Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.082012

ARTICLE

Deadlock Verification of the Room Allocation Workflow in an EHR System Using UPPAAL

Acep Taryana[1,2], **Dieky Adzkiya**[1,*], **Muhammad Syifa'ul Mufid**[1] **and Imam Mukhlash**[1]

[1]Department of Mathematics, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia
[2]Department of Electrical Engineering, Universitas Jenderal Soedirman, Purwokerto, Indonesia

*Corresponding Author: Dieky Adzkiya. Email: dieky@its.ac.id

Received: 11 March 2026; Accepted: 16 July 2026; Published: 13 August 2026

ABSTRACT: A systematic approach is necessary to address concurrency issues, timing violations, and logical safety breaches in healthcare systems, as traditional empirical testing often fails to uncover non-deterministic flaws that can endanger patient safety. This study proposes a rigorous formal verification methodology based on a triple-constraint framework to prove system reliability prior to the implementation phase. The patient room allocation workflow was modeled as a network of Timed Automata (TA) within the UPPAAL model checker, allowing for the formal evaluation of concurrent processes and shared resources. Our methodology follows three core phases: (1) Formalization, where critical requirements are translated into Timed Computation Tree Logic (TCTL) properties targeting three pillars: temporal bounds, safety invariants (e.g., category matching), and deadlock-freedom guarantees (e.g., $A\,\Box\,\neg deadlock$); (2) Iterative Formal Refinement, a systematic algorithm that utilizes UPPAAL to identify and eliminate design flaws by dynamically refining automata components—such as location invariants, transition guards, and synchronization channels—based on counter-example traces; and (3) Artifact Generation, where the verified formal model is systematically translated into reliable software engineering blueprints. The iterative refinement process culminated in a final TA model mathematically proven to be free from deadlocks, timelocks, and unsafe states, which was successfully scaled under high queueing loads via Statistical Model Checking (SMC). This verified mathematical blueprint was then synthesized into Unified Modeling Language (UML) design artifacts, specifically class, sequence, and state machine diagrams. By integrating a triple-constraint formal verification at the design stage, this methodology proactively prevents concurrency failures, bridging mathematical guarantees with practical software engineering. Furthermore, the comparative performance evaluation reveals that while exact state-space verification triggers Out-of-Memory (OOM) failures at merely $N \geq 6$ concurrent processes due to state-space explosion, the proposed TA-SMC framework successfully neutralizes this limitation. By bridging the gap between lightweight simulation and formal rigor, our approach smoothly scales up to $N = 25$ concurrent processes, maintaining a strictly constant peak memory footprint of ≈14 MB, while simultaneously guaranteeing ≥99% statistical confidence for critical safety invariants. These quantitative results firmly establish the proposed methodology as a highly robust and practically viable solution for evaluating complex, resource-contended clinical workflows.

KEYWORDS: Formal verification; timed automata; model checking; software design; electronic health record; deadlock prevention; triple-constraint

1 Introduction

Electronic Health Record (EHR) systems are critical healthcare infrastructure where software failures directly compromise patient safety [1–3]. These systems face significant concurrency challenges, particularly resource allocation conflicts in workflows like patient room assignment, which can trigger system deadlocks [4–6]. A deadlock occurs when processes permanently block each other while waiting for resources, satisfying the four Coffman Conditions [7]. Modern application layers using Object-Relational Mapping (ORM) further obscure these database-level locks, causing elusive failures [8]. Traditional testing is inadequate for detecting these "Heisenbugs" [9] due to non-determinism and the combinatorial explosion of execution states [10–12].

To overcome the fundamental limitations of traditional testing, the research area of formal verification employs rigorous mathematical techniques to prove system correctness. While advanced methods such as static analysis [13] and dynamic test generation [14] exist, they focus on implementation-level artifacts rather than on architectural design. Similarly, research into deadlock resolution through non-classical logics focuses on recovery rather than prevention [5]. Previous efforts to bridge the design gap, such as the use of Coloured Petri Nets (CPN) [15] and architectural simulations [16], primarily address untimed logical data flows. Consequently, these existing techniques either intervene too late in the development process or fail to account for the strict temporal constraints inherent in safety-critical healthcare workflows. Therefore, a prevention-by-design strategy [5,17,18] that identifies deadlocks at the fundamental design stage while explicitly capturing real-time behavior is essential.

