

Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering



MATHEMATICAL MODELING - CO2011

Assignment Symbolic and Algebraic Reasoning in Petri Nets

Class: CC04 – Instructor: Nguyễn An Khương

Students:

Name	Student ID	Workload
Nguyễn Lương Quốc Thịnh	2453203	Code Task 2, Task 3
Nguyễn Ngọc Gia Hân	2452320	Report Assembly
Nguyễn Trung Kiên	2452615	Code Task 5, Code Assembly
Phạm Minh Tiến	2453246	Code Task 4, Code Assembly
Lê Như Nhã Uyên	2453402	Code Task 1, Code Assembly

Contents

1 Task Overview	2
2 Theoretical Background	2
2.1 Petri Nets and 1-Safe Petri Nets	2
2.2 Reachability Analysis	3
2.3 Binary Decision Diagrams (BDD)	4
3 Development Environment and Design	4
3.1 Programming Language and Library Selection	4
3.2 Data Structures	5
4 Methods and Algorithms	5
4.1 Task 1: Reading Petri Nets from PNML Files	5
4.2 Task 2: Explicit Reachability Computation using BFS	6
4.3 Task 3: Symbolic Reachability using BDD	7
4.4 Task 4: Deadlock Detection (ILP + BDD)	9
4.5 Task 5: Optimization over Reachable Markings	10
5 Testing and Results	11
5.1 Test Models	11
5.2 Comparison: Explicit vs Symbolic Reachability	12
5.3 Deadlock and Optimization Results	12
6 Conclusion	13
6.1 Challenges Encountered and Possible Improvements	13
6.2 Conclusion	14



1 Task Overview

Petri nets are fundamental mathematical models for analyzing concurrent and distributed systems. This report presents an implementation that combines Binary Decision Diagrams (BDDs) and Integer Linear Programming (ILP) to efficiently analyze 1-safe Petri nets.

The implementation addresses five key tasks:

1. Parsing PNML files
2. Explicit reachability computation using BFS/DFS
3. Symbolic reachability analysis with BDDs
4. Deadlock detection using ILP
5. Optimization over reachable markings

By integrating symbolic and optimization techniques, this approach aims to overcome state-space explosion challenges while enabling formal verification of system properties.

2 Theoretical Background

2.1 Petri Nets and 1-Safe Petri Nets

A Petri net is a mathematical modeling language for describing distributed and concurrent systems. Formally, a Petri net is formally defined as a 5-tuple [5].

$$PN = (P, T, F, W, M_0)$$

where

- P is a finite set of places,
- T is a finite set of transitions, with $P \cap T = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
- $W : F \rightarrow \mathbb{N}^+$ is the weight function assigning positive integer weights to arcs,
- $M_0 : P \rightarrow \mathbb{N}$ is the initial marking.

Transition Firing:

A transition $t \in T$ is enabled at marking M if all its input places contain at least one token. When an enabled transition fires, it consumes one token from each input place and produces one token in each output place, resulting in a new marking M' [6].

1-Safe Petri Nets:

A Petri net is called 1-safe (or 1-bounded) if every place contains at most one token in any reachable marking [3]. Formally:

$$\forall M \in \text{Reach}(M_0), \forall p \in P : M(p) \leq 1 \quad (1)$$

This property enables binary encoding of markings and efficient BDD-based analysis.

2.2 Reachability Analysis

Reachability Set and Reachable Markings:

The reachability set, denoted $\text{Reach}(M_0)$, is the set of all markings that can be reached from the initial marking M_0 by firing a sequence of enabled transitions. Formally:

$$\text{Reach}(M_0) = \{M \mid M_0 \xrightarrow{\sigma} M \text{ for some firing sequence } \sigma\} \quad (2)$$

Understanding the reachability set is crucial for analyzing system behavior, verifying safety properties, and detecting potential issues such as deadlocks.

Deadlock:

A deadlock is a reachable marking where no transition is enabled [6]. Formally, a marking M is a deadlock if:

$$M \in \text{Reach}(M_0) \wedge \forall t \in T : t \text{ is not enabled at } M \quad (3)$$

State Space Explosion:

For a 1-safe Petri net with $|P|$ places, the number of reachable markings can reach $2^{|P|}$, making explicit enumeration infeasible for large systems.

