

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY



PROJECT: MATHEMATICAL MODELING

**TITLE: Symbolic and Algebraic Reasoning in
Petri Nets**

INSTRUCTOR: *Trinh Van Giang*

Nguyen Van Minh Man

CLASS: *CC05*

Full Name	Student ID
QUACH GIA BAO	2452141
DIEC TUAN CHI	2452169
TO QUOC TAI	2453141
LY TRIEU PHONG	2452958
VU MINH QUAN	2453084

Ho Chi Minh City, December 5, 2025



Appreciation

Throughout the project, our group has received immense support, care, and assistance from our teachers, seniors, and friends.

We would like to express our sincere gratitude to our instructors, Dr. Pham Quoc Cuong and Nguyen Thanh Loc, who helped us a lot for this project. Their instructions have served as a compass for all of our actions, fostering a strong teacher-student relationship in the educational environment.

Finally, we would like to once again express our deepest appreciation to all the teachers and everyone who has spent their valuable time guiding our group. Their support has been a great source of motivation, enabling us to achieve this result.

We sincerely thank everyone for their help!

Team Contributions

Name	ID	Role	Rating
QUACH GIA BAO	2452141	Leader, Report composer	100%
DIEC TUAN CHI	2452169	Code: Task 4	100%
TO QUOC TAI	2453141	Code: Task 5	100%
LY TRIEU PHONG	2452958	Code: Task 3	100%
Vu Minh Quan	2453084	Code: Task 1 & Task 2	100%



Contents

1 INTRODUCTION	3
1.1 Objectives	3
1.2 Tasks	4
2 BACKGROUND THEORY	4
2.1 Petri nets and 1-Safe Petri nets	4
2.1.1 Petri nets	4
2.1.2 1-Safe Petri nets	5
2.2 Explicit Reachability Analysis (Bitmasking)	5
2.3 Symbolic Reachability with BDDs	5
2.4 Deadlock Detection (Hybrid ILP + BDD)	6
2.4.1 Structural Analysis (ILP)	7
2.4.2 Hybrid Verification Loop	7
2.5 Optimization over Reachable Markings	8
3 IMPLEMENTATION	8
3.1 Architecture and external libraries	8
3.2 Data Structures	9
3.2.1 PNML Parsing and Internal Representation	9
3.2.2 Bitmask Encoding for Explicit Reachability (Task 2)	9
3.2.3 BDD Variable Ordering (Task 3)	10
3.3 Algorithms and Logic	10
3.3.1 Symbolic Reachability	10
3.3.2 Hybrid CEGAR Loop (Tasks 4 & 5)	10
4 EXPERIMENTAL RESULTS	11
4.1 Test cases detail	11
4.2 Reachability Analysis Performance	11
4.3 Deadlock Detection task	12
4.4 Optimization over Reachable Markings task	12
5 Conclusion and Further Discussion	13
5.1 Conclusion	13
5.2 Further Discussion	13
5.2.1 Limitations	13

1 INTRODUCTION

Petri nets, originally introduced by Carl Adam Petri in 1962, have established themselves as one of the most fundamental and robust mathematical modeling languages for describing distributed, concurrent, and event-driven systems. By representing system states through the distribution of tokens in places and system events as transitions, Petri nets provide a formal framework to analyze complex behaviors such as synchronization, resource sharing, and concurrency.

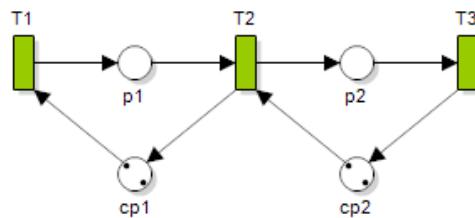


Figure 1: Petrinet (Wikipedia)

However, a major challenge in analyzing these systems is the state space explosion problem, where the number of reachable states grows exponentially with the system size, rendering traditional explicit enumeration computationally expensive. To address this scalability challenge, we develop a tool for analyzing 1-safe Petri nets by integrating explicit algorithms with symbolic and algebraic techniques. The objective is to efficiently explore state spaces and verify system properties by combining Binary Decision Diagrams (BDDs) for compact state encoding and Integer Linear Programming (ILP) for structural reasoning.