To address this need, one must evaluate broader formal verification approaches. The model abstraction approach is used to simplify a model with large states to a smaller model which is invariant with respect to some properties [19,20], while the idea of verification based on Satisfiability Modulo Theories (SMT) is transforming the original verification problem into a satisfiability problem to be solved using an SMT solver [21,22]. While both approaches are highly effective, applying them directly as the primary gateway for real-time concurrent systems requires complex, manual encoding of continuous time and native concurrency semantics. Furthermore, probabilistic verification frameworks focus on the verification of stochastic models, providing probability-based reliability rather than the absolute deterministic safety guarantees required by safety-critical systems [18,23,24]. Because the problem considered in this paper is a hard real-time verification problem that demands strict, deterministic deadlock prevention, directly utilizing those three approaches as the initial modeling framework is not the most suitable solution. Consequently, we utilize the UPPAAL tool to verify real-time systems modeled as networks of Timed Automata (TA) [25] against Timed Computation Tree Logic (TCTL) properties [24,26], as it natively provides intuitive semantics for continuous clocks, guards, and synchronization.

Translating semi-formal design models like Unified Modeling Language (UML) into formal verification tools often introduces a semantic gap, where critical system behaviors become ambiguous or lost in translation [27]. Unlike related work translating UML to UPPAAL, our methodology uses a verified UPPAAL model to generate reliable UML artifacts (UPPAAL → UML), bridging formal mathematical guarantees with practical software engineering. Specifically, we aim to: (1) model the patient room allocation workflow as a network of TA; (2) iteratively utilize the UPPAAL model checker to enforce a triple-constraint specification; (3) produce a formally verified model mathematically proven to satisfy all critical temporal, safety, and liveness properties; and (4) demonstrate the systematic translation of this mathematical model into standard UML artifacts. To realize this framework, the subsequent sections are structured as follows. Section 2 details the Materials and Methods, establishing the mathematical foundation and constructing the Timed Automata models for the patient room allocation workflow. Following this, Section 3 presents the iterative verification results, confirming the elimination of deadlocks and the satisfaction of safety constraints, and directly

translates these mathematical proofs into actionable software engineering blueprints through systematic UML artifact generation. Section 4 provides a comprehensive discussion on the architectural implications, comparative analysis, and practical deployment feasibility of the proposed framework. Finally, Section 5 provides concluding remarks and future outlooks.

2 Material and Methods

This research employs a formal verification methodology using the model checking approach to prove the system's design reliability prior to the implementation phase. That is, before the development team writes implementation code or even creates detailed design diagrams, the core logic of the system's workflow is modeled and analyzed first. The design and development stages in the Software Development Life Cycle (SDLC), in Fig. 1, will only formally begin after the model (M) is proven to satisfy all critical specifications (ϕ). This approach ensures that the design to be implemented already has mathematical guarantees regarding reliability properties, thereby reducing the risk of discovering fundamental design flaws in the later stages of development.

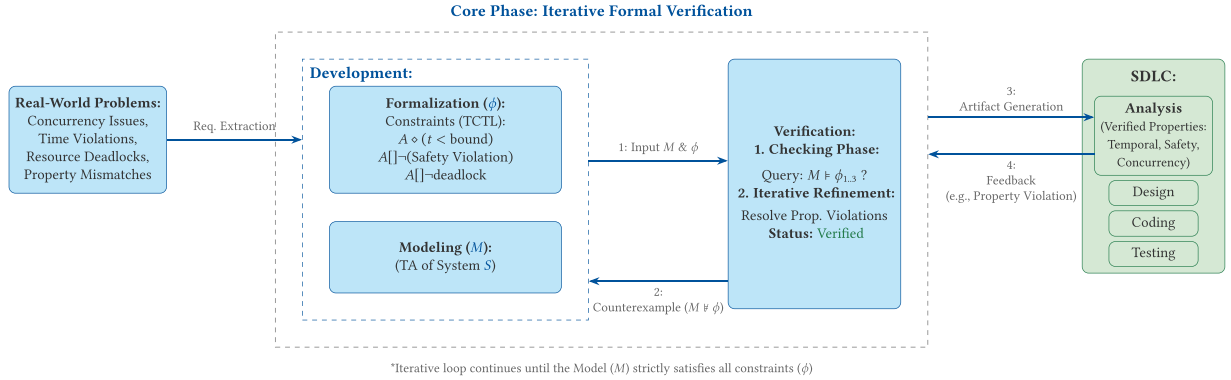


Figure 1: Formal verification framework for an EHR system within the context of problems and the software development life cycle.

The proposed methodology is specifically engineered to address a triple-constraint safety requirement: (1) Temporal Constraints, ensuring all critical operations meet strict timing upper-bounds; (2) Safety Invariants, preventing illegal system states such as category mismatches; and (3) Deadlock/Timelock Prevention, guaranteeing continuous system liveness in concurrent environments. These three pillars form the basis of our formal specifications (ϕ) and are verified across all system iterations.