Approaches to Computing Reachable Markings:

1. **Explicit Approach (BFS/DFS):** Enumerates each marking individually through graph search, providing complete reachability graphs.
2. **Symbolic Approach:** Represents entire sets of markings using compressed data structures, achieving superior scalability through exponential space savings.

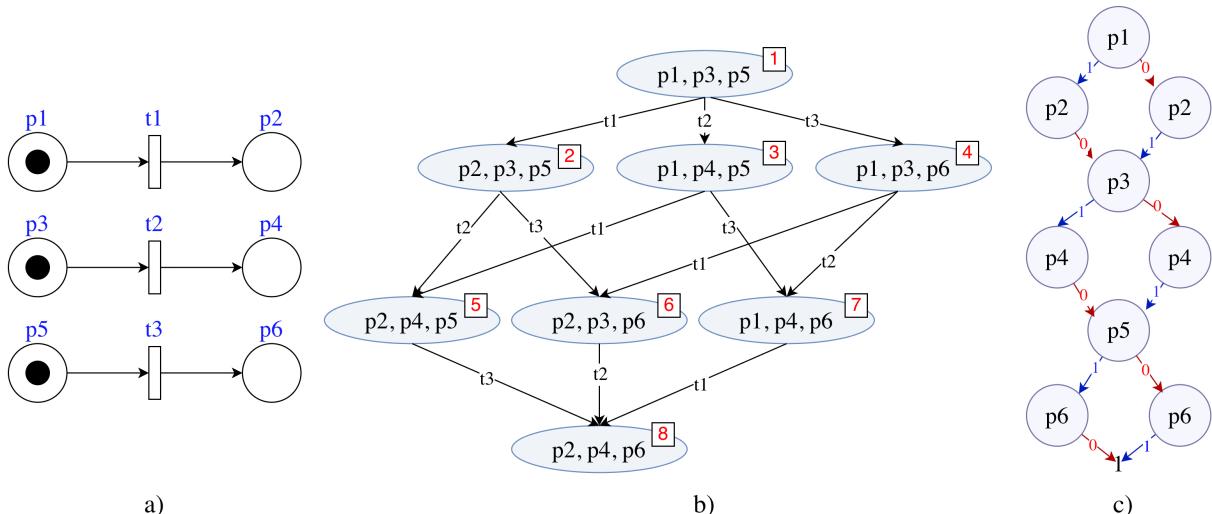


Figure 1: (a) Petri net example. (b) its explicit reachability graph with BFS orders. (c) BDD representation of the reachable markings.



2.3 Binary Decision Diagrams (BDD)

Binary Decision Diagrams (BDD)

Binary Decision Diagram (BDD) is a compact, canonical data structure for Boolean functions. It is efficient because it shares logical substructures rather than storing every state explicitly.

- **Structure:** It is a directed acyclic graph (DAG) with a single root and two leaf nodes (0 and 1) [2].
- **Nodes:** Internal nodes represent variables with two outgoing arcs (0 and 1).
- **Evaluation:** each variable assignment has a corresponding path that goes from the root node to one of the leaf nodes. The label of the leaf node is the value of the function for that assignment

Example: The BDD in **Figure 1c** represents:

$$f(p_1, \dots, p_6) = (p_1 \oplus p_2) \wedge (p_3 \oplus p_4) \wedge (p_5 \oplus p_6)$$

Symbolic Reachability Analysis:

Symbolic reachability replaces explicit state exploration with operations on BDDs representing sets of markings. Instead of generating markings one by one, the algorithm evolves a symbolic representation of all markings reachable after each step. The workflow typically consists of:

1. **Variable Declaration:** For each place p , declare Boolean variables p (current state) and p' (next state)
2. **Transition Relation:** Build global transition relation $T(s, s')$ encoding all possible state transitions [6]
3. **Fixpoint Computation:** Iteratively compute reachable states until convergence ($R_{new} = R$):

$$R_{new} = R \cup \text{Post}(R), \quad \text{where } \text{Post}(R) = \exists s. (R(s) \wedge T(s, s')) \quad (4)$$

This approach reduces memory usage and overcomes state explosion by manipulating entire sets of states symbolically through Boolean operations.