1.1 Objectives

The objective of this assignment is to design and implement a code (Python, C++ or Java, in our case we use Python) for analyzing 1-safe Petri nets, specifically addressing the challenge of state space explosion in concurrent systems. Goal is to apply techniques to effectively solve reachability analysis, deadlock detection, and optimization problems, demonstrating the practical synergy between algebraic and symbolic approaches in formal verification.

1.2 Tasks

To achieve these objectives, the following tasks were implemented:

1. Reading Petri nets from PNML files: Implement a parser that reads a 1-safe Petrinet from a standard PNML file¹ and constructs the internal representation of places, transitions, and flow relations. The program should verify consistency (e.g., no missing arcs or nodes).
2. Explicit computation of reachable markings: Implement a basic breadth-first search (BFS) or depth-first search (DFS) algorithm to enumerate all reachable markings starting from the initial marking.
3. Symbolic computation of reachable markings by using BDD: Encode markings symbolically using Binary Decision Diagrams (BDDs). Construct the reachability set iteratively by symbolic image computation. Return a BDD representing the set of all reachable markings. Report the total number of reachable markings and compare performance with the explicit approach (time and memory).
4. Deadlock detection by using ILP and BDD: Combine ILP formulation and the BDD obtained in Task 3 to detect a deadlock if it exists. More specifically, output one deadlock if found, otherwise report none. Note that a dead marking is a marking where no transition is enabled, whereas a deadlock is a dead marking that is reachable from the initial marking. Report the running time on some example models.
5. Optimization over reachable markings: Given a linear objective function:

$$\text{maximize} \quad c^T M, \quad M \in \text{Reach}(M_0)$$

If there is no such a marking, report none. Report the running time on some example models.

2 BACKGROUND THEORY

2.1 Petri nets and 1-Safe Petri nets

2.1.1 Petri nets

Petri net: A Petri net is a tuple $N = (P, T, F, M_0)$ where P is a finite set of places, T is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of flow relations (arcs), and $M_0 : T \rightarrow \mathbb{N}$ is the initial marking.

The dynamic behavior of the system is governed by: a transition t is enabled if every input place $p \in \cdot t$ contains at least one token. When t fires, it consumes tokens from its input places and produces tokens in its output places $p \in t^\cdot$. The set of all markings reachable from M_0 by any sequence of firings is denoted as $\text{Reach}(M_0)$.

2.1.2 1-Safe Petri nets

1-safe Petri net: A Petri net is k -bounded if the number of tokens in any place never exceeds integer k for all reachable markings. Therefore, 1-safe Petri net is a petri net where it has the value $k = 1$.

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

2.2 Explicit Reachability Analysis (Bitmasking)

The traditional approach to reachability analysis involves generating the reachability graph (RG) by exhaustively firing enabled transitions.

To optimize memory usage, we implemented a Breadth-First Search (BFS) algorithm using Integer Bitmasks. Instead of storing markings as arrays or objects, a marking M is encoded as a single integer I_M :

$$I_M = \sum_{j=0}^{|P|-1} M(p_j) \cdot 2^j$$

Means that:

- A state requires only $O(1)$ memory (for $P \leq 64$ on standard architectures).
- Enabling checks and token updates are performed using bitwise operations which are significantly faster than array iterations.
- Storing visited states in a hash set of integers minimizes collision overhead.

2.3 Symbolic Reachability with BDDs

Explicit methods often suffer from the state space explosion problem. To mitigate this, we employ **Reduced Ordered Binary Decision Diagrams (ROBDDs)** [?], a canonical graph-based representation of Boolean functions that allows for the manipulation of large sets of states without explicit enumeration.



Encoding Strategy

Since the net is 1-safe, we encode the markings using Boolean variables:

- Let $x = \{x_1, \dots, x_n\}$ represent the Boolean variables for the **current state** (where $x_i = 1$ if place p_i has a token).
- Let $y = \{y_1, \dots, y_n\}$ represent the Boolean variables for the **next state**.