The research process, conducted using the UPPAAL tool, consists of four main stages as shown in Fig. 1. The process begins with "Formalization", where system requirements are translated into a formal property, ϕ, alongside the concurrent "Modeling" stage, which involves creating a TA model, M, of the system's behavior. The third stage is "Formal Verification", which uses UPPAAL to check if the model satisfies the property ($M \models \phi$). If the property is not satisfied, a feedback loop initiates a refinement of the model (flow 2), enabling iterative system improvement. Once the model is proven correct, the final stage (flow 3) involves the synthesis of UML artifacts, where the verified TA is translated into *class*, *sequence*, and *state* diagrams to serve as a reliable implementation blueprint.

The execution of this research was supported by UPPAAL version 5.1.0 beta4 for modeling and verification, our custom-developed tool—PRISMA (Purwokerto Surabaya Integrated Software Engineering-Mathematic-Automata) Translator—for translating verified models into UML artifacts, and PlantUML for design visualization. The core process of this methodology can be summarized in the iterative refinement

algorithm, as detailed in Algorithm 1. This algorithm enhances the iterative verification procedure established in the precedent LTL model checking study by Taryana et al. [28]. Furthermore, it leverages the framework by Famelis et al. [29] as the mathematical justification for this formal refinement loop, operationalizing concretization as the systematic transformation of an abstract model into a precise architecture. While their fundamental loop of verifying a model and refining it based on counterexamples remains the same, this enhanced approach utilizes TA to systematically incorporate and verify specific timing constraints—a critical step not addressed in their original logical framework. This gradual model refinement strategy, also known as the abstraction and refinement approach, has proven effective in verifying complex systems by explicitly integrating temporal parameters during the modeling phase, similar to the methodology applied in contemporary telerehabilitation system verifications [27].

This method involves modeling TA with UPPAAL to detect deadlocks, verifying TCTL properties like $A\,\square\,\neg$deadlock, and refining the model through an iterative refinement algorithm. This approach considers time constraints and shared resources, a methodology well-described in the UPPAAL tutorial [30]. Data structures like arrays are used to track resource status (e.g., locked or available), while clocks support time constraints. This approach is common in modeling shared resource problems to minimize the risk of deadlock [24].

2.1 Timed Automata Formalism and Notation

Before detailing the iterative refinement procedure, it is essential to establish the fundamental modeling formalism of the TA [31]. The core operations within our proposed algorithm systematically evaluate and manipulate the mathematical components of this formalism to ensure system safety and liveness.

Formally, a TA system is modeled as a tuple $M = (L, l_0, C, A, E, I)$, where:

- L is a finite set of locations (states) representing the system's discrete configurations.
- $l_0 \in L$ is the designated initial location.
- C is a finite set of real-valued clocks that advance synchronously to measure the continuous passage of time.
- A is a set of actions or synchronization labels used for communication between concurrent automata via channels.
- $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ defines the set of directed edges (transitions) between locations. A single transition $e \in E$ consists of a source location, an action label, a guard $g \in \mathcal{B}(C)$ (a logical clock constraint that must be true to enable the transition), a set of clocks to reset upon transition, and a target location.
- $I : L \to \mathcal{B}(C)$ is a function mapping each location to a local invariant. An invariant specifies a maximum time bound (e.g., $c \leq 5$) that dictates how long the system is permitted to remain in that specific location before it is forced to transition.

During the verification process, UPPAAL explores the state-space $\mathcal{S}$ of the TA network. A specific execution path is represented as a sequence of transitions, termed a trace, denoted by $\sigma = s_0 \xrightarrow{a_0} s_1 \ldots \xrightarrow{a_{n-1}} s_n$. When checking a set of formal properties Φ (expressed in TCTL), the verifier determines if the state-space satisfies the constraints ($\mathcal{S} \vDash \Phi$). If a violation occurs, the resulting counter-example trace σ is fed back into the refinement loop.

As formulated in Algorithm 1, our methodology directly modifies the components of the tuple M to resolve these violations. For instance, timing violations are mitigated by actively tightening location invariants $I(l)$ and transition guards E, while deadlocks and resource conflicts are resolved by restructuring synchronization channels A and applying priority ordering to transitions.

2.2 Workflow Modeling

Given the complexity of EHR systems, we narrow our formal verification scope to a specific Functional Requirement (FR), denoted as FR-Y (Patient Room Allocation Management), a workflow inherently susceptible to concurrent deadlocks [8]. The system is abstracted as a network of TA in UPPAAL, modeling components like patients and staff as individual automata. Functionally, the system maps patient data (ID, disease category) to available rooms, outputting either a compatible allocation or an unavailability warning. Consequently, this formal abstraction strictly isolates the deterministic Functional Requirements (FRs) of the system's core logic, intentionally abstracting away Non-Functional Requirements (NFRs) such as network latency or hardware failures. To guarantee reliability, this TA network is subjected to a triple-constraint specification, mapping real-world requirements to mathematically verifiable properties via transition guards and state invariants:

- **Temporal Constraints (ϕ_{time})**: Allocation must strictly complete within 5 time units to prevent critical care latency ($A \Diamond (time \leq 5)$).
- **Safety Invariants (ϕ_{safety})**: The model must prevent illegal state transitions, such as incompatible category allocations (e.g., enforcing $cat_{pat} == cat_{room}$).
- **Deadlock Prevention ($\phi_{deadlock}$)**: The system must robustly manage concurrent requests without entering a *deadlock* or *timelock* ($A \Box \neg deadlock$).
- **Resource Exhaustion Handling**: The idealistic assumption of perpetual room availability must be formally validated, ensuring safe patient queueing without violating the aforementioned temporal or liveness bounds.

