N} \cup \{\omega\}$.

Input: An EOS (cEOS) E, an initial marking μ_0 and a target marking μ_1 for E.

Output of reachability: Is there a run $\sigma \in Runs_\ell(\prec, \mu_0)$ such that $\mu_0 \leadsto^\sigma \mu_1$?

Output of coverability: Is there a run $\sigma \in Runs_\ell(\prec, \mu_0)$ such that $\mu_0 \leadsto^\sigma \mu \geqslant \mu_1$ for some μ?

We call these problems *lossy-problems*. A *$\prec$-lossy-problem* is a lossy-problem under $\prec$. The *degree* of a $(\prec, \ell)$-reachability/coverability problem is ℓ. If $\ell = 0$ we obtain standard reachability/coverability, i.e., over perfect runs. Our objective is to fully chart the decidability status of the lossy-problems for $\leqslant_f$, $\leqslant_s$, and $\leqslant_o$. Previous results for EOSs are available only for $\ell = 0$ and the relation $\leqslant_f$. Consequently, they do not inform us on the status of the other (proper) lossy-problems, whose study still requires a careful and in-depth analysis.

7.4 Coincident Problems

The well known notion of compatibility from WSTS has a strong impact on ℓ-reachability problems. In fact, we now show that all these problems, for $\ell \geqslant 1$ (including ω) and any quasi order $\preceq$, collapse to $(\preceq, 0)$-coverability if and only if the lossiness relation is compatible.

Lemma 2. *Each yes-instance of $(\preceq, \ell)$-reachability is also a yes instance of $(\preceq, \ell)$-coverability. [36]*

Proof. Immediate consequence of reflexivity of the quasi order $\preceq$. □

Lemma 3. *Each yes-instance of $(\preceq, \ell)$-coverability is also a yes instance of $(\preceq, \ell + 1)$-reachability, if ℓ is finite, and a yes-instance of $(\preceq, \omega)$-reachability, if $\ell = \omega$. [36]*

Proof. If μ_1 is coverable from μ_0, then there is a $(\preceq, \ell)$-run σ from μ_0 to μ and $\mu \geq \mu_1$. Take the run σ' as the run σ followed by the lossy step $\mu \geq \mu_1$. Thus, σ' reaches μ_1 from μ_0. Moreover, if ℓ is finite, then σ' is a $(\preceq, \ell + 1)$-run and, otherwise, σ' is a $(\preceq, \omega)$-run. □

Corollary 1. *$(\preceq, \omega)$-reachability and $(\preceq, \omega)$-coverability coincide. [36]*

Note that Cor. 1 is consistent with other lossy PN models (see, e.g., [25]).

Lemma 4. *Each yes-instance of $(\preceq, \ell)$-reachability or $(\preceq, \ell)$-coverability is also a yes instance of $(\preceq, \omega)$-reachability or $(\preceq, \omega)$-coverability, respectively. [36]*

Proof. Immediate consequence of the fact that each $(\preceq, \ell)$-run is also a $(\preceq, \omega)$-run. □

Summarising, for each quasi order $\preceq$, the $\preceq$-lossy-problems form a hierarchy ordered according to inclusion of the yes-instance sets. For each $i \in \mathbb{N}$, the i-th hierarchy level for $\preceq$ is the $(\preceq, i/2)$-reachability problem, if i is even, and the $(\preceq, (i-1)/2)$-coverability problem, if i is odd. We say that the hierarchy *collapses* if all the $(\preceq, \ell)$-reachability and $(\preceq, \ell)$-coverability problems with $\ell \geqslant 1$ coincide with (standard) $(\preceq, 0)$-coverability or, equivalently, if the yes-instances of $(\preceq, \omega)$-reachability are also yes-instances of $(\preceq, 0)$-coverability. The next lemma states that this latter property is equivalent to compatibility, that is, if $\mu_1 \succeq \mu_2 \to \mu_3$, then there is a μ_4 such that $\mu_1 \to^* \mu_4 \succeq \mu_3$.

Lemma 5. *$\preceq$ is compatible iff each yes-instance of $(\preceq, \omega)$-reachability is also a yes-instance of $(\preceq, 0)$-coverability. [36]*

Proof. Assume that $\preceq$ is compatible. If v_1 is reachable from v_0 via an ω-run σ, then, without loss of generality, we can assume that σ is finite and, thus, it is a ℓ-run for some finite ℓ. By compatibility, we can push, one by one, the finitely many lossy steps in σ at the end of the run, obtaining an ℓ-run σ' (possibly with different length) where all lossy steps occur at the end, i.e., σ' is of the form $v_0 \to^* w_1 \succeq w_2 \cdots \succeq w_\ell \succeq v_1$. By transitivity of $\succeq$, there is also a run σ'' of the form $v_0 \to^* w_1 \succeq v_1$, which witness that v_1 is $\preceq$-coverable from v_0. Vice-versa, assume that each yes-instance of $(\preceq, \omega)$-reachability is also a yes-instance of $(\preceq, 0)$-coverability. If $v_0 \succeq v_1 \to v_2$, then v_2 is $(\preceq, 1)$-reachable from v_0, as well as $(\preceq, \omega)$-reachable. Thus, v_2 is also coverable from v_0, i.e., $v_0 \to^* v_1 \succeq v_2$. □

Corollary 2. *The hierarchy of lossy-problems induced by $\preceq$ collapses iff $\preceq$ is compatible. [36]*

Note that Cor. 2 can be generalized to other lossy models, since its proof does not take advantage of the technical details of lossy EOSs, but relies only on a compatible quasi order. This fact has some immediate consequence on the lossy-problems we are studying. In fact, it is known that $\leqslant_f$ is strong compatible on cEOSs, that is, if $\mu_1 \geqslant_f \mu_2 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_3$, then there is some μ_4 such that $\mu_1 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_4$ (Lemma 5.1 in [65]). Thus, the hierarchy for full-lossy cEOSs collapses. Note that $\leqslant_f$ is not compatible over EOSs. This helps to prove the undecidability of reachability and coverability over them (cf. Th.4.3 in [65]). Since $(\leqslant_f, 0)$-coverability for cEOSs is decidable (Th. 5.2 in [65]), we obtain the following theorem.

Theorem 1. *For $\ell \geqslant 1$, $(\leqslant_f, \ell)$-reachability and $(\leqslant_f, \ell)$-coverability for cEOSs are decidable. [36]*

Instead $\leqslant_s$ is compatible for both EOSs and cEOSs, as shown next.

Lemma 6. *If (λ, ρ) enables the event e on μ and $\mu' \geqslant_s \mu$, then (λ, ρ) enables the event e also on μ'. [36]*

Proof. If (λ, ρ) enables the event e on μ, then the enabledness formula $\Phi(e, \lambda, \rho)$ holds and $\lambda \leqslant_s \mu$. Since $\mu \leqslant_s \mu'$, by transitivity of $\leqslant_s$, we have that $\lambda \leqslant_s \mu'$. Thus, (λ, ρ) enables e on μ'. □

Lemma 7. *$\leqslant_s$ is strong compatible on EOSs. [36]*

Proof. If $\mu_1 \geqslant_s \mu_2 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_3$, then $\lambda \leqslant_s \mu_2$, $\mu_3 = \mu_2 - \lambda + \rho$, and there is some $\Delta(\mu_2)$ such that $\mu_1 = \mu_2 + \Delta(\mu_2)$. Moreover, since (λ, ρ) enables e on μ_2, then, by Lemma 6, (λ, ρ) enables e on μ_1. Thus, there is a μ_4 such that $\mu_1 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_4$. Moreover, by EOS semantics and the fact that $\lambda \leqslant_s \mu_2$, we have that $\mu_4 = \mu_1 - \lambda + \rho = \mu_2 + \Delta(\mu_2) - \lambda + \rho \geqslant_s \mu_2 - \lambda + \rho = \mu_3$. □

Theorem 2. *The hierarchies for system-lossy EOSs and system-lossy cEOSs collapse. [36]*

Thus, the study of system-lossiness on EOSs and cEOSs boils down to $(\leqslant_s, 0)$-coverability for EOSs and cEOSs (we study them in Th. 6 below). Finally, we show that $\leqslant_o$ is compatible on cEOSs. The following preliminary remarks can be easily proved.

Remark 1. $\leqslant_f = \leqslant_o \circ \leqslant_s = \leqslant_s \circ \leqslant_o$.

Remark 2. $\Pi^1(\mu_1) + \Pi^1(\mu_2) = \Pi^1(\mu_1 + \mu_2)$. If $\mu_1 \leqslant_o \mu_2$, then $\Pi^1(\mu_1) = \Pi^1(\mu_2)$.

Lemma 8. *$\leqslant_o$ is strong compatible on cEOSs. [36]*

Proof. The proof is depicted in Fig. 7.3. If $\mu_1 \geqslant_o \mu_2 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_3$, then, since $\leqslant_o \subseteq \leqslant_f$ and $\leqslant_f$ is strong compatible on cEOSs (Th. 5.1 in [65]), there is some μ_4 such that $\mu_1 \xrightarrow{\langle e,\lambda,\rho\rangle} \mu_4 \geqslant_f \mu_3$. Since $\leqslant_f = \leqslant_s \circ \leqslant_o$ (Rem. 1), there is also some μ_4' such that $\mu_3 \leqslant_s \mu_4' \leqslant_o \mu_4$. Thus, there is some $\Delta(\mu_3)$ such that $\mu_4' = \mu_3 + \Delta(\mu_3)$ and, by Rem. 2, we have $\Pi^1(\mu_4') = \Pi^1(\mu_4)$. Note that, by EOS semantics, μ_1 and $\mu_2 + \Delta(\mu_3)$ have respectively the predecessors $\mu_4 = \mu_1 - \lambda + \rho$ and $\mu_4' = \mu_2 + \Delta(\mu_3) - \lambda + \rho$. By some simple algebra,[5] $\Pi^1(\mu_1) = \Pi^1(\mu_2 + \Delta(\mu_3))$. However, again by Rem. 2, since $\mu_2 \leqslant_o \mu_1$, we have $\Pi^1(\mu_1) = \Pi^1(\mu_2)$ and, summarising, $\Pi^1(\mu_2) + \Pi^1(\Delta(\mu_3)) = \Pi^1(\mu_2 + \Delta(\mu_3)) = \Pi^1(\mu_1) = \Pi^1(\mu_2)$. Thus, $\Pi^1(\Delta(\mu_3)) = \varnothing$. Consequently, $\Delta(\mu_3) = \varnothing$ and $\mu_4 \geqslant_o \mu_4' = \mu_3 + \Delta(\mu_3) = \mu_3$.