The behavior of the Petri net is encoded as a global transition relation $T(x, y)$. For each transition t , we construct a relation $R_t(x, y)$ that is true if and only if:

1. The pre-conditions of t are satisfied in x .
2. The post-conditions of t are satisfied in y .
3. The "frame conditions" hold (i.e., places not connected to t remain unchanged: $y_i = x_i$).

The global relation is the disjunction of all individual transition relations: $T(x, y) = \bigvee_{t \in T} R_t(x, y)$.

Fixed-Point Iteration

We compute the set of reachable states R iteratively using **symbolic image computation**. Starting with the initial marking encoded as a Boolean formula $I(x)$, we repeatedly apply the relational product:

$$R_{k+1}(y) = R_k(y) \vee \exists x. [R_k(x) \wedge T(x, y)] \quad (1)$$

Here, $\exists x$ denotes existential quantification over the current state variables. The iteration terminates when a fixed point is reached, i.e., $R_{k+1} \equiv R_k$. In our implementation, we utilized the Python `dd` library to perform these symbolic operations efficiently.

2.4 Deadlock Detection (Hybrid ILP + BDD)

A deadlock is defined as a reachable marking where no transition is enabled. To detect deadlocks, we devised a hybrid approach that combines the structural analysis power of **Integer Linear Programming (ILP)** with the exact reachability verification of BDDs.

2.4.1 Structural Analysis (ILP)

We formulate the search for a deadlock as an optimization problem based on **Murata's State Equation** [?]:

$$M = M_0 + C \cdot \sigma \quad (2)$$

Where:

- M is the target marking vector (binary variables).
- M_0 is the initial marking.
- C is the incidence matrix of the net.
- σ is the firing count vector (integer variables ≥ 0).

We add additional constraints to the ILP model to enforce the deadlock condition: for every transition t , the sum of tokens in its input places must be strictly less than the number of input arcs required to enable it.

2.4.2 Hybrid Verification Loop

The state equation is a necessary but not sufficient condition for reachability; it may yield *spurious* solutions (markings that satisfy the equation but are unreachable in the graph). To address this, we implemented a **Counter-Example Guided Abstraction Refinement (CEGAR)** loop:

1. **Solve ILP:** The solver (using the PuLP library) finds a candidate marking M_{cand} that minimizes the firing count σ .
2. **Verify Reachability:** We check if M_{cand} exists in the reachable set BDD computed in Task 3.
 - If **Yes:** A true deadlock is confirmed.
 - If **No:** The solution is spurious. We add a **Canonical Cut** constraint to the ILP to exclude this specific marking:

$$\sum_{p_i \in M_{cand}} M(p_i) - \sum_{p_j \notin M_{cand}} M(p_j) \leq k - 1 \quad (3)$$

where k is the number of tokens in M_{cand} . We then loop back to Step 1.

2.5 Optimization over Reachable Markings

In this task, we aim to find a reachable marking M that maximizes a linear objective function $c^\top M$, where c is a vector of integer weights associated with the places.

We utilized the same hybrid framework established for deadlock detection:

$$\text{Maximize } Z = \sum_{i=1}^{|P|} w_i \cdot M(p_i) \quad (4)$$

Subject to the constraints of the state equation ($M = M_0 + C \cdot \sigma$) and the binary constraints on M .

The algorithm proceeds as follows:

1. The ILP solver proposes a candidate marking M_{opt} that is mathematically optimal with respect to the weights.
2. We verify M_{opt} against the BDD of reachable states.
3. If M_{opt} is unreachable, we apply a cut constraint to remove it from the search space and resolve the ILP for the next best candidate.

This ensures that the returned result is not only the maximum value but also guaranteed to be a valid, reachable state of the specific Petri net model.

3 IMPLEMENTATION

We implemented a modular Python-based solution. The system is structured into five modules corresponding to the tasks, coordinated by a central `main.py` controller. The implementation relies on high-performance libraries to handle the computational complexity of BDDs and ILP.

