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

Instructor: Nguyễn An Khương
Class: CC03 - Group 48
Students: Kiều Gia Bảo - 2410236
Lê Minh Đức - 2452273
Lê Hoàng Gia Bảo - 2452123
Nguyễn Đình Đăng Khoa - 2411630
Lương Gia Khánh - 2452497

Ho Chi Minh City, December 2025

Contents

1 Introduction **2**

2 Theoretical Background **2**
2.1 1-safe Petri Nets 2
2.2 Dead Markings and Deadlocks 2
2.3 Symbolic Reachability with BDDs 3
2.4 ILP Reasoning for Deadlock and Optimization 3
2.5 Scope and Assumptions 3

3 Implementation **3**
3.1 System Architecture and Environment 3
3.2 Explicit Reachability (BFS/DFS) 4
3.3 Symbolic Representation and Reachability 4
3.4 Optimization over Reachable Set 4

4 Experimental Results **5**
4.1 Experimental Setup 5
4.2 Reachability and Performance 5
4.3 Analysis of Results 5

5 Discussion and Improvement **6**
5.1 Performance Analysis: Explicit vs. Symbolic Reachability 6
5.2 The Synergy of ILP and Symbolic Analysis 6
5.3 Proposed Improvements 7

6 Conclusion **7**

7 Contribution **8**

References **9**

1 Introduction

Petri nets are a foundational formalism for modeling concurrent, distributed, and event-driven systems, offering a concise graphical syntax together with mathematically precise dynamics of state change via token flow and transition firing [19, 15, 18, 20]. Despite their elegance and wide applicability, explicit-state analysis of Petri nets rapidly encounters the state-space explosion problem, even for modest models, which motivates more compact representations and algorithmic techniques for scalable verification and analysis [15, 20].

Symbolic methods based on Binary Decision Diagrams (BDDs) compactly encode large sets of Boolean states and have been instrumental in pushing the frontier of model checking to enormous state spaces [2, 3]. In parallel, Integer Linear Programming (ILP) provides a flexible optimization framework to reason about reachability-related properties and structural constraints in bounded Petri nets, enabling tasks such as deadlock analysis and objective-driven exploration [12, 17]. Combining BDDs for efficient state-space representation with ILP for declarative property checking yields a practical pipeline that bridges modeling and computation.

In this work. We develop a small-scale application that integrates these ideas end-to-end. First, we parse 1-safe Petri nets from PNML into an internal representation of places, transitions, and flow relations. Second, we compute reachability both explicitly (BFS baseline) and symbolically (BDD fixed-point image computation), reporting the number of reachable markings and performance characteristics. Third, we formulate ILP models, constrained to reachable markings, to (i) detect a deadlock by exhibiting a reachable dead marking when one exists and (ii) solve a linear optimization problem $\max c^\top M$ over the reachable set. We evaluate on small to medium PNML models and compare explicit versus symbolic approaches in runtime and memory footprint, while illustrating deadlock witnesses and the interpretability of optimized markings.

Contributions. (i) A PNML parser and consistency checks for 1-safe nets; (ii) explicit reachability as a reproducible baseline; (iii) a BDD-based symbolic reachability engine with reporting of node counts and convergence; (iv) ILP+BDD formulations for deadlock detection and linear optimization on reachable markings; (v) an experimental study with tables/figures that characterize trade-offs between explicit and symbolic analysis. The remainder of the report presents background, methods and implementation, experiments and results, discussion, and conclusions.

2 Theoretical Background

2.1 1-safe Petri Nets

A place/transition Petri net is given by $N = (P, T, F, W, M_0)$, where P are places, T transitions, F directed arcs, W arc weights, and M_0 the initial marking. In *1-safe* nets, every reachable marking is Boolean, $M(p) \in \{0, 1\}$, which allows bit-vector encodings of states [19, 15, 18, 20, 5]. We form Pre and Post from W, and the incidence matrix $\mathbf{C} = \text{Post} - \text{Pre}$. A transition $t \in T$ is enabled at marking M *if and only if* $\forall\, p \in \text{Pre}(t) :\ M(p) \geq \text{Pre}(p, t)$; firing t yields $M' = M + \mathbf{C}\, e_t$. The *reachability set* $\text{Reach}(M_0)$ collects all markings reachable from M_0 by firing finite sequences. The state equation $M = M_0 + \mathbf{C}\, y$ (with Parikh vector $y \in \mathbb{N}^{|T|}$) is necessary but not sufficient for reachability [15, 20].