2.3 Property Specification

To rigorously evaluate the triple-constraint specification ($\Phi = \{\phi_{time}, \phi_{safety}, \phi_{deadlock}\}$), system requirements were formalized using Timed Computation Tree Logic (TCTL) [26]. UPPAAL utilizes a practical subset of TCTL employing path quantifiers (A: for all paths, E: there exists a path) and temporal state operators ($\Box$: always, $\Diamond$: eventually). By expressing requirements in TCTL, we ensure that correctness depends not only on logical event sequences but also on precise timing bounds across all possible execution trees.

The primary constraints are formally specified and verified as follows:

1. Deadlock Prevention ($\phi_{deadlock}$): Ensures the system globally avoids terminal halts, meaning no sequence of concurrent events can lead to a state where no further transitions are possible.

$$A \Box \neg \text{deadlock} \tag{1}$$

2. Safety Invariants (ϕ_{safety}): Prevents illegal logical states. Specifically, it guarantees that every allocated patient is mapped to a room matching their specific disease category constraint.

$$A \Box \ \forall (p \in Patients) \, (Allocated(p) \implies cat_{pat}(p) == cat_{room}(alloc(p))) \tag{2}$$

3. Temporal Constraints (ϕ_{time}): Guarantees strict upper-bound timing for critical operations. It asserts that any patient entering the waiting state will be processed within the 5-time-unit threshold.

$$A \Box \ \forall (p \in Patients) \, (Waiting(p) \implies clock(p) \leq 5) \tag{3}$$

Algorithm 1: Iterative formal refinement for triple-constraint verification

Input:
- Initial TA Network $M = (L, l_0, C, A, E, I)$
- TCTL Property Set $\Phi = \{\phi_{time}, \phi_{safety}, \phi_{deadlock}\}$

Output: Verified Deadlock-free TA Model M^*
Initialize: $M^* \leftarrow M$
while $\exists \phi \in \Phi$ such that $M^* \not\models \phi$ **do**
    Execute *UPPAAL Model Checker* to explore State-Space $\mathcal{S}$
    Evaluate Satisfaction: $\mathcal{S} \models (\phi_{time} \wedge \phi_{safety} \wedge \phi_{deadlock})$
    if Property Violation Detected **then**
        Extract *Counter-example Trace* $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$
        Refine Automata Components:
        if $\sigma \implies$ Timing Violation (ϕ_{time}) **then**
            Update Location Invariants: $I(l) \leftarrow \{c \leq bound \mid c \in C, l \in L_{active}\}$
            Tighten Transition Guards: $g \in \mathcal{B}(C)$ on $e \in E$
        else if $\sigma \implies$ Safety Violation (ϕ_{safety}) **then**
            Enforce Logic Guards: $g_{logic} \leftarrow (\text{cat}_{pat} == \text{cat}_{room})$
            Update Edge Constraints: $E \leftarrow E \cup \{g_{logic}\}$
        else if $\sigma \implies$ Deadlock Violation $(\phi_{deadlock})$ **then**
            Resolve Deadlock/Timelock via *Channel Synchronization A*
            Apply Priority/Order: $e_i \prec e_j$ to break Circular Wait
        end if
        $M^* \leftarrow$ Update Automata Structure
    end if
end while
Return: M^* s.t. all TCTL properties in Φ are satisfied.

2.4 Iterative Verification and Refinement

The UPPAAL model checker was used to systematically explore the entire state-space of the model against the specified properties. This process was iterative: if the verifier found a property violation and produced a counterexample, the model was analyzed to identify the design flaw. The model was then refined and reverified. This cycle was repeated until the model was proven to satisfy all specified properties. Here are the four main iteration steps we performed.

2.4.1 Iteration 1 (Initial Model & Liveness Violation)

Our process began with an intuitive initial model. As depicted in Fig. 2, this iteration consisted of three main automata: *Patient*, *Room*, and a centralized *AllocationSystem*. The UPPAAL verifier extracted a counter-example trace (σ) revealing a liveness violation. Specifically, a deadlock occurred due to a circular dependency between the automata. Following the refinement algorithm, we applied priority ordering ($e_i \prec e_j$) and restructured the synchronization channels A to break the circular wait.