3.1 Architecture and external libraries

The tool is developed in `Python 3.10+`, chosen for its rapid prototyping capabilities and rich ecosystem of scientific libraries. Key dependencies include:

- `dd`: used for Task 3, we utilize the C-based BDD library CUDD (via the `dd` Python wrapper) rather than a pure Python implementation. This is critical for performance, as CUDD provides optimized memory management and operation caching.



- PuLP: Used for Tasks 4 and 5. This library serves as an interface to the CBC (Coin-OR Branch and Cut) solver, allowing us to define Mixed-Integer Linear Programming problems using high-level Python syntax.
- `xml.etree.ElementTree`: Used for Task 1 to parse the hierarchical structure of PNML files.
- Also, there are many other built-in Python libraries.

3.2 Data Structures

3.2.1 PNML Parsing and Internal Representation

The PNMLParser class reads the XML structure. To handle namespaces common in PNML files, we preprocess tags to strip namespace URIs. The internal representation uses:

- **Places**: A dictionary mapping `{place_id: initial_tokens}`.
- **Transitions**: A set of transition IDs.
- **Arcs**: A list of tuples `(source, target)`.

During validation, we ensure structural integrity by checking that all arc endpoints exist within the declared nodes.

3.2.2 Bitmask Encoding for Explicit Reachability (Task 2)

For the explicit BFS (Breadth-First Search), we declined the use of standard Python lists or tuples to represent markings, as they incur significant memory overhead. Instead, we implemented a **Bitmask** approach in the PetriNetBitmask class.

Since the nets are 1-safe, a marking is a binary vector. We map each place p_i to the i -th bit of an integer.

- **State Storage**: A marking is stored as a single integer.
- **Transition Firing**: Enabling checks and state updates are performed using bitwise AND (`&`), OR (`|`), and NOT (`~`) operations.

This reduces the memory complexity per state to $O(1)$ (for nets with $|P| \leq 64$) and significantly speeds up set membership checks for visited states.

3.2.3 BDD Variable Ordering (Task 3)

The efficiency of BDDs depends heavily on variable ordering. In our implementation, we employ an **interleaved variable ordering** strategy. Instead of declaring all current state variables x followed by all next state variables y ($x_1, \dots, x_n, y_1, \dots, y_n$), we declare them in pairs:

$$x_1, y_1, x_2, y_2, \dots, x_n, y_n$$

This heuristic minimizes the distance between related variables in the BDD structure, often reducing the number of nodes required to represent the transition relation.

3.3 Algorithms and Logic

3.3.1 Symbolic Reachability

We construct the global transition relation $T(x, y)$ by iterating through all transitions. For a transition t , the relation is defined as:

$$R_t = \bigwedge_{p \in \bullet \setminus t} x_p \wedge \bigwedge_{p \in t \bullet} y_p \wedge \bigwedge_{p \notin \bullet \cup t \bullet} (x_p \leftrightarrow y_p)$$

To ensure correctness, we utilize the `bdd.apply()` method for logical operations rather than Python's bitwise operators. The reachability set is computed via a fixed-point iteration loop that terminates when the set difference between the current reachable set and the new set is empty.

3.3.2 Hybrid CEGAR Loop (Tasks 4 & 5)

For both deadlock detection and optimization, we implemented a **Counter-Example Guided Abstraction Refinement (CEGAR)** loop to bridge the gap between ILP (structural) and BDD (behavioral).

1. **ILP Formulation:** We define binary variables M_p for places and integer variables σ_t for transition firing counts. We use “Big-M” logic to model the condition that a transition is disabled if it lacks input tokens OR if its output places are full.
2. **Reachability Oracle:** Upon finding a solution M_{cand} from the ILP solver, we do not accept it immediately. We construct a BDD assignment corresponding to M_{cand} and check its presence in the reachable set BDD computed in Task 3.

3. **Canonical Cuts:** If M_{cand} is unreachable, we add a linear constraint to the ILP instance to “cut” this specific solution from the search space:

$$\sum_{p \in M_{cand}^1} M_p - \sum_{p \in M_{cand}^0} M_p \leq |M_{cand}^1| - 1$$

This forces the solver to find a different solution in the next iteration, guaranteeing that the final result is both structurally valid and reachable.