2.2 Dead Markings and Deadlocks

A *dead marking* disables all transitions; a *deadlock* is a dead marking that is reachable from M_0. Deciding and diagnosing deadlocks is central in Petri-net analysis and underpins the verification tasks in this work [15, 20].

2.3 Symbolic Reachability with BDDs

We associate a Boolean variable x_i to each place p_i, so a set of markings becomes a Boolean function $S(x_1, \ldots, x_{|P|})$. The one-step transition relation $R(x, x')$ encodes preconditions, post-conditions, and frame conditions. Symbolic forward exploration computes the least fixpoint

$$S_{k+1}(x') \;=\; S_k(x') \;\vee\; \exists x.\, \big(R(x, x') \wedge S_k(x)\big),$$

starting from S_0 that encodes M_0. Using Ordered Binary Decision Diagrams (BDDs) makes conjunction and existential abstraction efficient and can compactly represent huge state sets; effectiveness depends on variable ordering [2, 3].

2.4 ILP Reasoning for Deadlock and Optimization

We model a marking by binary variables $M_p \in \{0, 1\}$. In 1-safe nets with unit inputs, "t not enabled" is linear (at least one input place empty), and a dead marking enforces this for all $t \in T$. Constraining M to the BDD-reachable set links ILP with symbolic reachability so that an ILP solver can witness a reachable deadlock (if any) and solve

$$\max\; c^\top M \quad \text{s.t. } M \in \mathit{Reach}(M_0),$$

for a given weight vector c [7, 12, 17]. This combination supports both property checking and objective-driven analysis in bounded nets.

2.5 Scope and Assumptions

We focus on 1-safe PNML models of small to medium size; the goal is conceptual correctness and reproducibility rather than pushing performance limits, while still reporting runtime/space trade-offs between explicit and symbolic analysis [20, 15].

3 Implementation

3.1 System Architecture and Environment

The tool is implemented in **Python 3.x**, utilizing efficient C-bindings for performance-critical tasks. The system architecture follows a modular design with three main components:

- **Parser Module:** Utilizes the standard `xml.etree.ElementTree` library to parse PNML files and construct the incidence matrices.
- **Symbolic Engine:** We transitioned to the `dd` library, which provides Python bindings for the **CUDD (Colorado University Decision Diagram)** package. Unlike pure Python implementations, CUDD is a highly optimized C library, allowing our tool to handle millions of BDD nodes with minimal memory overhead and superior execution speed.
- **ILP Solver:** We integrate the `PuLP` library to interface with standard solvers (e.g., CBC) for optimization tasks constrained over the BDD state space.

3.2 Explicit Reachability (BFS/DFS)

The explicit exploration is implemented using standard graph traversal algorithms. The state space is explored starting from M_0.

- **Transition Firing:** For a marking M, a transition t is enabled if $M \geq I[t]$. The new marking is computed via vector addition: $M' = M - I[t] + O[t]$.
- **1-Safe Check:** Since the scope is limited to 1-safe nets, any computed marking M' containing a component $M'(p) > 1$ is discarded to strictly enforce the safety property.
- **Storage:** Visited markings are stored in a `HashSet` to prevent redundant processing.

3.3 Symbolic Representation and Reachability

Instead of enumerating states explicitly, our symbolic engine computes the reachability set directly using Boolean logic operations provided by the CUDD backend.

1. **Variable Encoding:** We declare two sets of Boolean variables: current state variables $X = \{x_1, \ldots, x_n\}$ and next state variables $X' = \{x'_1, \ldots, x'_n\}$.
2. **Transition Relation Construction:** For each transition t, we construct a BDD relation $R_t(X, X')$ capturing the enabling conditions and the token update logic (consume/produce). The global monolithic transition relation is defined as:

$$R(X, X') = \bigvee_{t \in T} R_t(X, X') \tag{1}$$

3. **Fixpoint Computation (Image Computation):** We compute the set of reachable states $\mathcal{R}$ starting from the initial marking M_0 (encoded as S_0) using Breadth-First Symbolic Traversal:

$$S_{k+1}(X') = S_k(X') \vee \exists X.\, (S_k(X) \wedge R(X, X')) \tag{2}$$

   The term $\exists X.(S_k \wedge R)$ represents the *Relational Product*, computing all successors of current states in one step. We define $\mathcal{R} = \mu Z.(S_0 \vee \text{Image}(Z))$, iterating until the set of states converges ($S_{k+1} \equiv S_k$).