3 Development Environment and Design

3.1 Programming Language and Library Selection

The implementation is developed in **Python 3.9**. The following libraries are utilized:

- **xml.etree.ElementTree:** Built-in XML parser for reading PNML files
- **dd** (specifically `dd.autoref`): used for symbolic reachability analysis [1]
- **pulp:** ILP solver used for deadlock detection. [7]
- **pytest:** Testing framework for validating correctness of modules.



3.2 Data Structures

a) PetriNet Class

```
class PetriNet:
    places: set           # Set of place ids
    transitions: set      # Set of transition ids
    arcs: dict            # {id: {'input': [...], 'output': [...]}}
    initial_marking: Marking # Initial token distribution
    place_names: dict     # {place_id: place_name}
    transition_names: dict # {transition_id: transition_name}
```

Design Rationale:

- Sets for places/transitions: $O(1)$ membership testing
- Dictionary for arcs: Indexed by transition for efficient enabling checks
- Arcs stored by transition enable $O(1)$ access for transition-centric operations

b) Marking Class

```
class Marking:
    marking: dict  # {place_id: bool}
```

Key Features:

- **Binary representation:** Each place stores boolean (0 or 1 token)
- **Hashable:** Implements `__hash__()` for use in sets/dictionaries during analysis
- **Immutable operations:** Methods like `fire_transition()` return new markings

4 Methods and Algorithms

4.1 Task 1: Reading Petri Nets from PNML Files

The PNML parser follows a sequential pipeline with immediate 1-safe validation at parse time.

Input: PNML file path

Output: Petri Net $N = (P, T, F)$ with initial marking M_0 or validation error

1. XML Parsing

- $tree \leftarrow \text{ParseXML}(filepath)$
- $root \leftarrow tree.getroot()$



2. Namespace Handling

- $ns \leftarrow \{ \text{'pnml'} : \text{'http://www.pnml.org/version-2009/grammar/pnml'} \}$
- $places \leftarrow \text{root.findall}(\cdot//\text{pnml:place}', ns)$

3. Place Extraction with 1-Safe Validation

- $P \leftarrow \emptyset, M_0 \leftarrow \{ \}$
- For each place element in $places$:
 - $p \leftarrow \text{place.get('id')}$
 - $token_count \leftarrow \text{extract from initialMarking (default 0)}$
 - If $token_count > 1$: raise ValueError("Not 1-safe")
 - $M_0(p) \leftarrow (token_count = 1)$ // Binary: True or False
 - $P \leftarrow P \cup \{p\}$

4. Transition Extraction

- $T \leftarrow \emptyset$
- For each transition element: $T \leftarrow T \cup \{t\}$

5. Arc Extraction

- $F \leftarrow \emptyset$
- For each arc element:
 - Extract *source* and *target*
 - If $\text{source} \in P$ and $\text{target} \in T$: $F \leftarrow F \cup \{(\text{source}, \text{target})\}$ // Input
 - If $\text{source} \in T$ and $\text{target} \in P$: $F \leftarrow F \cup \{(\text{source}, \text{target})\}$ // Output

6. Consistency Validation

- For each $t \in T$: verify t has connected arcs
- For each $p \in P$: if $p \notin M_0$, set $M_0(p) \leftarrow 0$

4.2 Task 2: Explicit Reachability Computation using BFS

a) Theoretical Background

The explicit reachability graph (ERG) is defined as:

$$G = (R, E), \quad E = \{(M, t, M') \mid M \xrightarrow{t} M'\},$$

where R is the set of markings reachable from the initial marking M_0 . BFS explores markings in increasing distance from M_0 and avoids redundancy using a visited set.



b) Algorithm

Algorithm: Explicit Reachability using BFS

Require: Petri Net $N = (P, T, F)$, Initial marking M_0
Ensure: Reachable markings R , Transition graph G