□

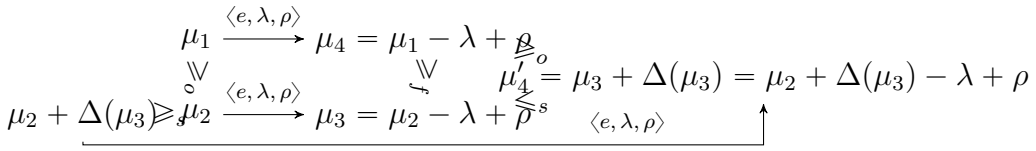


Figure 7.3: Depiction of proof of Th. 8.

Theorem 3. *The hierarchy for object-lossy cEOSs collapses. [36]*

Thus, the study of object-lossiness on cEOSs boils down to $(\leqslant_o, 0)$-coverability (we study this problem in Th. 5). Summarising, compatibility considerably simplifies the landscape of lossy problems for lossy cEOSs and for system-lossy EOSs. Specifically, for cEOSs, the only relevant problems are only the status of $(\leqslant_o, 0)$-coverability and of $(\leqslant_s, 0)$-coverability. Similarly, for system-lossy EOSs, the only relevant question is the status of $(\leqslant_s, 0)$-coverability.

7.5 Distinct Problems

Unfortunately, compatibility does not hold for $\leqslant_o$ and $\leqslant_f$ on EOSs. This is the main fact allowing the simulation of inhibitory nets via EOSs in [65]. Thus, the hierarchies induced by $\leqslant_o$ and $\leqslant_f$ do not collapse. In fact, we now show that all the problems in the hierarchy are distinct. We make use of a gadget with a dedicated place that counts the lossy steps.

Definition 9. The *lossiness-counter gadget* $\mathcal{G}$ is the EOS $(\hat{N}, \mathcal{N}, d, \Theta)$ depicted in Fig. 7.4 where

1. $\hat{N}$=$(\hat{P}, \hat{T}, \hat{F})$ where $\hat{P} = \{p_1, p_2, count\}$, $\hat{T} = \{\tau_1, \tau_2\}$, $\hat{F}(x) = 1$ if $x \in \{(p_1, \tau_1), (\tau_1, count), (\tau_1, p_2), (p_2, \tau_2), (\tau_2, count), (\tau_2, p_1)\}$ and $\hat{F}(x) = 0$ otherwise,
2. $\mathcal{N} = \{N_1, N_2\}$ where $N_1 = (\{p\}, \{inc_1\}, F_1)$, $F_1(\{(inc_1, p)\}) = 1$, $F_1(\{(p, inc_1)\})$, $N_2 = (\{q\}, \{inc_2\}, F_2\})$, $F_2(\{(inc_2, q)\}) = 1$, and $F_2(\{(q, inc_1 2)\})$,
3. $d(p_1) = N_1$, $d(p_2) = N_2$, and $d(count) = \blacksquare$, and
4. $\Theta = \{\langle \tau_1, inc_2\rangle, \langle \tau_2, inc_1\rangle\}$.

In what follows, we work with a fixed initial nested marking $\mu_0 = \langle p_1, \{\{p\}\}\rangle$ of $\mathcal{G}$.

[5] $\Pi^1(\mu_1) + \Pi^1(\lambda) + \Pi^1(\rho) = \Pi^1(\mu_1 - \lambda + \rho) = \Pi^1(\mu_4) = \Pi^1(\mu'_4) = \Pi^1(\mu_2 + \Delta(\mu_3) - \lambda + \rho) = \Pi^1(\mu_2 + \Delta(\mu_3)) - \Pi^1(\lambda) + \Pi^1(\rho)$.

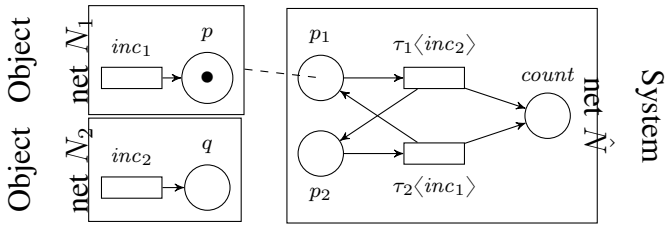


Figure 7.4: The lossiness-counter gadget $\mathcal{G}$ in Def. 9 (where $d(p_1) = N_1, d(p_2) = N_2, d(count) = \blacksquare$) with initial marking $\mu_0 = \{\{\langle p_1, \{\{p\}\}\rangle\}\}$.

7.5.1 Distinct Problems for Object-lossy EOSs

We first study the case of object-lossiness.

Lemma 9. *Let $\prec$ be a transitive relation. Given $\ell \in \mathbb{N} \cup \{\omega\}$, we have that μ is $(\preceq, \ell)$-reachable from μ_0 in $\mathcal{G}$ iff μ is $(\precnsim, \ell)$-reachable from μ_0. [36]*

Proof. Each $\precnsim$-lossy step $\mu_1 \succnsim \mu_2$ can be interpreted as a $\preceq$-lossy step $\mu_1 \succeq \mu_2$. Thus, each $(\precnsim, \ell)$-run can be interpreted as a $(\preceq, \ell)$-run. Similarly, each $(\preceq, \ell)$-run that does not contain *reflexive* lossy steps of the form $\mu \succeq \mu$ can be interpreted as a $(\precnsim, \ell)$-run. Moreover, each maximal finite or infinite sub-run $\mu \succeq \mu \succeq \ldots \mu \succeq \ldots$ of an arbitrary $(\preceq, \ell)$-run σ can be substituted by a single occurrence of μ, obtaining a $(\preceq, \ell)$-run without reflexive lossy steps. □

Lemma 10. *For each $\ell \geqslant 0$, the lossiness-counter gadget $\mathcal{G}$ exhibits a single maximal ℓ-strong $(\precnsim_o, \ell)$-run from its initial nested-marking μ_0. This run has the form*

$$\mu_0 \succnsim_o \mu'_0 \rightarrow \mu_1 \succnsim_o \mu'_1 \rightarrow \mu_2 \succnsim_o \ldots \mu_{\ell-1} \succnsim_o \mu'_{\ell-1} \rightarrow \mu_\ell$$

where, for each $i \leqslant \ell$, we have $\mu_i = \langle x_i, \{\{y_i\}\}\rangle + i\langle count, \varepsilon\rangle$ and $\mu'_i = \langle x_{i+1}, \varnothing\rangle + i\langle count, \varepsilon\rangle$ where, for each $j \in \mathbb{N}$, $x_j = p_1$ and $y_j = p$ if j is even and $x_j = p_2$ and $y_j = q$ if j is odd. [36]

Proof. By induction on ℓ. If $\ell = 0$, then, since μ_0 is a deadlock for $\rightarrow$, the run μ_0 of length 0 is the only $(\precnsim_o, \ell)$-run. Moreover, 0 is even and $\mu_0 = \langle p_1, \{\{p\}\}\rangle + 0\langle count, \varepsilon\rangle$.

If the inductive hypothesis holds for an arbitrary even ℓ, then $\mu_\ell = \langle p_1, \{\{p\}\}\rangle + \ell\langle count, \varepsilon\rangle$ is a deadlock for $\rightarrow$. The only way to continue the run is by one $\succnsim_o$ step. However, the only token that can be lost under $\precnsim_o$ is the token inside $\langle p_1, \{\{p\}\}\rangle$, thus, the only $\precnsim_o$-successor of μ_ℓ is $\mu'_\ell = \langle p_1, \varnothing\rangle + \ell\langle count, \varepsilon\rangle$. On μ'_ℓ there is no available $\succnsim_o$ step and the only enabled event is (τ_1, inc_2) with mode (λ, ρ) where $\lambda = \langle p_1, \varnothing\rangle$ and

$\rho = \langle p_2, \{\{q\}\}\rangle + \langle count, \varepsilon\rangle$. Its firing reaches $\mu_{\ell+1} = \langle p_2, \{\{q\}\}\rangle + (\ell+1)\langle count, \varepsilon\rangle$. This configuration is a deadlock for $\rightarrow$ and the so obtained $(\lesssim_o, \ell+1)$-run from μ_0 to $\mu_{\ell+1}$ already contains $\ell+1$ $\lesssim_o$-steps. Thus, this run is maximal among the $(\lesssim_o, \ell+1)$-runs.

If the inductive hypothesis holds for an arbitrary odd ℓ, the same argument applies with the provision that p_1 and p_2, p and q, as well as (τ_1, inc_2) and (τ_2, inc_1), have to be swapped. □

Corollary 3. *For each finite ℓ, the set of nested markings which are $(\leqslant_o, \ell)$-reachable from μ_0 is $\{\mu_i \mid i \in \{0,\ldots,\ell\}\} \cup \{\mu'_i \mid i \in \{0,\ldots,\ell-1\}\}$ where μ_i and μ'_i are defined as in Lemma 10. [36]*

Consequently, using the same notation as in Lem. 10, for each $\ell \in \mathbb{N}$, we have that μ'_ℓ is $(\leqslant_o, \ell)$-coverable but not $(\leqslant_o, \ell)$-reachable and $\mu_{\ell+1}$ is $(\leqslant_o, \ell+1)$-reachable but not $(\leqslant_o, \ell)$-coverable. Thus, the sequence of yes-instance sets of $(\leqslant_o, \ell)$-reachability/coverability problems (recall Lem. 2 and Lem. 3) is a sequence of proper subsets. Consequently, for each finite ℓ, all $(\leqslant_o, \ell)$-reachability/coverability problems are pairwise distinct. Moreover, while $(\leqslant_o, \omega)$-reachability coincides with $(\leqslant_o, \omega)$-coverability by Lem. 5, also $(\leqslant_o, \omega)$-reachability is distinct from $(\leqslant_o, \ell)$-reachability/coverability for each ℓ.[6]

Corollary 4. *For each $\ell_1 < \ell_2 \leqslant \omega$, we have that $(\leqslant_o, \ell_1)$-reachability, $(\leqslant_o, \ell_1)$-coverability, $(\leqslant_o, \ell_2)$-reachability, and $(\leqslant_o, \ell_2)$-coverability are pair-wise distinct problems. [36]*

7.5.2 Distinct Problems for Full-lossy EOSs

A fact analogous to Cor. 4 applies also to $\leqslant_f$. In fact, even if $\mathcal{G}$ exhibits more complex maximal $(\lesssim_f, \ell)$-runs, we still need ℓ lossy steps to put ℓ tokens on $count$. We first show that if a marking is reachable/coverable using $\leqslant_f$-lossy steps, then it is also covered under $\leqslant_s$ by some marking reachable via a run as in Lem. 10.