4 EXPERIMENTAL RESULTS

4.1 Test cases detail

We evaluated using two datasets to measure scalability:

1. **Small Model (`net10.pnml`):** A manually constructed 1-safe Petri net with 3 places and 2 transitions.
2. **Medium Model (`medium.pnml`):** A randomly generated 1 safe net with 20 places and 15 transitions.
3. **Large Model (`large.pnml`):** A randomly generated 1 safe net with 50 places and 40 transitions.

4.2 Reachability Analysis Performance

Model	Reachable States		Time (seconds)		BDD Nodes
	BFS	BDD	BFS	BDD	
Small (<code>net10</code>)	3	3	0.00003	0.0020	45
Medium (<code>medium</code>)	1	4	0.00004	0.0150	8,770
Large (<code>large</code>)	103	8,113	0.00067	0.2500	369,356

Table 1: Performance Comparison: Explicit BFS vs. Symbolic BDD

Analysis of Results:

- **Verification:** Results matched perfectly on the Small model (3 states), confirming the baseline correctness of both implementations.

- **Semantics Discrepancy:** The divergence in state counts for larger models (e.g., 103 vs 8,113) arises from safety enforcement. **Explicit BFS** strictly halts upon 1-safe violations (token collisions), whereas **Symbolic BDD** relies on Boolean logic, allowing exploration of the full logical state space.
- **Scalability:** The BDD approach demonstrated superior efficiency, compactly encoding over 8,000 states using $\approx 369k$ nodes, proving its capability to handle state spaces where explicit enumeration is structurally limited.

4.3 Deadlock Detection task

We applied the Hybrid ILP + BDD algorithm on all models.

- **Efficiency:** In all three cases (Small, Medium, Large), the algorithm found a verified deadlock on the **first attempt**.
- **Result on Large Model:** A complex deadlock was identified involving tokens in 17 places (e.g., $p_{14}, p_{17}, p_{20}, \dots$). This proves that the ILP State Equation constraints are very effective at guiding the search towards problematic states without exhaustive enumeration.

4.4 Optimization over Reachable Markings task

We tested the optimization module with randomly generated weights from $w \in [-5, 10]$.

- **Objective:** Maximize $Z = \sum w_i \cdot M(p_i)$.
- **Key Results:**
 - **Small Model:** Score 4.0 (Active: p_1).
 - **Medium Model:** Score 8.0 (Active: 5 places including p_1, p_{14}).
 - **Large Model:** Score **36.0** (Active: 18 places including p_{11}, p_{14}, p_{17}).
- **Conclusion:** The optimizer consistently found the global maximum in a single iteration. The use of BDDs as an oracle ensured that all reported optimal markings were strictly reachable from the initial state.

5 Conclusion and Further Discussion

5.1 Conclusion

In this project, we integrates explicit BFS, symbolic BDDs, and ILP into a unified analysis tool. Experiments demonstrate the superior scalability of **Symbolic BDDs**, which efficiently encoded over 8,000 states where explicit methods faltered. Additionally, the **Hybrid ILP+BDD** framework proved robust, effectively leveraging reachability oracles to eliminate spurious solutions during deadlock detection and optimization.

5.2 Further Discussion

5.2.1 Limitations

While being effective, the current tool has some disadvantages:

- **Variable Ordering:** We used a static interleaved ordering (x_i, y_i) . Implementing dynamic variable reordering (sifting) could further reduce BDD sizes for unstructured nets.
- **Extension to k-bounded Nets:** The current Boolean encoding is limited to 1-safe nets. To support general k -bounded nets, we could extend the system using Multi-valued Decision Diagrams (MDDs) or bit-blasting techniques (allocating $\lceil \log_2(k + 1) \rceil$ bits per place).



References

- [1] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, 7th Edition, McGraw-Hill, 2012.
- [2] GeeksforGeeks, *Travelling Salesman Problem using Dynamic Programming*, 26 Nov. 2024. Available at: <https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/>
- [3] W3Schools, *The Traveling Salesman Problem*, Available at: https://www.w3schools.com/dsa/dsa_ref_traveling_salesman.php