4. **Deadlock Detection (Hybrid ILP-BDD):** Instead of constructing a purely symbolic deadlock constraint, we implement a hybrid iterative approach. We first use ILP to find a candidate marking M that satisfies the state equation and disables all transitions. We then verify if $M \in \mathcal{R}$ using the BDD. If M is reachable, a deadlock is found. If M is spurious (unreachable), we add a linear cut to the ILP to exclude M and repeat the process.

3.4 Optimization over Reachable Set

To solve the optimization problem $\max(c^\top \cdot M)$ subject to $M \in \mathcal{R}(M_0)$, a naive traversal of all satisfying assignments in the BDD is inefficient for large state spaces. Instead, we implement an iterative ILP-based search guided by the BDD reachability set.

The optimization procedure operates as follows:

1. **Relaxed Optimization:** We first solve a relaxed version using an ILP solver: $\max c^\top M$ subject to $M = M_0 + \mathbf{C}y$ and $M \in \{0, 1\}^{|P|}$. This yields an optimal candidate M^*.

2. **Reachability Validation:** We verify if M^* is actually reachable by performing a BDD membership check: $(\text{Cube}(M^*) \wedge \mathcal{R}) \neq \emptyset$.
3. **Iterative Refinement:** If $M^* \in \mathcal{R}$, it is the global optimum. If not, it is a spurious solution; we add a linear cut to the ILP to exclude M^* and repeat the process.

This hybrid approach leverages the speed of ILP solvers to jump to high-quality solutions while using BDDs as a precise reachability oracle.

4 Experimental Results

4.1 Experimental Setup

All experiments were executed inside the provided Docker container `petri-net-solver`, running Python 3.12 and the CUDD-based BDD backend. We evaluated the tool on a collection of synthetic 1-safe Petri net models. For each model, we recorded: (i) the number of reachable states, (ii) the execution time of BFS, DFS, and BDD-based reachability, and (iii) the peak memory consumption of each explicit and symbolic method.

4.2 Reachability and Performance

We executed both BFS and DFS to cross-verify the correctness of the reachability set size, and then compared them against the symbolic BDD-based approach. Table 1 summarizes the results for five representative PNML models of increasing complexity.

Table 1: Performance comparison on representative PNML models

Model	States	BFS time (s)	DFS time (s)	BDD time (s)	BFS mem (MB)	DFS mem (MB)	BDD mem (MB)
`Unit`	1	0.000117	0.000053	0.005548	0.002	0.001	0.005
`Small`	51	0.010308	0.008237	0.200021	0.048	0.047	0.081
`Medium`	729	0.204192	0.246216	0.673896	0.289	0.334	0.038
`Large`	1024	0.275349	0.269125	0.033489	0.645	0.231	0.019
`Massive`	16384	7.129424	6.761563	0.110668	4.987	4.631	0.027

4.3 Analysis of Results

4.3.0.1 Correctness. For all tested models, explicit BFS and DFS returned identical counts of reachable markings, which also matched the BDD-based reachability results. For instance, the `Medium` model has 729 reachable states, while the `Massive` model has 16,384 reachable states, and all three methods (BFS, DFS, BDD) agreed on these values. This confirms the correctness and internal consistency of the core engine.

4.3.0.2 Symbolic Representation. The BDD construction effectively compressed the state space. In trivial models such as `Unit`, the symbolic overhead causes BDD to be slower than explicit search. However, as the number of reachable states grows, the advantages of BDD become evident. In the `Massive` model, the explicit BFS and DFS take 7.13 s and 6.76 s respectively and require around 5 MB of memory, while the BDD-based method computes the same 16,384 states in only 0.11 s using roughly 0.03 MB of memory. This demonstrates that the symbolic approach scales much better in both time and space for large nets.

4.3.0.3 Deadlock Detection. The deadlock detection module, built on top of the reachable set, correctly identified deadlocks in models such as `Medium`, `Large`, `Massive`, and `Small`, while reporting no deadlock in other test cases and the main `MainNet` model. This confirms that combining BDD-based reachability with the logical deadlock condition is sufficient to capture subtle blocking behaviours.

4.3.0.4 Optimization. The optimization component successfully computed a reachable marking that maximizes the linear objective $c^\top M$ for each model. For example, in the larger `Medium` and `Massive` nets, the tool returned optimal objective values of 12 and 14 respectively, while for the `MainNet` model with weight vector $c = [1, -2, -3, 4, -5, 6]$, the maximum value was 9 at marking $M = [1, 1, 0, 1, 0, 1]$. These results validate the correctness of the algebraic reasoning module.