Lemma 11. *If $\sigma \in Runs_\ell(\leqslant_f, \mu_0)$ and $\mu_0 \rightsquigarrow^\sigma \mu$, then there is some $\lambda \geqslant_s \mu$ and there is an ℓ'-strong run $\sigma' \in Runs_{\ell'}(\lesssim_o, \mu_0)$ such that $\mu_0 \rightsquigarrow^{\sigma'} \lambda$, for some $\ell' \leqslant \ell$. [36]*

Proof. Since $\leqslant_f = \leqslant_o \circ \leqslant_s$, we can expand each $\leqslant_f$-lossy step in two subsequent $\leqslant_o$- and $\leqslant_s$-lossy steps. By compatibility of $\leqslant_s$ on EOSs (Lem. 8) and the fact that $\leqslant_s \circ \leqslant_o = \leqslant_o \circ \leqslant_s$ (Rem. 1), we can push all the $\leqslant_s$-lossy steps at the end of the run, obtaining a run $\sigma' \in Runs_\ell(\leqslant_o, \mu_0)$ such that $\mu_0 \rightsquigarrow^{\sigma'} \lambda \geqslant_s \cdots \geqslant_s \mu$, for some marking λ. By transitivity of $\leqslant_s$, we have

[6] Otherwise, it would coincide also with $(\leqslant_o, \ell+1)$-reachability/coverability. Thus, $(\leqslant_o, \ell)$-reachability/coverability and $(\leqslant_o, \ell+1)$-reachability/coverability would coincide, which is a contradiction.

$\lambda \geqslant_s \mu$. Moreover, we can drop the $\leqslant_o$-lossy steps in σ' of the form $\pi \geqslant_o \pi$ for some π, obtaining an ℓ'-strong run $\sigma'' \in Runs_{\ell'}(\lesssim_o, \mu_0)$ for some $\ell' \leqslant \ell$. □

We now show that also the sequence of yes-instance sets of $\leqslant_f$-lossy reachability/coverability problems (recall Lem. 2 and Lem. 3) is a sequence of proper subsets. We do that by using the same markings μ_ℓ and μ'_ℓ defined in Lem. 10.

Lemma 12. *Using the notation in Lem. 10, for each finite $\ell \in \mathbb{N}$, we have that $\mu_{\ell+1}$ is $(\leqslant_f, \ell+1)$-reachable, but not $(\leqslant_f, \ell)$-coverable. [36]*

Proof. By Lem. 10 we know that $\mu_{\ell+1}$ is $(\leqslant_o, \ell+1)$-reachable and, thus, also $(\leqslant_f, \ell+1)$-reachable. We now show that $\mu_{\ell+1}$ is not $(\leqslant_f, \ell)$-coverable. Assume by contradiction that $\mu_{\ell+1}$ is $(\leqslant_f, \ell)$-coverable. Then, there is some run $\sigma \in Runs_\ell(\leqslant_f, \mu_0)$ and a nested marking λ such that $\mu_0 \rightsquigarrow^\sigma \lambda \geqslant_f \mu_{\ell+1}$. Since $\mu_{\ell+1} \geqslant_s (\ell+1)\langle count, \varepsilon\rangle$, also $\lambda \geqslant_f (\ell+1)\langle count, \varepsilon\rangle$. Thus, $\lambda \geqslant_s (\ell+1)\langle count, \varepsilon\rangle$.[7] By applying Lem. 11 on the run σ, there is some nested marking $\lambda' \geqslant_s \lambda$ and, for some $\ell' \leqslant \ell$, a ℓ'-strong run $\sigma' \in Runs_{\ell'}(\lesssim_o, \mu_0)$ such that $\mu_0 \rightsquigarrow^{\sigma'} \lambda'$. Thus, by Lem. 10, we have that either $\lambda' = \mu_{\ell'}$ or $\lambda' = \mu'_{\ell'-1}$ and, hence, λ' places at most ℓ' black tokens on $count$. However, $\lambda' \geqslant_s \lambda \geqslant_s \ell+1\langle count, \varepsilon\rangle$, i.e., λ' puts at least $\ell+1$ tokens on $count$ even if $\ell' \leqslant \ell < \ell+1$, which is a contradiction. □

Lemma 13. *Using the notation in Lem. 10, for each finite $\ell \in \mathbb{N}$, we have that $\mu'_{\ell+1}$ is $(\leqslant_f, \ell)$-coverable, but not $(\leqslant_f, \ell)$-reachable. [36]*

Proof. By Lem. 10 we know that $\mu'_{\ell+1}$ is $(\leqslant_o, \ell)$-coverable and, thus, also $(\leqslant_f, \ell)$-coverable. We now show that $\mu'_{\ell+1}$ is not $(\leqslant_f, \ell)$-reachable. Assume by contradiction that $\mu'_{\ell+1}$ is $(\leqslant_f, \ell)$-reachable. Then, there is some run $\sigma \in Runs_\ell(\leqslant_f, \mu_0)$ such that $\mu_0 \rightsquigarrow^\sigma \mu'_{\ell+1}$. By applying Lem. 11 on the run σ, there is some nested marking $\lambda \geqslant_s \mu'_{\ell+1}$ and, for some $\ell' \leqslant \ell$, a ℓ'-strong run $\sigma' \in Runs_{\ell'}(\lesssim_o, \mu_0)$ such that $\mu_0 \rightsquigarrow^{\sigma'} \lambda$. Since $\lambda \geqslant_s \mu'_{\ell+1} \geqslant_s (\ell+1)\langle count, \varepsilon\rangle$, we have that λ puts at least $\ell+1$ tokens on $count$. However, by Lem. 10, we have that either $\lambda = \mu_{\ell'}$ or $\lambda = \mu'_{\ell'-1}$. Hence, λ puts at most $\ell' \leqslant \ell < \ell+1$ tokens on $count$, which is a contradiction. □

Corollary 5. *For each $\ell_1 < \ell_2 \leqslant \omega$, we have that $(\leqslant_f, \ell_1)$-reachability, $(\leqslant_f, \ell_1)$-coverability, $(\leqslant_f, \ell_2)$-reachability, and $(\leqslant_f, \ell_2)$-coverability are pair-wise distinct problems. [36]*

[7] In fact, $\lambda \geqslant_f (\ell+1)\langle count, \varepsilon\rangle$ implies that there is some π such that $\lambda \geqslant_s \pi \geqslant_o (\ell+1)\langle count, \varepsilon\rangle$; since the latter marking only places black tokens at the system net level, π is obtained by adding zero tokens at the object net level, i.e., $\pi = (\ell+1)\langle count, \varepsilon\rangle$. Hence $\lambda \geqslant_s (\ell+1)\langle count, \varepsilon\rangle$.

7.6 Undecidability for object- and system-lossy cEOSs

We now study the decidability status of $(\leqslant_o, 0)$-coverability and $(\leqslant_s, 0)$-coverability for cEOSs. In [64], reachability for cEOSs is proved undecidable via a reduction from reachability of 2CMs. We provide a variant of that reduction that reduces reachability of 2CMs to $\leqslant_o$-coverability and to $\leqslant_s$-coverability of cEOSs. This proves that both coverability problems are undecidable. By Th. 3 and Th. 2, all ℓ-reachability problems for object-lossy and system-lossy cEOSs are, thus, undecidable. Since cEOSs are a special case of EOSs, undecidability applies also to object-lossy and system-lossy EOSs.

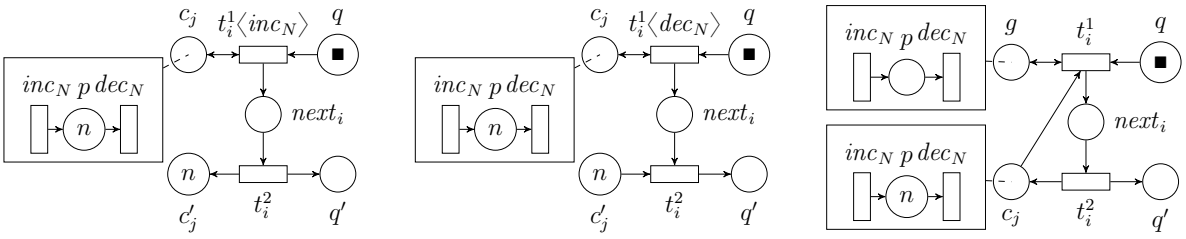
7.6.1 Reduction to Reachability

We show a reduction from reachability of a target configuration (q_f, n_1, n_2) of any 2CM $\mathcal{K} = (Q, \delta, q_0)$ with increment, decrement, and zero-check instructions to reachability of a cEOS $\mathbf{E}_\mathcal{K}$.

Definition 10. Given a 2CM $\mathcal{K} = (Q, \delta, q_0)$, we define the EOS $\mathbf{E}_\mathcal{K} = (\hat{N}, \mathcal{N}, d, \Theta)$ where

1. $\hat{N} = (\hat{P}, \hat{T}, \hat{F})$ is such that:
   - $\hat{P}$ contains Q, a place g, a place next_i for each instruction $i \in \delta$, and two places c_j and c'_j for each counter cnt_j;
   - $\hat{T}$ contains two transitions t_i^1 and t_i^2 for each instruction $i \in \delta$;
   - For each increment (resp., decrement, zero-check) instruction $i \in \delta$, $\hat{F}$ captures the pre- and post-conditions depicted in Fig. 7.5a (Fig. 7.5b, Fig. 7.5c).
2. $\mathcal{N}$ contains the net $\blacksquare$ and a single net $N = (P, T, F)$ where $P = \{p\}$, $T = \{inc_N, dec_N\}$, $F(inc_N, p) = F(p, dec_N) = 1$, and $F(p, inc_N) = F(dec_N, p) = 0$.
3. $d(c_1) = d(c_2) = d(g) = N$ and $d(x) = \blacksquare$ for each other place $x \in \hat{P} \backslash \{c_1, c_2, g\}$.
4. The synchronization structure Θ contains the events
   - $e_i^1 = (t_i^1, \{\{inc_N\}\})$ and $e_i^2 = (t_i^2, \varnothing)$ for each increment instruction i.
   - $e_i^1 = (t_i^1, \{\{dec_N\}\})$ and $e_i^2 = (t_i^2, \varnothing)$ for each decrement instruction i.
   - $e_i^1 = (t_i^1, \varnothing)$ and $e_i^2 = (t_i^2, \varnothing)$ for each zero-check instruction i.