1. Initialize:

$Q \leftarrow \{M_0\}$, $R \leftarrow \{M_0\}$, $G \leftarrow \emptyset$

2. BFS Traversal:

while $Q \neq \emptyset$ **do**

$M \leftarrow Q.\text{dequeue}()$

for each enabled transition t at M **do**

$M' \leftarrow \text{Fire}(t, M)$

$G \leftarrow G \cup \{(M, t, M')\}$

if $M' \notin R$ **then**

 add M' to R and Q

end if

end for

end while

3. Return: R, G

Complexity: Time $O(S \cdot |T|)$, Space $O(S \cdot |P| + S \cdot |T|)$. where S = reachable states

4.3 Task 3: Symbolic Reachability using BDD

a) Theoretical Background

Variable Declaration

For each place $p \in P$, we declare two Boolean variables:

- p : Current state variable
- p' : Next state variable

Transition Relation

For each transition $t \in T$, the enabling condition and transition relation are:

$$\text{enabled}_t(s) = \bigwedge_{p \in \text{pre}(t)} p \quad (5)$$

$$T(s, s') = \bigvee_t (\text{enabled}_t(s) \wedge \text{update}_t(s, s')) \quad (6)$$

where $\text{update}_t(s, s')$ encodes the state changes:



- $p \in \text{post}(t) \rightarrow p' = 1$ (token produced)
- $p \in \text{pre}(t) \rightarrow p' = 0$ (token consumed)
- Otherwise $\rightarrow p' = p$ (unchanged)

Initial Marking

The initial marking M_0 is encoded as a BDD over current state variables.

b) Algorithm

The symbolic reachability computes the fixpoint of reachable states using BDD operations:

Algorithm: Symbolic Reachability (BDD Fixpoint)

Require: Petri Net $N = (P, T, F)$, Initial marking M_0

Ensure: BDD \mathcal{R} representing all reachable markings

1. Initialize

$R \leftarrow \text{ENCODEBDD}(M_0)$ ▷ Encode initial marking as BDD

2. Fixpoint Computation

while True **do**

$Post_R \leftarrow \exists(\text{current_vars}, R \wedge T)$ ▷ Compute successors

$Post_R \leftarrow \text{RENAME}(Post_R, p' \rightarrow p)$ ▷ Map to current variables

$R_{\text{new}} \leftarrow R \vee Post_R$ ▷ Union with previous states

if $R_{\text{new}} = R$ **then** ▷ Fixpoint reached

break

end if

$R \leftarrow R_{\text{new}}$

end while

3. Extract Markings (Optional)

Use $\text{PICKITER}(R)$ to enumerate all satisfying assignments

return R

Key BDD Operations:

- \exists : Existential quantification to compute successor states
- let: Variable substitution ($p' \rightarrow p$)
- pick_iter: Enumerate concrete markings from BDD

Complexity Analysis:

- **Time:** $O(k \cdot |T|)$ where k = iterations to fixpoint, $|T|$ = number of transitions



- **Space:** $O(\log(S))$ where $S =$ number of reachable states (typically much smaller than explicit representation)

4.4 Task 4: Deadlock Detection (ILP + BDD)

a) Theoretical Background

Variables:

$$x_p \in \{0, 1\} \quad \text{for each place } p \in P,$$
$$n_t \in \mathbb{Z}_{\geq 0} \quad \text{for each transition } t \in T.$$

State Equation: State equation ensures that *every ILP candidate marking corresponds to a marking that is consistent with some firing vector* [3]. Although this does not guarantee reachability, it significantly restricts the search space.

$$x_p = M_0(p) + \sum_{t \in T} C(p, t) n_t, \quad \forall p \in P.$$

Here $C(p, t)$ is the incidence matrix.

Deadlock Constraints: A transition t is *disabled* if at least one input place is empty. To find a dead marking, we require **all** transitions to be disabled [3, 6]. The constraints are:

$$\text{Subject to: } \forall t \in T : \sum_{p \in \bullet t} n_t \quad (7)$$

Objective Function: Minimizing the total number of transition firings biases the ILP solver toward markings that are “close” to the initial marking.

$$\min \sum_{t \in T} n_t.$$

Canonical Cut (No-Good Constraint): If M_{cand} is unreachable, we eliminate it from the search space. The “No-good” constraint ensures that any future solution must differ from M_{cand} by at least one bit (place):

$$\sum_{p \in P : M_{cand}(p)=1} (1 - x_p) + \sum_{p \in P : M_{cand}(p)=0} x_p \geq 1 \quad (8)$$

b) Algorithm

We propose a "Generate and Test" iterative algorithm as follows:
This hybrid approach avoids the exhaustive enumeration of the state space, particularly advantageous in scenarios where the number of deadlocks is significantly smaller than the number of all reachable markings ($|D| \ll |Reach(M_0)|$).