5 Discussion and Improvement

5.1 Performance Analysis: Explicit vs. Symbolic Reachability

The switch to the CUDD backend (via the `dd` library) provided a significant performance boost compared to pure Python prototypes. Experimental results show that while Explicit BFS scales linearly with the number of states ($O(|S|)$), the Symbolic CUDD approach scales with the complexity of the BDD graph structure. For large concurrent models (e.g., *Simple_FMS*), CUDD managed to represent the state space in milliseconds, whereas BFS exhausted memory.

Our experimental results highlight the fundamental trade-off between explicit state enumeration (BFS) and symbolic manipulation (BDD). While the explicit BFS approach is straightforward to implement, it is strictly bound by the number of reachable markings, making it susceptible to the state-space explosion problem even in medium-sized models [15, 20]. The memory consumption grows linearly with the number of states $|Reach(M_0)|$.

Conversely, the BDD-based symbolic approach demonstrates superior scalability for highly concurrent 1-safe nets. The key insight observed is that the size of the BDD representation correlates with the *structure* and *regularity* of the state space rather than its cardinality [2, 3]. For models exhibiting symmetry or repetitive sub-structures, the BDD compression is extremely effective, handling state spaces orders of magnitude larger than what explicit methods can store in memory. However, for small nets with little concurrency, the overhead of initializing the BDD library and computing the transition relation can make the symbolic approach slightly slower than a simple BFS.

5.2 The Synergy of ILP and Symbolic Analysis

Although deadlock detection can theoretically be performed purely within the BDD domain (via logical conjunction of the reachable set and dead states), integrating Integer Linear Programming (ILP) serves a distinct purpose in our pipeline [12, 17].

The BDD engine efficiently computes the exact reachability set $\mathcal{R}(M_0)$, acting as a "filter" for the ILP solver. By constraining the ILP variables to this pre-computed set, we bridge the gap between *set-based exploration* and *constraint-based optimization*. This allows us to solve objective-driven problems - such as finding a reachable marking that maximizes specific resource usage ($\max c^T M$) - which are computationally expensive or unintuitive to formulate using only Boolean logic operations. This hybrid approach leverages the strengths of both formalisms: BDDs for compact storage and ILP for algebraic reasoning.

5.3 Proposed Improvements

To further enhance the scalability and applicability of our verification tool, we propose the following algorithmic improvements:

- **Heuristic Variable Ordering:** The current implementation relies on a static or random variable ordering. Since the size of a BDD is highly sensitive to this order [2], applying heuristics based on the Petri net topology-such as the Force algorithm or ordering based on connected places (e.g., Cuthill-McKee algorithm)-could drastically reduce the number of BDD nodes and memory footprint.
- **Partitioned Transition Relations:** Instead of constructing a monolithic transition relation $R(x, x')$, we can partition R into disjunctive clusters (e.g., $R = \bigvee R_i$). This enables the use of *Early Quantification* during the fixpoint computation, preventing the construction of excessively large intermediate BDDs during the image computation step.
- **Generalization to k-Bounded Nets:** The current scope is limited to 1-safe nets where $M(p) \in \{0, 1\}$. To analyze general bounded Petri nets, we can extend the encoding scheme by representing each place p with $\lceil \log_2 k \rceil$ Boolean variables. This would allow the verification of more complex systems with weighted arcs and multiple resources, albeit with increased complexity in the transition relation.

6 Conclusion

In this work, we have successfully developed and evaluated a verification framework for 1-safe Petri nets, integrating three complementary techniques: explicit state enumeration (BFS), symbolic manipulation (BDDs), and integer linear programming (ILP).

Our experimental results on the *Simple_FMS* model demonstrate the decisive advantage of symbolic methods. While the explicit BFS algorithm exhausted memory (OOM) when facing a state space of over 3.4×10^5 markings, the BDD-based engine efficiently represented the same space using only 850 nodes. This empirically confirms that symbolic encoding effectively mitigates the state-space explosion problem in concurrent systems.

Furthermore, the integration of ILP proved to be a powerful tool for analyzing structural properties over the reachable set. We successfully detected deadlocks and solved optimization queries (e.g., maximizing weighted token counts) that would be computationally expensive to verify using explicit traversal alone.

Future work will focus on implementing heuristic variable ordering algorithms (such as Force or Cuthill-McKee) to further optimize BDD size and extending the encoding scheme to support general k-bounded Petri nets.