Note that $\mathbf{E}_\mathcal{K}$ is a cEOS. It weakly simulates the increment and decrement instructions of $\mathcal{K}$ and performs zero-guesses in place of zero-check instructions. These guesses may be wrong but leave behind them two irreversible evidences: tokens in the internal marking of the object net at g and non-matching numbers of tokens on the place c_j and in the object net on c'_j, for



(a) (b) (c)

Figure 7.5: Part of $\mathbf{E}_{\mathcal{K}}$ capturing an (a) increment, (b) decrement, or (c) zero-check instruction $i \in \delta$.

some $j \in \{0, 1\}$. The former can be detected by $(\leqslant_s, 0)$-coverability; the latter can be detected by $(\leqslant_o, 0)$-coverability. We make these notions precise.

Definition 11. We say that a nested marking μ for $\mathbf{E}_{\mathcal{K}}$ is *legal* if it places exactly one object-token N at c_0, c_1, and g, exactly one ■ on exactly one place $q \in Q$, and no token on any place in $\hat{P}\backslash\{c'_0, c'_1, q\}$. If μ is legal, we denote

1. by $[\mu]'_j$ the number of black tokens placed at c'_j,
2. by $[\mu]_j$ the number of black-tokens in the object-token at c_j,
3. by $[\mu]_g$ the number of black-tokens in the object-token at g, and
4. by $[\mu]_Q$ the place $q \in Q$ marked by μ with a black-token.

Moreover, we say that μ:

1. is *broken at (counter)* j if $[\mu]_j \neq [\mu]'_j$, *sub-broken at* j if $[\mu]_j < [\mu]'_j$, *broken at* g if $[\mu]_g \neq 0$,
2. is *broken* if it is broken at 0, at 1, or at g,
3. *encodes* the 2CM configuration $c = (q, n_0, n_1)$, denoted by $\mu = \langle c\rangle$, if μ is non-broken and c=$([\mu]_Q, [\mu]_0, [\mu]_1)$.

The following lemma is a direct consequence of the pre- and post-conditions of the transitions in $\mathbf{E}_{\mathcal{K}}$, the shape of its synchronization structure Θ, the EOS semantics, and Def. 10. Its proof consists in simple, yet space-consuming algebraic checks and, thus, it is omitted.

Lemma 14. *Let* $i \in \{(q, +, j, q'), (q, -, j, q'), (q, =, j, q')\} \cap \delta$ *and* μ *a legal nested marking for* $\mathbf{E}_{\mathcal{K}}$. *If* e_i^1 *is enabled on* μ, *there is some* μ' *and* μ'' *such that* $\mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$ *and*

1. $[\mu]_Q = q$, μ'' *is legal* , $[\mu'']_Q = q'$, $[\mu'']_{1-j} = [\mu]_{1-j}$, *and* $[\mu'']'_{1-j} = [\mu]'_{1-j}$.
2. *if* $i = (q, +, j, q')$, *then* $[\mu'']_j = [\mu]_j + 1$, $[\mu'']'_j = [\mu]'_j + 1$ *and* $[\mu'']_g = [\mu]_g$.

3. *if* $i = (q, -, j, q')$, *then* $[\mu'']_j = [\mu]_j - 1$, $[\mu'']'_j = [\mu]'_j - 1$ *and* $[\mu'']_g = [\mu]_g$.
4. *if* $i = (q, =, j, q')$, *then* $[\mu'']_j = 0$, $[\mu'']'_j = [\mu]'_j$, *and* $[\mu'']_g = [\mu]_g + [\mu]_j$. *[36]*

Corollary 6. *Let* $\mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$ *and* μ *legal. If* μ *is sub-broken at* 0 *or at* 1, *then* μ'' *is sub-broken at* 0 *or at* 1, *respectively. If* μ *is broken at* g, *then* μ'' *is broken at* g. *[36]*

Corollary 7. *Let* $\mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$, *and* μ *legal and non-broken at* 0 *or at* 1. *If* μ'' *is broken at* 0 *or at* 1, *respectively, then* μ'' *is sub-broken at* 0 *or at* 1. *[36]*

Corollary 8. *Let* $\mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$, μ *legal, and* $i \in \{(q, +, j, q'), (q, -, j, q')\} \cap \delta$. *If* μ *is non-broken, then* μ'' *is non-broken. [36]*

Cor. 8 indicates that the simulation of increment or decrement instructions cannot lead, on their own, to broken markings. We now show that this is not the case for some simulation of zero-checks, called *wrong zero-guesses*.

Definition 12. A *wrong zero-guess* on counter j is a run $\mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$ where i is a zero-check instruction on counter j, μ is legal, and $[\mu]_j > 0$.

Lemma 15. *If* $\sigma : \mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$ *and* μ *is legal and non-broken, then* μ'' *is broken at* j *for some* $j \in \{0, 1\}$ *iff* σ *is a wrong zero-guess on counter* j. *[36]*

Proof. By Cor. 8, since μ is not broken at j but μ'' is, i is a zero-check instruction on counter j. If $[\mu]_j = [\mu]'_j = 0$, then $[\mu'']_j = 0 = [\mu]'_j = [\mu'']'_j$ and μ'' is not broken at j, contradiction. Thus, $[\mu]_j = [\mu]'_j > 0$ and, thus, σ is a wrong zero-guess. Vice-versa, if σ is a wrong-zero guess on counter j, then, by Lemma 14, μ'' is broken at j. □

The next lemma is proved analogously.

Lemma 16. *If* $\sigma : \mu \xrightarrow{e_i^1} \mu' \xrightarrow{e_i^2} \mu''$ *and* μ *is legal and non-broken, then* μ *is broken at* g *iff* σ *is a wrong zero-guess. [36]*

Corollary 9. *If* $\sigma : \langle (q_0, 0, 0) \rangle \rightarrow^* \mu$ *is a run of even length, then the following are equivalent: (1)* μ *is sub-broken at* 0 *or* 1, *(2)* μ *is broken at* g, *(3)* μ *is broken, and (4)* σ *has a wrong zero-guess. [36]*

Clearly, for each run $c_0 \xrightarrow{i_0} c_1 \rightarrow i_1 \ldots$ in $\mathcal{K}$ there is a run $\langle c_0 \rangle \xrightarrow{e_{i_0}^1} \mu_0 \xrightarrow{e_{i_0}^2} \langle c_1 \rangle \xrightarrow{e_{i_1}^1} \mu_1 \xrightarrow{e_{i_1}^2} \ldots$. If the target configuration (q_f, n_0, n_1) is reachable from $(q_0, 0, 0)$ in $\mathcal{K}$, then $\mu_f = \langle (q_f, n_0, n_1) \rangle$ is reachable from $\mu_0 = \langle (q_0, 0, 0) \rangle$ in $\mathbf{E}_{\mathcal{K}}$.

Vice-versa, if μ_f is reachable from μ_0 in $\mathbf{E}_\mathcal{K}$ via a run σ, then, μ_f is legal, non-broken, $[\mu_f]_Q = q_f$, and σ has even length. Thus, σ does not have any wrong zero-guess and can be simulated by a corresponding run in $\mathcal{K}$ from $(q_0, 0, 0)$.

Theorem 4. *(q_f, n_0, n_1) is reachable from $(q_0, 0, 0)$ in $\mathcal{K}$ iff $\mu_f = \langle (q_f, n_0, n_1) \rangle$ is reachable from $\mu_0 = \langle (q_0, 0, 0) \rangle$ in $\mathbf{E}_\mathcal{K}$. [36]*

Note that the EOS in Def. 10 is a cEOS. Thus, since 2CM reachability is undecidable, our result confirms that reachability for cEOSs is undecidable. However, we can conclude more.

7.6.2 From Reachability to $(\leqslant_o, 0)$-coverability

We now show that our construction yields undecidability also for $(\leqslant_o, 0)$-coverability. This is based on the fact that the nested marking μ_f in Th. 4 is reachable if and only if it is $(\leqslant_o, 0)$-coverable. In fact, if $\mu_0 \to^* \mu \geqslant_o \mu_f$, then, for each $j \in \{0, 1\}$,

1. μ and μ_f mark in the same way all system net places of type ■.
2. $[\mu]_Q$ is well-defined, $[\mu]_Q = [\mu_f]_Q = q_f$, μ is reachable only via runs of even length, and μ is legal.
3. for each $j \in \{0, 1\}$, $[\mu]'_j = [\mu_f]'_j = [\mu_f]_j \leqslant [\mu]_j$;
4. if μ is broken at j, then by Cor. 6 and Cor. 7 it is sub-broken at j and, thus, $[\mu]_j < [\mu]'_j \leqslant [\mu]_j$, contradiction; thus, μ is not broken at j and is not broken at 0 and, consequently, not broken at g.
5. $[\mu]_j = [\mu]'_j = [\mu_f]'_j = [\mu_f]_j$ and $[\mu]_g = 0 = [\mu_f]_g$.

Summarising μ and μ_f coincide. Thus, μ_f is $(\leqslant_o, 0)$-coverable from μ_0 iff it is reachable. By Th. 4, we obtain the following result.

Theorem 5. *$(\leqslant_o, 0)$-coverability for cEOSs is undecidable. [36]*

By Th. 3, by undecidability of reachability for cEOSs (Th.5.5 in [65]), and by the fact that each cEOS is also an EOS, we obtain the following result.

Corollary 10. *For each $\ell \in \mathbb{N} \cup \{\omega\}$, we have that $(\leqslant_o, \ell)$-reachability and $(\leqslant_o, \ell)$-coverability for cEOSs and for EOSs are undecidable. [36]*

7.6.3 From Reachability to $(\leqslant_s, 0)$-coverability

We now show that similar statements apply also for $(\leqslant_s, 0)$-coverability. In fact, if $\mu_0 \to^* \mu \geqslant_s \mu_f$, then, for each $j \in \{0, 1\}$,

1. μ places at least one black-token on $[\mu_f]_Q = q_f$, thus μ is reachable only via runs of even length, and μ is legal.
2. possibly with the exception of c'_0 and c'_1, μ and μ_f coincide on all system net places, including all object-tokens on them.
3. $[\mu]_j = [\mu_f]_j = [\mu_f]'_j \leqslant [\mu]'_j$ and $[\mu]_g = [\mu_f]_g$, thus μ is not broken and $[\mu]'_j = [\mu]_j = [\mu_f]_j$.

Summarising μ and μ_f coincide. Thus, μ_f is $(\leqslant_s, 0)$-coverable from μ_0 iff it is reachable. By Th. 4, we obtain the following result.

Theorem 6. *$(\leqslant_s, 0)$-coverability for cEOSs is undecidable. [36]*

By Th. 2, by undecidability of reachability for cEOSs (Th.5.5 in [65]), and the fact that each cEOS is also an EOS, we obtain the following corollaries.