In such "sparse deadlock" cases, checking dead marking property for every single reachable marking may be costly. In contrast, the ILP solver targets the small number of dead markings directly. Since BDD performs reachability checks in $O(h)$ time (where h is the BDD height), the verification step is negligible, making this method significantly faster than explicit search strategies.

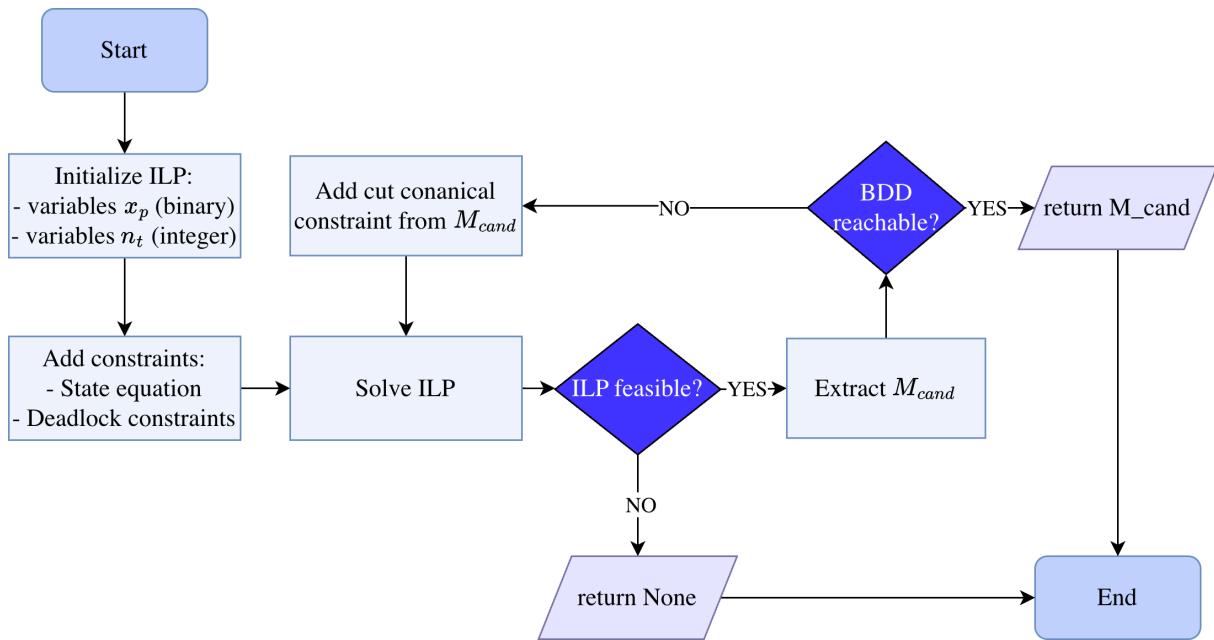


Figure 2: Flowchart for ILP+BDD Deadlock detector

4.5 Task 5: Optimization over Reachable Markings

a) Method

Problem Definition

Given a weight vector c for places, find marking M that maximizes:

$$\max c^T M, \quad \text{subject to } M \in \text{Reach}(M_0) \quad (9)$$

Constraints:

- $\text{Reach}(M_0)$ denotes the set of markings reachable from the initial marking
- For 1-safe nets: $M(p) \in \{0, 1\}$ for all places p
- Objective function simplifies to: $\max \sum_{p \in P: M(p)=1} c_p$

Approach Selection

We employ a **Symbolic Branch-and-Bound** algorithm that operates directly over BDD computed in Task 3. By maintaining an "optimistic bound" based on suffix sums, the algorithm proves which branches cannot yield a better solution and prunes them early.

b) Algorithm

The algorithm proceeds in four specific steps:

1. **Preprocessing (Variable Ordering & Lookahead):** Variables are sorted by descending weight to prioritize high-value decisions. A suffix sum array is precomputed to enable $O(1)$ calculation of the maximum possible remaining score at any depth.