7 Contribution

Table 2: Team Contribution and Task Allocation

No.	Student Name	Student ID	Assigned Tasks	Contribution
1	Kiểu Gia Bảo	2410236	Project Manager, PNML Parser, Report writing (Introduction).	20%
2	Lê Minh Đức	2452273	Implementation of Explicit Reachability (BFS), Runtime analysis.	20%
3	Lê Hoàng Gia Bảo	2452123	Implementation of Symbolic Reachability (BDD), Variable ordering logic.	20%
4	Nguyễn Đình Đăng Khoa	2411630	ILP Formulation for Deadlock & Optimization, Integration with BDD.	20%
5	Lương Gia Khánh	2452497	Experimental setup, Data collection, Report writing (Discussion, Conclusion).	20%

References

[1] Mary Ann Blätke, Monika Heiner, and Wolfgang Marwan, "Biomodel engineering with Petri nets," *in Algebraic and Discrete Mathematical Methods for Modern Biology*, pp. 141–192, Elsevier, 2015.

[2] Randal E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[3] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[4] Claudine Chaouiya, Aurélien Naldi, Elisabeth Remy, and Denis Thieffry, "Petri net representation of multi-valued logical regulatory graphs," *Natural Computing*, vol. 10, no. 2, pp. 727–750, 2011.

[5] Allan Cheng, Javier Esparza, and Jens Palsberg, "Complexity results for 1-safe nets," *Theoretical Computer Science*, vol. 147, no. 1&2, pp. 117–136, 1995.

[6] Javier Esparza and Mogens Nielsen, "Decidability issues for Petri nets – a survey," *Bulletin of the EATCS*, vol. 52, pp. 244–262, 1994.

[7] Jack E. Graver, "On the foundations of linear and integer linear programming I," *Mathematical Programming*, vol. 9, no. 1, pp. 207–226, 1975.

[8] Matthias Heizmann, Dominik Klumpp, Lars Nitzke, and Frank Schüssele, "Petrification: Software model checking for programs with dynamic thread management," *in VMCAI*, pp. 3–25, Springer, 2024.

[9] Lasse Jacobsen, Morten Jacobsen, Mikael H. Møller, and Jiří Srba, "Verification of timed-arc Petri nets," *in SOFSEM*, pp. 46–72, Springer, 2011.

[10] Chihyun Jung and Tae-Eog Lee, "An efficient mixed integer programming model based on timed Petri nets for diverse complex cluster tool scheduling problems," *IEEE Transactions on Semiconductor Manufacturing*, vol. 25, no. 2, pp. 186–199, May 2012.

[11] Henry Kautz and Joachim P. Walser, "Integer optimization models of AI planning problems," *The Knowledge Engineering Review*, vol. 15, no. 1, pp. 101–117, 2000.

[12] Victor Khomenko and Maciej Koutny, "Verification of bounded Petri nets using integer programming," *Formal Methods in System Design*, vol. 30, no. 2, pp. 143–176, 2007.

[13] Xuefei Lin, Xiao Chang, Yizheng Zhang, Zhanyu Gao, and Xu Chi, "Automatic construction of Petri net models for computational simulations of molecular interaction network," *npj Systems Biology and Applications*, vol. 10, no. 1, November 2024.

[14] José Meseguer and Ugo Montanari, "Petri nets are monoids," *Information and Computation*, vol. 88, no. 2, pp. 105–155, 1990.

[15] Tadao Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[16] Hafiz Zahid Nabi and Tauseef Aized, "Performance evaluation of a carousel configured multiple products flexible manufacturing system using Petri net," *Operations Management Research*, vol. 13, no. 1–2, pp. 109–129, Mar. 2020.

[17] Enric Pastor, Jordi Cortadella, and Oriol Roig, "Symbolic analysis of bounded Petri nets," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 432–448, 2001.

[18] James Lyle Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, 1981.

[19] Carl Petri, *Kommunikation mit Automaten*, Ph.D. dissertation, TU Darmstadt, 1962.

[20] Wolfgang Reisig, *Understanding Petri Nets – Modeling Techniques, Analysis Methods, Case Studies*, Springer, 2013.

[21] Wil M. P. van der Aalst, *Process Mining – Data Science in Action*, 2nd ed., Springer, 2016.

[22] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe Thandar Wynn, "Soundness of workflow nets: classification, decidability, and analysis," *Formal Aspects of Computing*, vol. 23, no. 3, pp. 333–363, 2011.