Corollary 11. *For each $\ell \in \mathbb{N} \cup \{\omega\}$, we have that $(\leqslant_s, \ell)$-reachability and $(\leqslant_s, \ell)$-coverability for cEOSs and for EOSs are undecidable. [36]*

7.7 Undecidability for full-lossy EOSs

We now show that all $(\leqslant_f, \ell)$-reachability problems for EOSs with finite $\ell \geqslant 1$ are undecidable. This is achieved via a reduction of reachability of 2CMs to $(\leqslant_f, \ell)$-reachability with any given ℓ.

Fix the value of $\ell \geqslant 1$. Given an arbitrary 2CM $\mathcal{K} = (Q, \delta, q_0)$, a target configuration (q_f, n_1, n_2) of $\mathcal{K}$, and its simulating EOS $\mathbf{E}_\mathcal{K} = (\hat{N}_\mathcal{K}, \mathcal{N}_\mathcal{K}, d_\mathcal{K}, \Theta_\mathcal{K})$ as in Sec. 7.6, we merge $\mathbf{E}_\mathcal{K}$ with the lossiness-counter gadget $\mathcal{G} = (\hat{N}_\mathcal{G}, \mathcal{N}_\mathcal{G}, d_\mathcal{G}, \Theta_\mathcal{G})$ from Sec. 7.4. Specifically, for $\hat{N}_\mathcal{K} = (\hat{P}_\mathcal{K}, \hat{T}_\mathcal{K}, \hat{F}_\mathcal{K})$ and $\hat{N}_\mathcal{G} = (\hat{P}_\mathcal{G}, \hat{T}_\mathcal{G}, \hat{F}_\mathcal{G})$, we start with the EOS $\mathcal{M} = (\hat{M}, \mathcal{N}_\mathcal{K} \cup \mathcal{N}_\mathcal{G}, d_\mathcal{K} \cup d_\mathcal{G}, \Theta_\mathcal{K} \cup \Theta_\mathcal{G})$ where $\hat{M} = (\hat{P}_\mathcal{K} \cup \hat{P}_\mathcal{G}, \hat{T}_\mathcal{K} \cup \hat{T}_\mathcal{G}, \hat{F}_\mathcal{K} \cup \hat{F}_\mathcal{G})$.

We now add to $\mathcal{M}$ a system net transition *enabling* together with a dedicated event $e = \langle enabling, \varnothing \rangle$ (see Fig. 7.6). This transition consumes *(1)* ℓ tokens from *count*, *(2)* one from p_1 if ℓ is even, *(3)* one from p_2 if ℓ is odd. The transition *enabling* produces *(1)* one in q_0,

and *(2)* one object-token with empty internal marking in each of g, c_0, and c_1. We call the so obtained EOS $\mathcal{F}$. The initial marking μ_0 of $\mathcal{F}$ is $\langle p_1, \{\{p\}\}\rangle$, as for $\mathcal{G}$.

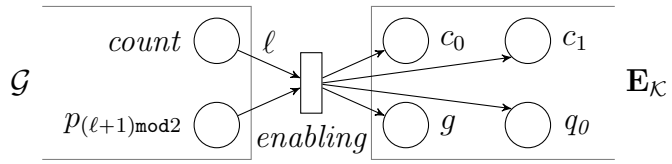


Figure 7.6: The places, transitions, and conditions we add on top of $\mathcal{M}$.

Note that, along the runs of $\mathcal{F}$, the event $e = \langle enabling, \varnothing\rangle$ fires at most once. Before firing e, there is no token on $\mathbf{E}_{\mathcal{K}}$ while, after firing e, there is no token on $\mathcal{G}$. Let σ be a $(\leqslant_f, \ell)$-run of $\mathcal{F}$. If e is not fired along σ, then σ does not reach the target marking $\mu_f = \langle(q_f, n_1, n_2)\rangle$ (which puts some token on $\mathbf{E}_{\mathcal{K}}$). Otherwise, σ can be split into two runs σ_1 and σ_2, such that $\sigma = \sigma_1 \xrightarrow{e} \sigma_2$. Since e consumes ℓ tokens from $count$ and one from either p_1 or p_2 in $\mathcal{G}$, the last marking μ_1 in σ_1 has to put at least ℓ tokens on $count$ and at least one token on either p_1 or p_2, respectively.

If σ_1 is not ℓ-strong, then it is ℓ'-strong for some $\ell' < \ell$. By Lem. 11, there is some ℓ''-strong run $\sigma \in Runs_{\ell''}(\lesssim_o, \mu_0)$ for some $\ell'' \leqslant \ell'$ such that $\mu_0 \rightsquigarrow^{\sigma_1} \lambda \geqslant_s \mu_1$ for some marking λ. Thus, by Lem. 10, λ puts on $count$ at most $\ell'' < \ell$ tokens. Since $\lambda \geqslant_s \mu_1$, so does μ_1, which is a contradiction. Thus, σ_1 is ℓ-strong.

Consequently, since σ is a $(\leqslant_f, \ell)$-run, σ_2 must be a (perfect) $(\leqslant_f, 0)$-run of $\mathbf{E}_{\mathcal{K}}$. Also, because of the post-conditions of $enabling$, the first marking in σ_2 is $\langle(q_0, 0, 0)\rangle$. Thus, the 2CM $\mathcal{K}$ reaches the target (q_f, n_1, n_2) from $(q_0, 0, 0)$ iff $\mathbf{E}_{\mathcal{K}}$ has a (perfect) $(\leqslant_f, 0)$-run from $\langle(q_0, 0, 0)\rangle$ to $\mu_f = \langle(q_f, n_1, n_2)\rangle$ iff $\mathcal{F}$ exhibits an ℓ-strong $(\leqslant_f, \ell)$-run from μ_0 to μ_f iff $\mathcal{F}$ exhibits $(\leqslant_f, \ell)$-run from μ_0 to μ_f. Since reachability of 2CM is undecidable, we get the next theorem.

Theorem 7. *For each finite $\ell \in \mathbb{N}$, $(\leqslant_f, \ell)$-reachability for EOSs is undecidable. [36]*

One can adapt this construction so as to concatenate the lossy-counter gadget $\mathcal{G}$ with any EOS $\mathbf{E}$ (in place of $\mathbf{E}_{\mathcal{K}}$ for any 2CM $\mathcal{K}$). If the initial marking of $\mathbf{E}$ contains several nested tokens, a chain of enabling events like e may be necessary to initialize it. As above, in order to fire them, the $(\leqslant_f, \ell)$-runs of the concatenated EOS $\mathcal{F}$ have to preliminary fire all their lossy steps. Moreover, after firing e, the intended initial marking of $\mathbf{E}$ is initialized and the $(\leqslant_f, \ell)$-runs of $\mathcal{F}$ can continue only by simulating $\mathbf{E}$ without any further lossy step. Thus,

$\mathbf{E}$ reaches/covers a target marking μ_f iff $\mathcal{F}$ $(\leqslant_f, \ell)$-reaches/covers the same target μ_f from $\mu_0 = \langle p_1, \{\{p\}\}\rangle$. Since (perfect) $(\leqslant_f, 0)$-reachability is known to be undecidable for EOSs (Th. 4.3 in [65]), one get again Th. 7. Moreover, also (perfect) $(\leqslant_f, 0)$-coverability is known to be undecidable (again, Th. 4.3 in [65]). Thus, we get undecidability also for $(\leqslant_f, \ell)$-coverability for each finite ℓ.

Theorem 8. *For each finite $\ell \in \mathbb{N}$, $(\leqslant_f, \ell)$-coverability for EOSs is undecidable. [36]*

7.8 Decidability of $(\leqslant_f, \omega)$-reachability for EOSs

We now show that $(\leqslant_f, \omega)$-reachability is decidable for EOSs. We use a modified semantics for full-lossy EOSs which merges standard and lossy steps.

Definition 13. A *merged-EOS* (mEOS) is an EOS interpreted under the semantics induced by the step relation $\rightsquigarrow$ where $\rightsquigarrow = \geqslant_f \cup \rightarrow$.

Since EOSs and mEOSs are syntactically the same, given an EOS $\mathbf{E}$, we denote by $\mathbf{E}_S$ the EOS $\mathbf{E}$ interpreted under the standard $\rightarrow$ step relation (S stands for standard), and by $\mathbf{E}_M$ the EOS $\mathbf{E}$ interpreted under the $\rightsquigarrow$ step relation (M stands for merged). Similarly, we denote the set of predecessors of μ in $\mathbf{E}_S$ by $Pred_S(\mu)$ and in $\mathbf{E}_M$ by $Pred_M(\mu)$. The benefit of mEOSs is that $\leqslant_f$ becomes trivially compatible. This solves the major source of undecidability behind the undecidability result of $\leqslant_f$-reachability for EOSs.

Lemma 17. *$\leqslant_f$ is compatible for mEOSs. [36]*

Proof. By definition of $\rightsquigarrow$, if $\mu_1 \geqslant_f \mu_2 \rightsquigarrow \mu_3$, then either *(1)* $\mu_1 \geqslant_f \mu_2 \geqslant_f \mu_3$ and, by transitivity of $\geqslant_f$, also $\mu_1 \rightsquigarrow^* \mu_1 \geqslant_f \mu_3$, or *(2)* $\mu_1 \geqslant_f \mu_2 \rightarrow \mu_3$ and, by definition of $\rightsquigarrow$ and reflexivity of $\leqslant_f$, also $\mu_1 \rightsquigarrow \mu_3 \geqslant_f \mu_3$. □

Consequently, since $\leqslant_f$ is a well-quasi order (see Th.5.2 in [65]), mEOSs with $\leqslant_f$ are well-structured transition systems (WSTS; see [49]). Clearly, $\leqslant_f$ is decidable. Moreover, mEOSs have the effective pred-basis property. This is because, $\uparrow Pred_M(\uparrow \mu) = \uparrow \{\mu\} \cup \uparrow Pred_S(\uparrow \mu)$ and $\mathbf{E}_S$ has the effective pred-basis property [65], where $\uparrow X$ denotes the upward-closure of X [8]

[8] I.e., if X is a set of markings, $\uparrow X$ is the set of markings μ such that $\mu \geqslant_f x$ for some $x \in X$; also, $\uparrow \mu$ denotes $\uparrow \{\mu\}$.