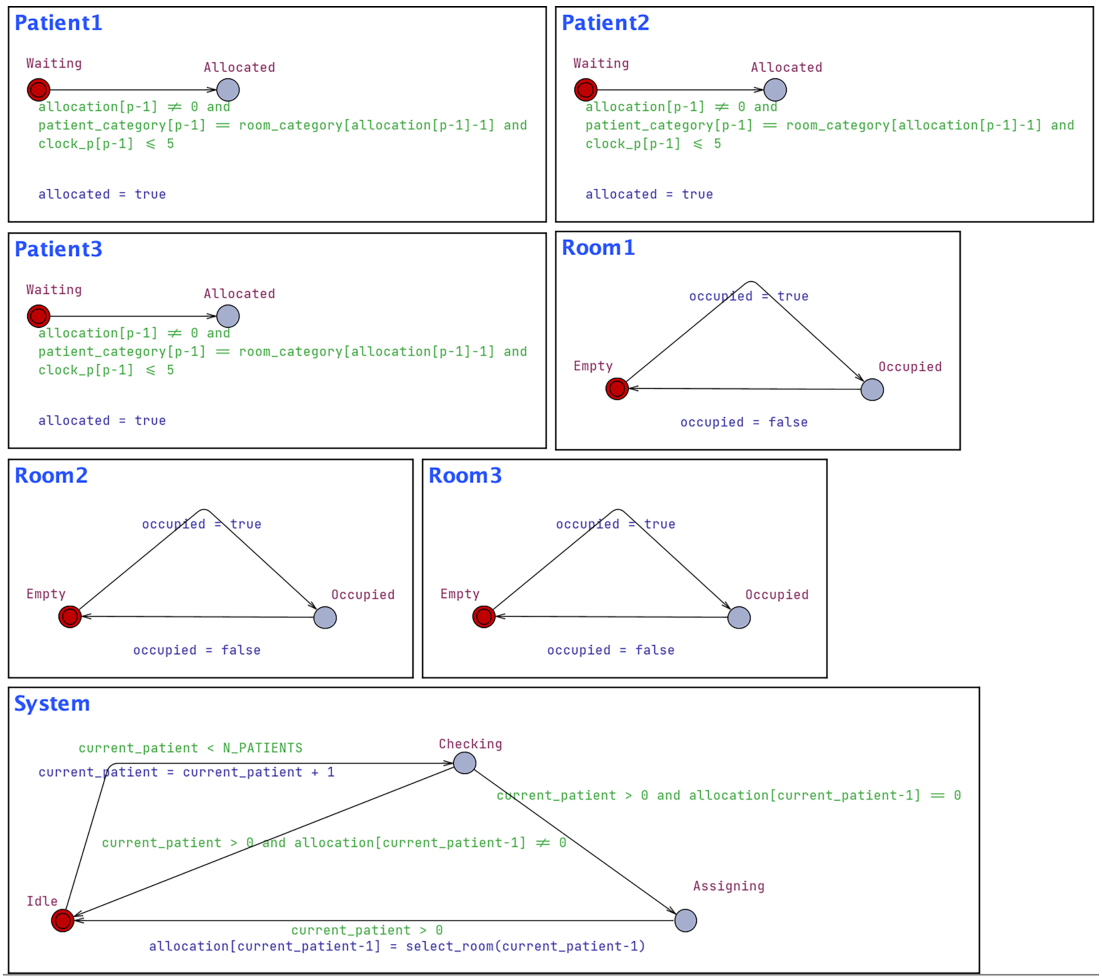


Figure 2: UPPAAL model in iteration 1. A deadlock occurred due to circular dependency.

2.4.2 Iteration 2 (Decomposition & Safety Violation)

The *AllocationSystem* was broken down to better manage discrete categories. However, evaluating this new structure produced a new trace (σ) indicating a safety violation ($\neg\phi_{safety}$). The model erroneously allowed a patient to be allocated to a room with an incompatible category. To resolve this, we enforced strict logic guards ($g_{logic} \leftarrow cat_{pat} == cat_{room}$) and updated the edge constraints E. Visually, as shown in Fig. 3, this refinement is reflected by the addition of new conditional guard labels (typically depicted in green text on the transitions) that strictly evaluate category matching before permitting the allocation step.

2.4.3 Iteration 3 (Resource Release & Timing Violation)

In this iteration, logic for releasing a room was added to handle resource exhaustion. However, the verifier revealed a timing violation ($\neg\phi_{time}$) where patients were trapped in the waiting location for more than 5 time units when all rooms were occupied. Guided by the algorithm, we actively updated the location invariants ($I(l) \leftarrow c \leq 5$) and tightened the transition guards ($g \in \mathcal{B}(C)$). As depicted in Fig. 4, this structural update is visually evident through the introduction of a local clock variable c, the addition of the invariant $c \leq 5$ (depicted in purple) inside the *Waiting* state, and the corresponding clock reset $c := 0$ on its incoming edge.