2. **Initialization (Baseline Selection):** The solver retrieves a single valid reachable marking from the BDD (via arbitrary path selection) to establish an initial global lower bound ($Score_{best}$). This ensures the pruning mechanism has a valid baseline to discard inferior branches immediately.
3. **Recursive DFS with Pruning:** The BDD is traversed using Depth-First Search. At each node, the algorithm performs:
 - **Optimistic Bound:** Calculate $Bound = CurrentScore + SuffixSum(remaining)$.
 - **Pruning:** If $Bound \leq Score_{best}$, the branch is discarded as it cannot improve the solution.
 - **Branching:** If not pruned, the search attempts to extend the path:
 - First, try the **High Branch** ($p = 1$). Check validity via BDD intersection; if valid, recurse adding weight.
 - Second, try the **Low Branch** ($p = 0$). Check validity; if valid, recurse without adding weight.
4. **Termination:** The algorithm terminates when all paths are explored or pruned, returning the marking with the global maximum score.

Complexity Analysis:

- **Time:** Worst-case $O(2^{|P|})$ for unpruned search, but efficient in practice due to Branch-and-Bound cuts and BDD compression.
- **Space:** $O(|P|)$ for the recursion stack and auxiliary arrays.

5 Testing and Results

5.1 Test Models

We prepared several small-to-average Petri net models in PNML format to evaluate the performance of our implementation. Some were obtained from the Model Checking Contest (MCC) [4] repository¹. Table 1 summarizes their structural characteristics.

Table 1: Summary of test Petri net models used in experiments.

Model	#Places	#Transitions	#Arcs	Notes
Simple01	3	2	4	linear; no branches, no cycles.
Simple02	5	4	10	with sync, fork, and cycles.
CircadianClock01	14	16	58	deadlock-free, conservative.
AutonomousCar03a	41	121	745	dense, connected.
AutonomousCar04a	49	193	1306	dense, connected.

All experiments were executed on an M4 MacBook Air (15-inch) with 10-core CPU, 10-core GPU, 16 GB unified memory, and a 512 GB SSD.²

¹<https://mcc.lip6.fr/2025/models.php>

²<https://support.apple.com/en-us/122210>



5.2 Comparison: Explicit vs Symbolic Reachability

In this section, we evaluate and compare the performance of explicit-state reachability (Task 2) and symbolic BDD-based reachability (Task 3) across the test models.

Table 2: Explicit vs Symbolic Reachability Performance

Model	#Reachable markings	Explicit Time(s)	#BDD Nodes	Symbolic Time(s)
Simple01	3	0.0001	5	0.0011
Simple02	5	0.0002	13	0.0034
CircadianClock01	128	0.0198	21	0.0298
AutonomousCar03a	22521	4.5900	492	1.1388
AutonomousCar04a	206492	63.6838	698	2.9329

Observations:

- **Explicit reachability scales poorly with state-space size.** For small models, explicit BFS completes almost instantly. However, for larger models, the number of reachable markings grows dramatically causing explicit search time to rocket.
- **Symbolic BDD-based reachability benefits from structural sharing.** Even when the number of reachable markings grows significantly, the BDD size increases modestly (e.g., 492 → 698 nodes), showing that the symbolic method captures concurrency and repeated structure compactly.
- **Symbolic reachability outperforms explicit search on medium-to-large models.** For *AutonomousCar03a*, symbolic computation (1.1388 s) is almost 3× faster than explicit search (4.5900 s).
- **BDD size does not correlate linearly with the number of reachable markings.** For example, *AutonomousCar04a* has about 9× more states than *AutonomousCar03a*, but the BDD grows only from 492 to 698 nodes.