Lemma 18. *For an EOS* $\mathbf{E}$ *and a nested marking* μ*, we have* $\uparrow Pred_M(\uparrow \mu) = \uparrow \mu \cup \uparrow Pred_S(\uparrow \mu)$*.* [36]

Proof. If $\mu_1 \in \uparrow Pred_M(\uparrow \mu)$, then there are μ' and μ'' such that $\mu_1 \geqslant_f \mu' \rightsquigarrow \mu'' \geqslant_f \mu$. If $\mu' \geqslant_f \mu''$, then, by transitivity of $\leqslant_f$, we have that $\mu_1 \in \uparrow \mu$. If $\mu' \rightarrow \mu''$, then $\mu' \in Pred_S(\uparrow \mu)$ and, hence, $\mu_1 \in \uparrow Pred_S(\uparrow \mu)$. Vice-versa, since $\rightarrow \subseteq \rightsquigarrow$, we have $Pred_S(\mu) \subseteq Pred_M(\mu)$ and, thus, $\uparrow Pred_S(\uparrow \mu) \subseteq \uparrow Pred_M(\uparrow \mu)$. Moreover, since $\mu' \geqslant_f \mu$ implies $\mu' \in Pred_M(\mu)$, we have $\uparrow \{\mu'\} \subseteq \uparrow Pred_M \uparrow (\mu)$. □

We can then apply the theory of WSTS and obtain decidability of coverability for mEOSs.

Lemma 19. $(\leqslant_f, 0)$*-coverability for mEOSs is decidable.* [36]

Since each $(\leqslant_f, \omega)$-run in $\mathbf{E}_S$ is a $(\leqslant_f, 0)$-run in $\mathbf{E}_M$ and vice-versa, we have that $(\leqslant_f, 0)$-coverability for $\mathbf{E}_M$ coincides with $(\leqslant_f, \omega)$-coverability for $\mathbf{E}_S$, which, in turn, coincides with $(\leqslant_f, \omega)$-reachability for $\mathbf{E}_S$ (Cor. 5). We thus obtain the following theorem.

Theorem 9. $(\leqslant_f, \omega)$*-reachability is decidable for EOSs.* [36]

Concerning the complexity of $(\leqslant_f, \omega)$-reachability for EOSs, this problem extends $(\leqslant_f, \omega)$-reachability for cEOSs, which is equivalent to $(\leqslant_f, 0)$-coverability for cEOSs. By noting that these can encode PN coverability, we obtain a lower bound for $(\leqslant_f, \omega)$-reachability for EOSs.

Theorem 10. $(\leqslant_f, \omega)$*-reachability is EXPSPACE-hard for EOSs.* [36]

We have completely charted the decidability status of all lossy-reachability problems for three lossiness relations: full-lossiness $\leqslant_f$, object-lossiness $\leqslant_o$, and system-lossiness $\leqslant_s$. The decidability landscape is summarized in Tab. 7.1. For cEOSs, proper lossy-reachability coincides with standard coverability under the respective lossiness quasi order. All problems for object- and system-lossy EOSs and cEOSs are undecidable. This is enabled by the fact that the orders $\leqslant_o$ and $\leqslant_s$ are not well-quasi orders (cf. [72]). For full-lossy EOSs, all $(\leqslant_f, \ell)$-reachability problems are undecidable even if they do not coincide with standard coverability under $\leqslant_f$. The most interesting result is the decidability of $(\leqslant_f, \omega)$-reachability for EOSs. This result follows from the fact that each quasi order $\preceq$ induces a WSTS when interpreted over $(\preceq, \omega)$-runs (cf. [76]). This problem is at least as hard as $\leqslant_f$-coverability for cEOSs, which, in

	Problem	$\leqslant_f$	$\leqslant_o$	$\leqslant_s$
cEOS	0-reach.	undec. (Th 5.5 [65])	undec. (Th 5.5 [65])	undec. (Th 5.5 [65])
	cover.	dec. (Th 5.2 [65])	undec. [2CM]	undec. [2CM]
	ℓ-reach./cover. for $\ell \in \mathbb{N}_0$	dec. [comp.]	undec. [comp.]	undec. [comp.]
	ω-reach./cover	dec. [comp.]	undec. [comp.]	undec. [comp.]
EOS	0-reach	undec. (Th 4.3 [65])	undec. (Th 4.3 [65])	undec. (Th 4.3 [65])
	cover.	undec. (Th 4.3 [65])	undec. [cEOS]	undec. [cEOS]
	ℓ-reach./cover. for $\ell \in \mathbb{N}_0$	undec. [$\mathcal{G}$]	undec. [cEOS]	undec. [comp.]
	ω-reach./cover	dec. [WSTS]	undec. [cEOS]	undec. [comp.]

Table 7.1: Decidability status of lossy problems for full-, object-, and system-lossy EOS and cEOS. $\mathbb{N}_0$ denotes $\mathbb{N}\backslash\{0\}$. References are put next to already known results. The labels next to our results indicate the techniques used to obtain them: *[comp.]* - compatibility; *[2CM]* - 2CM reachability; *[cEOS]* - generalization of cEOS results; *[$\mathcal{G}$]* - lossiness-counter gadget $\mathcal{G}$ merging; *[WSTS]* - WSTS theory.

turn, extends PN coverability. This yields an EXPSPACE lower-bound. The precise complexity of $(\leqslant_f, \omega)$-reachability for EOSs and, to the best of our knowledge, of $\leqslant_f$-coverability for cEOSs, remains uncharted. We aim to fill this gap in future works.

Decidability of $(\leqslant_f, \omega)$-reachability enables, in principle, the analysis of EOS models, e.g., of business processes where resources may be lost both at the system and object level. However, where undecidability applies, we may still perform verification by employing partial procedures, e.g., by resorting to bounded model checking approaches. Recently, in [30], a Maude encoding of EOSs was proposed and reachability searches on a bounded EOS were performed. However, to the best of our knowledge, there is no tool that natively addresses (bounded) model checking of lossy EOSs. Such a tool should also be able to express formulas about the number and distribution of the lossy steps in the runs. This feature is reminiscent of program definitions in the temporal operators of Dynamic Propositional Logic (PDL) [15, 50]. Interestingly, recent Answer Set Programming tools [52], such as Telingo [28], support PDL constraints, which may be used to perform bounded model checking. A recent prototype [90] is discussed in [37]. The development of such a tool would enable us to practically verify the robustness of EOS models.

7.9 Tool to verify lossy Petri Nets and lossy EOSs

In this section, we discuss the prototype for the simulation and analysis of lossy-PNs and lossy-EOSs based on Answer Set Programming (ASP) technologies. Specifically, we aim at simulating and analyzing bounded lossy-EOS runs using Telingo [28], a specialization of the ASP system Clingo [51] to temporal domains. Implementations of PN variants in Clingo are [10] and [44]. However, none of them takes lossy PNs and EOSs in consideration and does not experiment with the temporal features of Telingo. In fact, the most appealing feature of Telingo is its support in constraints of Temporal Equilibrium Logic over finite traces (TEL_f) formulas, which we expect to be an excellent tool to flexibly control the amount of lossiness in the simulated runs.

7.9.1 Telingo

Telingo specializes the ASP solver Clingo to temporal domains and uses a logic program to specify finite runs. The program can use the scopes: *initial*,*dynamic*,*always*,*final*, which are evaluated at the first, each except first, each, and last step, respectively. Rule bodies can refer to the extension of the previous configuration by prefixing the literals with a prime. Finite-, linear-time formulas can appear in the dedicated atom `&tel` in constraints and behind default negation. For example, the constraint `:- &tel{>?(a>a)}` filters out all runs that eventually reach (`>?` stands for eventually reach) a configuration C with successor C' (`>` stands for next) where `a` is true on both. Telingo can be called setting the option `--imax` to a number of maximal step to be simulated. In the standard configuration, Telingo stops as soon as it finds a finite run satisfying the program or exceeds `--imax`.

7.10 PNs in Telingo

7.10.1 PNs in Logic Programs

Example 8. Fig. 7.7 depicts the PN $(\{p_0, p_1, p_2\}, \{t_1\}, F)$ where $F(p_0, t_1) = F(p1, t_1) = F(t_1, p_2) = 1$ and $F(t_1, p_0) = F(t_1, p_1) = F(p_2, t_1)$, with initial marking $\mu_0 = \{\{2p_0, 5p_1\}\}$. The net reaches $\mu_1 = \{\{3p_0, 2p_2\}\}$, covers $\mu_2 = \{\{\}\}$, reaches a deadlock after two steps, and is not 1-safe.

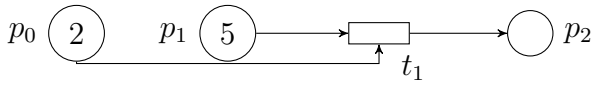


Figure 7.7: The PN with initial marking in Ex. 8.

The standard syntax for PNs is the PNML language [57]. For example, the input PNs provided by the MCC [69] are provided in PNML syntax. The first task we faced was the translation of PNML files into Logic Program (LP) files for Telingo. This required to fix a syntax to specify PNs in LP. Inspired by previous works (see, e.g., [10, 44]), we used the following solutions:

- The number n of places and m of transitions are specified by two constants via the directives `#const numPlaces=n` and `#const numPlaces=m`. The names of the places and transition is abstracted away, i.e., they range in $\{0, \ldots, n-1\}$ and $\{0, \ldots, m-1\}$ respectively.
- Each pre-condition $F(p,t) = n$ and post-condition $F(t,p) = n$ is explicitly specified only if $n > 0$ using the fact `pre(p,t,n)` and `post(p,t,n)`, respectively. These facts must be available during the whole computation and, thus, are put in the scope of the `#program always` directive.
- The file contains also the specification of the initial marking, which is represented as a function from places to numbers. Specifically, for each place p hosting $n \in \mathbb{N}$ tokens (possibly $n = 0$), we add the fact `mark(p,n)` to the scope of the `#program initial` directive. Finally, we clean the Telingo output using the directive `#show mark/2.`

Example 9. The LP specification of the PN in Ex. 8 is:

```
1 #program always.
2 #const numPlaces=3.
3 #const numTransitions=1.
4 pre(0,0,1).
5 pre(1,0,1).
6 post(2,0,1).
7 #program initial.
8 mark(0,2).
9 mark(1,5).
10 mark(2,0).
11 #show mark/2.
```

Using ANTLR [82], we built an open source tool [90], implemented in C++, to produce LP specifications out of MCC benchmarks.