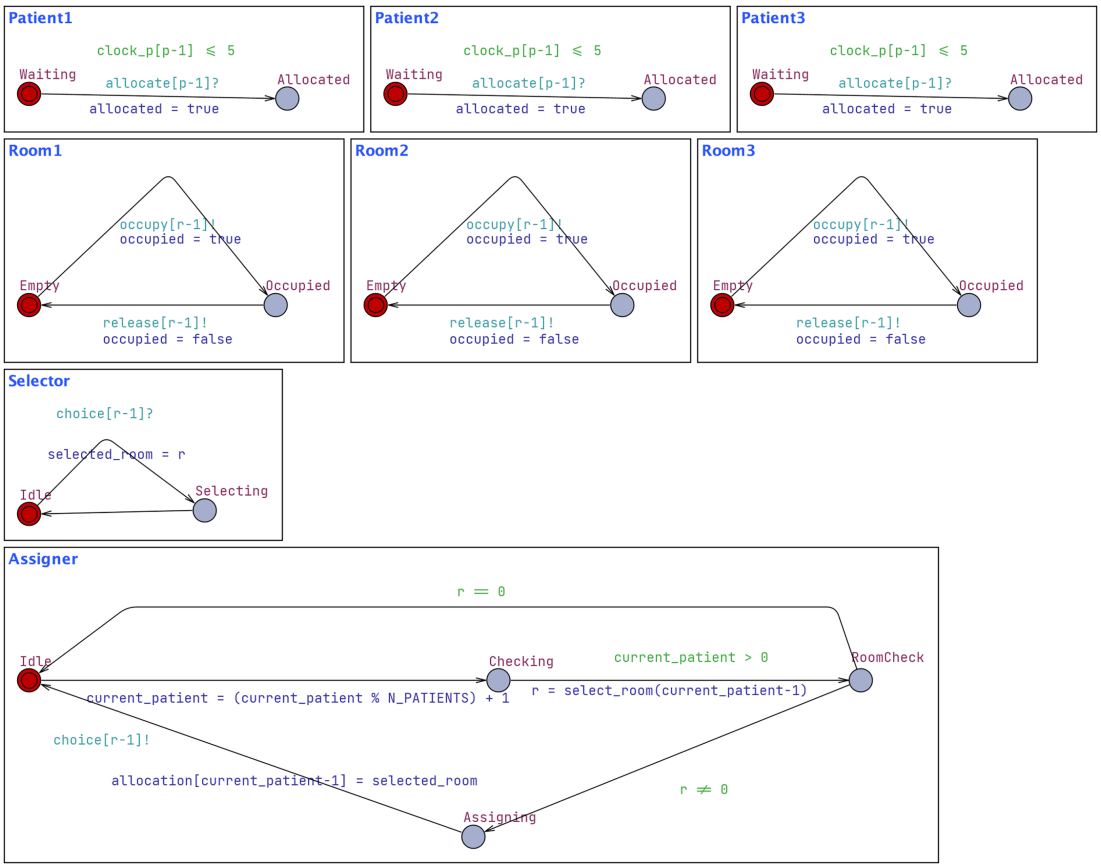


Figure 3: UPPAAL model in iteration 2. A safety violation emerged, resolved by adding explicit category-matching guards on the transitions.