5.3 Deadlock and Optimization Results

a) Deadlock Detection

Table 3: Deadlock Detection Results (ILP + BDD)

Model	ILP Time(s)	Total Time(s)	Deadlock Found?
Simple01	0.042541	0.0434	Y
Simple02	0.064116	0.0649	N
CircadianClock01	0.041242	0.0423	N
AutonomousCar03a	0.088908	0.0992	Y
AutonomousCar04a	0.107402	0.1287	Y

Observations:

- **Complexity vs. Execution Time:** ILP time does not grow linearly suggests that ILP difficulty is driven by constraint density rather than just model size.
- **Overall Efficiency:** All tested models were processed in under 0.2 seconds, confirming that the ILP reduction effectively minimizes the computational overhead.
- **Dominance of ILP Phase:** The total execution time is primarily determined by the frequency and complexity of the ILP calculations, showing that BDD verification step is negligible.

b) Optimization Over Reachable Markings

To evaluate the optimization capabilities, we defined two objective weight vectors (c):

1. **Alternating (Strategy A):** The set of places P is sorted lexicographically. The k -th place receives a weight of $c_i = (-1)^{k+1} \cdot k$.
Example: p_1, p_2, p_3, p_4 have the coefficients $c_1 = 1, c_2 = -2, c_3 = 3, c_4 = -4, \dots$
2. **Uniform (Strategy B):** All places are assigned a weight of $c_i = 1$.

Table 4: Optimization Results: Alternating vs. Uniform Strategies

Model	Alternating ($c \approx \pm k$)			Uniform ($c = 1$)		
	Obj	Value	Time(s)	Obj	Value	Time(s)
Simple01		3	0.0002		1	0.0003
Simple02		6	0.0005		2	0.0008
CircadianClock01		49	0.0010		7	0.0096
AutonomousCar03a		201	0.2004		8	5.4819
AutonomousCar04a		278	0.3813		9	61.5810

Observations:

- **Performance Analysis:** The significant latency in the Uniform strategy is due to the inefficiency of BB pruning. In the Alternating strategy, large weight magnitudes allow tight upper bounds and prune sub-optimal branches early. However, under Uniform weights, upper bound decreases slowly. The solver needs to explore nearly the entire state space to distinguish between markings with N and $N + 1$ tokens.
- **Capacity Analysis:** The results from the Uniform strategy reveal the maximum token capacity for bounded nets, providing a baseline for structural complexity.

6 Conclusion

6.1 Challenges Encountered and Possible Improvements

Throughout the implementation, the following challenges were encountered:



- **ILP–BDD deadlock detection:** ILP solving time dominated the deadlock detection runtime. We improved performance by adding marking-equation constraints to the ILP, reducing the number of ILP calls from thousands to just tens.
- **Optimization over reachable markings:** Iterating over candidate markings without enumerating the full reachable set proved inefficiency. To address this, we applied a Branch-and-Bound strategy combined with BDD constraints. While this issue can also be resolved using the same method as Task 4, BB are utilized to introduce a different approach on BDD structure.

6.2 Conclusion

This project successfully implements a comprehensive framework for analyzing 1-safe Petri nets through integration of symbolic computation and optimization techniques.

The symbolic BDD approach demonstrates significant advantages over explicit methods for large state spaces through compact representation and simultaneous state exploration. This implementation bridges theoretical understanding of Petri nets with practical computational analysis, providing a foundation for formal verification and system optimization.

References

- [1] “dd: Binary decision diagrams,” PyPI. [Online]. Available: <https://pypi.org/project/dd/>.
- [2] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, “Deriving Petri nets from finite transition systems,” *IEEE Transactions on Computers*, vol. 47, no. 8, pp. 859–882, Aug. 1998.
- [3] V. Khomenko and M. Koutny, “Verification of Bounded Petri Nets Using Integer Programming,” Department of Computing Science, University of Newcastle, Newcastle upon Tyne, U.K., Tech. Rep.
- [4] “Model Checking Contest 2025 - Models,” [Online]. Available: <https://mcc.lip6.fr/2025/models.php>.
- [5] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.
- [6] E. Pastor, J. Cortadella, and O. Roig, “Symbolic analysis of bounded Petri nets,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 4, pp. 432–448, Apr. 1999.
- [7] “PuLP: A Linear Programming Toolkit for Python,” PyPI. [Online]. Available: <https://pypi.org/project/PuLP/>.