7.10.2 PN dynamics

The PN dynamics can be easily implemented in Telingo using the standard guess-and-check methodology of ASP: at each step, we *1)* sample a transition using a choice rule, *2)* check

whether it is enabled on the previous marking using a constraint, and *3)* deduce the facts encoding the new marking using a couple of simple rules that take in consideration the sampled transition, its conditions, and the previous marking. This last step is done using the rules

```
1 mark(P,K-N+M) :- pre(P,T,N),
      post(P,T,M), 'mark(P,K), fire(T).
2 mark(P,K+M) :- not pre(P,T,_),
      post(P,T,M), 'mark(P,K), fire(T).
3 mark(P,K-N) :- pre(P,T,N), not
      post(P,T,_), 'mark(P,K), fire(T).
4 mark(P,K) :- not pre(P,T,_), not
      post(P,T,_), 'mark(P,K), fire(T).
```

The lossy dynamics is supported by allowing Telingo to possibly sample, next to the transitions at phase 1, also the `lossy` flag and, consequently, a sub-marking by firing the choice rule

```
1 {mark(P,1..N)}=1 :- 'mark(P,N),lossy.
```

Assuming that the dynamics is encoded in `dynamic.lp`, the simulation of the runs of maximum length n of a PN encoded in `pn.lp` can be executed as follows:

```
1 Telingo dynamic.lp pn.lp 0 --imax=n
```

7.10.3 PN Verification Problems

We considered several problems from the MCC. Since Telingo does not natively support branching time formulas, we focused on linear paths namely reachability/coverability, deadlock detection, and 1-safeness. Since Telingo simulates runs incrementally, it is sufficient to check these properties just at the last step of the finite run. In fact, reachability/coverability and deadlock detection are all eventuality properties. Moreover, also the opposite of 1-safeness, i.e., whether a non-1-safe marking can be reached, is of the same type.

Example 10. To check the reachability in Ex. 8 of the marking $(1, 0, 2)$, we need the rules

```
1 #program final.
2 :- not mark(0,1).
3 :- not mark(1,0).
4 :- not mark(2,2).
```

Coverability can be similarly be specified using the rules

```
1 #program final.
2 :- mark(1,N), N<1.
3 :- mark(1,N), N<0.
4 :- mark(2,N), N<2.
```

Deadlock reachability requires to check the enabledness of all transitions and, so, requires their enumeration in a dedicated predicate.

```
#program final.
transition(0..numTransitions-1).
disabled(T) :- transition(T),
    pre(P,T,N), mark(P,M), M<N.
enabled(T) :- not disabled(T),
    transition(T).
nonDeadlock :- enabled(T).
deadlock :- not nonDeadlock.
:- not deadlock.
```

Finally, (the opposite of) 1-safeness is specified by

```
#program final.
unsafe :- mark(P,N), N>1.
:- not unsafe.
```

By adding these rules, if no maximum number m of steps is signaled, Telingo will return a finite run witnessing the property, if it exists, or will never terminate, otherwise. If m is provided, Telingo will always terminate, but will return a witnessing run of at most m steps.

For all these properties, we can seamlessly restrict the analysis to runs with at most $\ell \in \mathbb{N} \cup \{|\mathbb{N}|\}$ many lossy-steps for any $\ell \in \mathbb{N}$. For example, the rules

```
#program initial.
:- &tel{>?(lossy >(>? lossy))}.
```

filter out all runs with at least two distinct lossy steps, so as to output only solutions witnessing the existence of suitable runs with at most $\ell = 1$ lossy step. The parameter ℓ can be controlled by nesting the string `>(>? lossy)` appropriately.

7.10.4 PN experiments

In the running example above, we used a simple PN to illustrate the concepts. However, for our experimentation, we considered several standard benchmarks taken from MCC [69]

which range across various industrial case studies and have different sizes of the PNs. We experimented with the analysis of deadlock reachability and 1-safeness, for various maximal numbers of steps (5, 10, and 20): these properties can be expressed without any expert knowledge on the PN structure. We compared the output of our prototype for all benchmarks with the output provided, on the same instances, by the state-of-the-art tool TAPAAL 3.9.3 [61]. The outputs match exactly; an indication of the correctness of the results, thereby giving our prototype a tool confidence of 100% (despite noncompetitive times). We analyzed the benchmarks for both runs without lossy steps and with arbitrarily many lossy steps. A subset of our results and comparisons (for no loss) on a single PN is reported in Tab. 7.2.

problem	lossiness	–imax =5 (s)	–imax =10 (s)	–imax =20 (s)	TAPAAL (s)
deadlock	none	UNSAT in 0.052	SAT in 0.622	SAT in 82.754	SAT in $5e-6$
deadlock	any	SAT in 0.009	SAT in 0.010	SAT in 0.009	NA
1-safeness	none	UNSAT in 0.010	UNSAT in 0.014	UNSAT in 0.027	UNSAT in 0
1-safeness	any	UNSAT in 0.013	UNSAT in 0.018	UNSAT in 0.032	NA

Table 7.2: Comparative Results with TAPAAL for the Eratosthenes-PT-010 PN from the MCC benchmarks [69].

Chapter 8

Conclusions

We summarize the work done in this dissertation:

- In Chapter. 2 we discussed the various models for unbounded concurency. First, we discussed Petri nets, the most widely studied model, where the tokens are undistinguishable. Second, we studied ν-nets, where the tokens can be distinguished using identifiers and there can be labeled arcs to enable the movement of particular tokens, which allows for a richer representation than Petri nets. We saw how ν-nets can be used to represent identifiable clients in a system with single server and unbounded number of clients. Third, we explored a particular class of higher order nets, where Petri nets can be nested. When the nesting depth is restricted to two, we obtain Elementary Object Systems, which are natural for representing clients as a nested Petri net within the server Petri net. We explored the suitability of each of these models to represent the unbounded client server systems case study.

- In Chapter. 3 we discussed the various temporal logics that can be used to specify properties of the models discussed in Chapter. 2. We explored the expressibility of Linear Temporal Logic, Linear Temporal Logic with integer arithmetic LTL_{LIA} and First Order Logic with Monodic restriction FOTL_1. In each of these logics, we looked at their applicability in expressing properties of the above concurrent models in a natural manner.

- In Chapter. 4 we explored the motivation behind verification of unbounded client server systems and the research questions that we aim to answer in this thesis. We discussed the algorithm of bounded model checking and its extension, two dimensional bounded model checking (2D-BMC). 2D-BMC is particularly useful to verify models such as Petri nets

where there is unboundedness in the concurrency ad well as with respect to the temporal modality.

- In Chapter. 5, we implemented a tool to perform 2D-BMC on unbounded Petri nets with specifications written in LTL_{LIA}, a variant of Linear Temporal Logic. We described the design of the tool, the SMT encoding and the experiments where it competes with the state of the art Petri net verification tools on competition benchmarks. This is the first of its kind to support true concurrent semantics of Petri nets.

- In Chapter. 6, we implemented another tool to verify properties of ν-nets, which are Petri nets where the tokens are distinguished. We employed a variants of First Order Logic FOTL_1, to represent the specifications and we verified the specifications using 2D-BMC algorithm and made use of SMT solvers to verify them.

- In Chapter. 7, we charted the decidability status of reachability, coverability problems in Elementary Object Systems, which are a type of nested Petri nets, in the context of lossiness. We additionally built a tool using Answer Set Programming, to verify the reachability, safety and deadlock properties on lossy Petri nets and lossy EOSs.

Future Work

In this work, we took the approach of representing single server multiple client systems as a single component and represented them as a single ν-net. As part of future work, we shall explore verification of a system of ν-nets, with several underlying components.

We represented the single server multiple client setting using EOSs. The following are natural extensions - the single client multiple server, multiple client and multiple server settings. In a single client multiple server setting, we may represent client process as a token net whose behaviour changes dynamically in various places, corresponding to the server behaviour. It would be interesting to implement tools to verify programs following these paradigms, as they are widely in use in industry.

We introduced 2D-BMC, which is an extension of the standard bounded model checking algorithm. It would be interesting to explore the complexity and completeness criteria of this algorithm.

The model checking problem for Linear Temporal Logic with integer arithmetic is undecidable, it would be interesting to see if decidability can be achieved in variants of this logic.

References

[1] Model Checking Contest 2022 Rules. https://mcc.lip6.fr/pdf/rules.pdf, September 2022.

[2] Results of Model Checking Contest. https://mcc.lip6.fr/results.php, September 2022.

[3] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. Unfoldings of unbounded Petri nets. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2000.

[4] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. SAT-solving the coverability problem for Petri nets. *Form. Methods Syst. Des.*, 24(1):25–43, January 2004.

[5] Parosh Aziz Abdulla and Bengt Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems. *Theor. Comput. Sci.*, 256(1-2):145–167, 2001.

[6] Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. Proving liveness by backwards reachability. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2006.

[7] C. Aiswarya. On network topologies and the decidability of reachability problem. In *International Conference on Networked Systems*, volume 12129 of *LNCS*, pages 3–10, 2020.

[8] Nicolas Amat, Bernard Berthomieu, and Silvano Dal-Zilio. On the combination of polyhedral abstraction and smt-based model checking for Petri nets. In *PETRI NETS*, LNCS, pages 164–185. Springer, 2021.

[9] Nicolas Amat, Silvano Dal-Zilio, and Thomas Hujsa. Property directed reachability for generalized Petri nets. In *Tools and Algorithms for the Construction and Analysis of Sys-*

tems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, pages 505–523. Springer, 2022.

[10] Saadat Anwar et al. Encoding higher level extensions of Petri nets in answer set programming. In *LPNMR*, volume 8148 of *LNCS*, pages 116–121, 2013.

[11] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[12] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*, volume 2529 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2002.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[14] A. R. Balasubramanian, Javier Esparza, and Marijana Lazic. Complexity of verification and synthesis of threshold automata. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2020.

[15] Philippe Balbiani and Emiliano Lorini. Ockhamist propositional dynamic logic: A natural link between PDL and CTL. In *Logic, Language, Information, and Computation Proceedings*, volume 8071 of *LNCS*, pages 251–265, 2013.

[16] Matteo Baldoni, Cristina Baroglio, and Roberto Micalizio. Fragility and robustness in multiagent systems. In *International Workshop on Engineering Multi-Agent Systems*, volume 12589 of *LNCS*, pages 61–77, 2020.

[17] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for*

the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[18] E. Bartocci and Y. Falcone. *Lectures on Runtime Verification: Introductory and Advanced Topics*. Lecture Notes in Computer Science. Springer International Publishing, 2018.

[19] Francesco Belardinelli, Panagiotis Kouvaros, and Alessio Lomuscio. Parameterised verification of data-aware multi-agent systems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 98–104. ijcai.org, 2017.

[20] Bernard Berthomieu and François Vernadat. Time Petri nets analysis with TINA. In *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*, pages 123–124. IEEE Computer Society, 2006.

[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, pages 117–148, 2003.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, LNCS, pages 193–207. Springer, 1999.

[23] Alexander Bilgram, Peter Gjøl Jensen, Thomas Pedersen, Jirí Srba, and Peter Haahr Taankvist. Methods for efficient unfolding of colored Petri nets. *Fundam. Informaticae*, 189(3-4):297–320, 2022.

[24] Michael Blondin, Alain Finkel, Christoph Haase, and Serge Haddad. Approaching the coverability problem continuously. In *TACAS*, LNCS, pages 480–496. Springer, 2016.