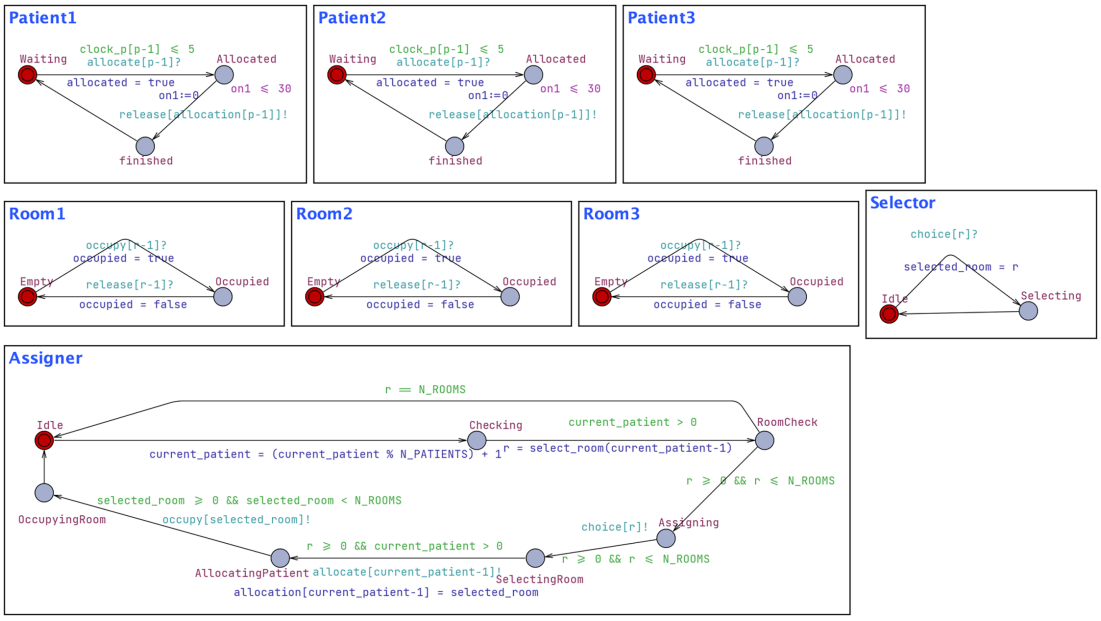


Figure 4: UPPAAL model in iteration 3. A timing violation occurred, resolved by adding clock invariants and reset actions to the waiting state.

2.4.4 Iteration 4 (Final Synchronized Model)

In the final iteration, the system's structural components—invariants I, transition guards E, and synchronization channels A—were completely and correctly aligned. Visually, the final automata network displays a streamlined topology where every synchronization channel (e.g., *req?*, *assign!*, depicted in cyan) pairs perfectly without orphaned states or unconstrained loops (the comprehensive structure of this verified final iteration is presented as Fig. 5). The UPPAAL verifier confirmed that the state-space $\mathcal{S}$ successfully satisfied the entire triple-constraint property ($\mathcal{S} \vDash \phi_{time} \wedge \phi_{safety} \wedge \phi_{deadlock}$). The refinement loop condition evaluated to false, terminating the algorithm and returning the mathematically verified TA model M^*.

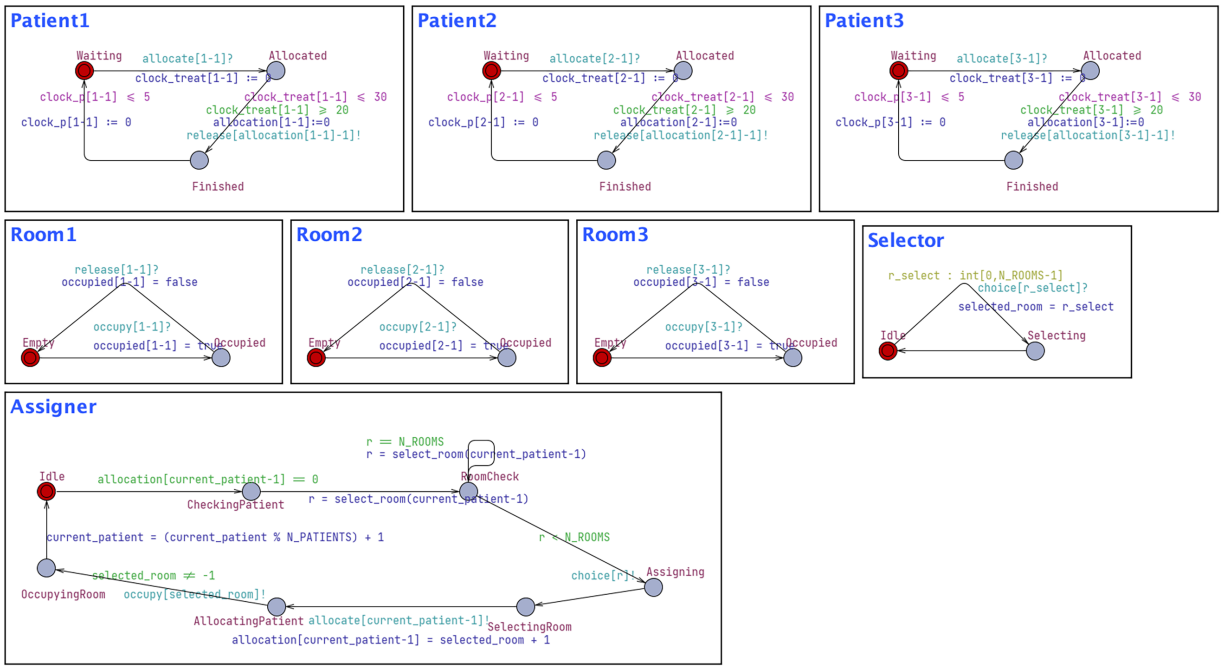


Figure 5: UPPAAL model in iteration 4. The final structure verified to be deadlock-free.