[25] Ahmed Bouajjani and Richard Mayr. Model checking lossy vector addition systems. In *Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *LNCS*, pages 323–333, 1999.

[26] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[27] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, 81(1):13–31, 1989.

[28] Pedro Cabalar. Temporal ASP: from logical foundations to practical use with telingo. In *Reasoning Web. Declarative Artificial Intelligence*, volume 13100 of *LNCS*, pages 94–114, 2021.

[29] Zhanghua Cai, Yantao Zhou, Yihong Qi, Weihua Zhuang, and Lei Deng. A millimeter wave dual-lens antenna for iot-based smart parking radar system. *IEEE Internet Things J.*, pages 418–427, 2021.

[30] Lorenzo Capra and Michael Köhler-Bussmeier. Modelling adaptive systems with nets-within-nets in maude. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 487–496, 2023.

[31] Pierre Chambart and Philippe Schnoebelen. Mixing lossy and perfect fifo channels. In *International Conference on Concurrency Theory*, volume 5201 of *LNCS*, pages 340–355, 2008.

[32] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 691–703. Association for Computing Machinery, 2024.

[33] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theor. Comput. Sci.*, 147(1&2):117–136, 1995.

[34] Vassilis Christophides, Richard Hull, Gregory Karvounarakis, Akhil Kumar, Geliang Tong, and Ming Xiong. Beyond discrete e-services: Composing session-oriented services in telecommunications. In *Technologies for E-Services, Second International Workshop, TES 2001, Rome, Italy, September 14-15, 2001, Proceedings*, volume 2193 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2001.

[35] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[36] Francesco Di Cosmo, Soumodev Mal, and Tephilla Prince. Deciding reachability and coverability in lossy EOS. In *Proceedings of the International Workshop on Petri Nets and Software Engineering 2024 co-located with the 45th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2024), June 24 - 25, 2024, Geneva, Switzerland*, volume 3730 of *CEUR Workshop Proceedings*, pages 74–95. CEUR-WS.org, 2024.

[37] Francesco Di Cosmo and Tephilla Prince. Bounded verification of petri nets and eoss using telingo: An experience report. In *Proceedings of the 39th Italian Conference on Computational Logic, Rome, Italy, June 26-28, 2024*, volume 3733 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2024.

[38] Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. *J. ACM*, 68(1):7:1–7:28, 2021.

[39] Silvano Dal-Zilio. MCC: A tool for unfolding colored Petri nets in PNML format. In *Application and Theory of Petri Nets and Concurrency*, volume 12152 of *Lecture Notes in Computer Science*, pages 426–435. Springer, 2020.

[40] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jirí Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *TACAS*, volume 7214 of *LNCS*, pages 492–497. Springer, 2012.

[41] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53, 01 2010.

[42] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, LNCS, pages 337–340. Springer, 2008.

[43] Nachum Dershowitz. Let's be honest. *Commun. ACM*, 64(5):37–41, 2021.

[44] Yannis Dimopoulos et al. Encoding reversing Petri nets in answer set programming. In *RC*, volume 12227 of *LNCS*, pages 264–271, 2020.

[45] Alex Dixon and Ranko Lazic. Kreach: A tool for reachability in Petri nets. In *TACAS*, LNCS, pages 405–412. Springer, 2020.

[46] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[47] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Niksic. An smt-based approach to coverability analysis. In *CAV*, LNCS, pages 603–619. Springer, 2014.

[48] Paolo Felli, Alessandro Gianola, and Marco Montali. Smt-based safety checking of parameterized multi-agent systems. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6321–6330. AAAI Press, 2021.

[49] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001. ISS.

[50] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[51] Martin Gebser et al. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

[52] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

[53] Thomas Geffroy, Jérôme Leroux, and Grégoire Sutre. Occam's razor applied to the Petri net coverability problem. *Theor. Comput. Sci.*, 2018.

[54] Michel Hack. The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems. In *15th Annual Symposium on Switching and Automata Theory*, pages 156–164. IEEE Computer Society, 1974.

[55] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. Error cost escalation through the project life cycle. *INCOSE International Symposium*, 14:1723–1737, 06 2004.

[56] Keijo Heljanko. Bounded reachability checking with process semantics. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2001.

[57] Lom-Messan Hillah et al. Extending pnml scope: A framework to combine Petri nets types. *TOPNOC*, 6:46–70, 2012.

[58] Lom-Messan Hillah, Fabrice Kordon, Laure Petrucci, and Nicolas Trèves. PNML framework: An extendable reference implementation of the Petri net markup language. In *PETRI NETS*, LNCS, pages 318–327. Springer, 2010.

[59] Ian M. Hodkinson, Roman Kontchakov, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. On the computational complexity of decidable fragments of first-order linear temporal logics. In *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003), 8-10 July 2003, Cairns, Queensland, Australia*, pages 91–98. IEEE Computer Society, 2003.

[60] Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Monodic fragments of first-order temporal logics: 2000-2001 a.d. In *LPAR*, pages 1–23, 2001.

[61] Jonas Finnemann Jensen et al. TAPAAL and reachability analysis of P/T nets. *TOPNOC*, 11:307–318, 2016.

[62] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer Publishing Company, Incorporated, 2010.

[63] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3–4):213–254, June 2007.

[64] Michael Köhler. Reachable markings of object Petri nets. *Fundamenta Informaticae*, 79(3-4):401–413, 2007.

[65] Michael Köhler-Bußmeier. A survey of decidability results for elementary object systems. *Fundamenta Informaticae*, 130(1):99–123, 2014.

[66] Michael Köhler-Bussmeier and Lorenzo Capra. Robustness: A natural definition based on nets-within-nets. In *International Workshop on Petri Nets and Software Engineering*, volume 3430 of *CEUR Workshop Proceedings*, pages 70–87, 2023.

[67] Igor V. Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Inf. Comput.*, 252:95–109, 2017.

[68] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, , S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, , J. Srba, . Thierry-Mieg, A. Walner, and K. Wolf. Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php, June 2021.

[69] F. Kordon et al. Complete Results for the Model Checking Contest. https://mcc.lip6.fr/2023/results.php, 2023.

[70] Panagiotis Kouvaros and Alessio Lomuscio. Parameterised verification for multi-agent systems. *Artif. Intell.*, 234:152–189, 2016.

[71] Orna Kupferman. Automata theory and model checking. In *Handbook of Model Checking*, pages 107–151. Springer, 2018.

[72] Slawomir Lasota. Decidability border for Petri nets with data: WQO dichotomy conjecture. In *Application and Theory of Petri Nets and Concurrency*, volume 9698 of *LNCS*, pages 20–36, 2016.

[73] Irina A. Lomazova. Nested Petri nets - a formalism for specification and verification of multi-agent distributed systems. *Fundamenta Informaticae*, 43(1-4):195–214, 2000.

[74] Riccardo De Masellis and Valentin Goranko. Logic-based specification and verification of homogeneous dynamic multi-agent systems. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 1727–1729. ACM, 2021.

[75] Richard Mayr. Lossy counter machines. Technical report, Institute of Computer Science, Technical Institute of Munich, 1998.

[76] Richard Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1-3):337–354, 2003.

[77] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[78] Artur Meski, Agata Pólrola, Wojciech Penczek, Bozena Wozna-Szczesniak, and Andrzej Zbrzezny. Bounded model checking approaches for verification of distributed time Petri nets. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011*, volume 723 of *CEUR Workshop Proceedings*, pages 72–91. CEUR-WS.org, 2011.

[79] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., USA, 1967.

[80] T. Murata. Petri nets: Properties, analysis and applications. *IEEE*, 77(4):541–580, 1989.

[81] Amat Nicolas. Benchmarks of Unbounded Petri Nets. https://github.com/nicolasAmat/SMPT, September 2022.

[82] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the ANTLR parser generator. In *PLDI*, pages 425–436, 2011.

[83] Carl Adam Petri. Kommunikation mit Automaten. Dissertation, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[84] Ramchandra Phawade, Tephilla Prince, and S. Sheerazuddin. Bounded model checking for unbounded client server systems, 2022.

[85] Ramchandra Phawade, Tephilla Prince, and S. Sheerazuddin. Dcmodelchecker 2.0: A bmc tool, 2022.

[86] Ramchandra Phawade, Tephilla Prince, and S. Sheerazuddin. Dcmodelchecker tool for verification of unbounded Petri nets, 2022.

[87] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[88] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, pages 107–122, 2002.

[89] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, pages 156–173, 2005.

[90] Tephilla Prince and Francesco Di Cosmo. Nets within nets telingo analyser, May 2024.

[91] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

[92] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in Petri net systems. In *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*, volume 4546 of *Lecture Notes in Computer Science*, pages 402–422. Springer, 2007.

[93] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in Petri net systems. *Fundam. Informaticae*, 88(3):329–356, 2008.

[94] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[95] Yunus Emre Sahin, Petter Nilsson, and Necmiye Ozay. Multirobot coordination with counting temporal logics. *IEEE Trans. Robotics*, 36(4):1189–1206, 2020.

[96] Philippe Schnoebelen. Lossy counter machines decidability cheat sheet. In *International Workshop on Reachability Problems*, volume 6227 of *LNCS*, pages 51–75, 2010.

[97] Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. Model checking unbounded concurrent lists. *Int. J. Softw. Tools Technol. Transf.*, 18(4):375–391, 2016.

[98] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[99] Yann Thierry-Mieg. Symbolic model-checking using its-tools. In *TACAS*, LNCS, pages 231–237. Springer, 2015.

[100] Yann Thierry-Mieg. Structural reductions revisited. In *PETRI NETS*, LNCS, pages 303–323. Springer, 2020.

[101] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 133–191. Elsevier and MIT Press, 1990.

[102] Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of time Petri nets with stopwatches using the state-class graph. In *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, volume 5215 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2008.

[103] Rüdiger Valk. Nets in computer organization. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 218–233, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[104] Rüdiger Valk. Object petri nets: Using the nets-within-nets paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer, 2003.

[105] Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Advances in Petri Nets*, volume 3098 of *LNCS*, pages 819–848, 2003.

[106] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986.

[107] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths (extended abstract). In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 185–194. IEEE Computer Society, 1983.

[108] Gongjun Yan, Weiming Yang, Danda B. Rawat, and Stephan Olariu. Smartparking: A secure and intelligent parking system. *IEEE Intell. Transp. Syst. Mag.*, pages 18–30, 2011.

[109] Richard M. Zahoransky, Julius Holderer, Adrian Lange, and Christian Brenig. Process analysis as first step towards automated business security. In *ECIS*, page Research Paper 46, 2016.