2.5 Translation to Design Artifacts

To bridge the gap between formal verification and practical implementation, the proven formal model was systematically translated into UML design artifacts. This process relies on a consistent set of mapping rules to ensure the software's architectural blueprint inherits the model's guaranteed reliability. By establishing a direct correspondence between model elements and UML symbols, we maintain the integrity of the verified logic throughout the transition. This systematic synthesis prevents the introduction of new design errors during the documentation phase of the SDLC.

The translation is defined for the following key diagrams:

- Class Diagram: Automata templates are mapped to classes, their variables become attributes, and channel interactions imply methods or associations [27].
- Sequence Diagram: Instantiations of automata are represented as objects, while the synchronization signals exchanged between them are translated into messages to visualize their dynamic interactions [32].
- State Diagram: An automaton's locations, transitions, and their corresponding guards and updates are directly mapped to states, transitions, and their respective labels to document each component's internal behavior [33,34].

To automate this rigorous transformation process, we developed PRISMA Translator, a dedicated lightweight application. This tool is specifically engineered to parse the verified TA models from UPPAAL's native Extensible Markup Language (XML) format and computationally convert them into PlantUML scripts representing the aforementioned Class, Sequence, and State Machine diagrams. Beyond automated generation, PRISMA Translator features an integrated interactive text editor. This allows software engineers to directly modify and refine the generated PlantUML scripts in real-time to seamlessly resolve any syntactical discrepancies or customize the final artifacts.

3 Results

The methodology was validated across three distinct dimensions to ensure a comprehensive assessment of the system's reliability, as illustrated in the integrated workflow in Fig. 6. The analysis focused on the logical correctness of safety properties, the efficiency of iterative model refinement, and the structural integrity of the synthesized UML artifacts. This multi-faceted validation demonstrates that the final model is both theoretically robust and practically applicable for safety-critical software engineering.

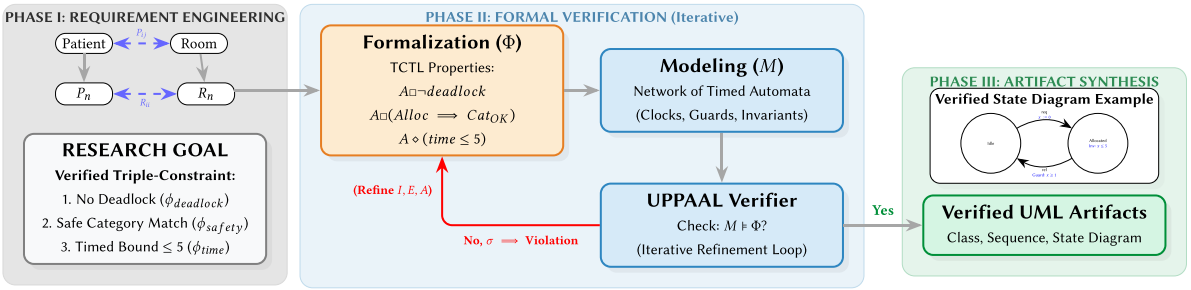


Figure 6: Integrated workflow of the triple-constraint formal verification approach for room allocation.

3.1 Verification of Safety and Liveness Properties

The primary validation was conducted using the UPPAAL model checker to exhaustively explore the state space of the room allocation workflow. The verifier confirmed that the core safety property ($A\,\Box\,\neg deadlock$) holds in the final iteration (Iteration 4). This signifies that no reachable state in the concurrent environment allows for a transition halt. The verification results confirm that the structural changes made to the *PatientAssigner* effectively eliminated the circular dependencies identified in the earlier stages. Fig. 5 illustrates the final refined architecture, which effectively eliminates all previously identified concurrency flaws.

In accordance with the proposed triple-constraint framework and the iterative refinement algorithm, we validated the functional integrity of the system through the following TCTL specifications:

- Category Matching (ϕ_{safety}): $A[]\forall(p : \text{id\_t}) \cdot (\text{Patient}(p).\text{Allocated} \implies \text{room\_cat}[\text{assigned\_room}[p]] == \text{pat\_cat}[p])$. Verified to be satisfied, ensuring 100% clinical compliance.
- Temporal Bound (ϕ_{time}): $A[]\forall(p : \text{id\_p}) \cdot (\text{Patient}(p).\text{Waiting} \implies \text{Patient}(p).clock \leq 5)$. Verified to hold, guaranteeing that no patient exceeds the latency threshold.
- Deadlock-Freedom & Progress ($\phi_{deadlock}$): Verified primarily through the global absence of deadlocks ($A\,\Box\,\neg$deadlock) to ensure no terminal stalls occur. Furthermore, to guarantee starvation-freedom, we verified that every valid request eventually reaches a completion state ($A \diamond (p : \text{id\_r}) \cdot \text{Patient}(p).\text{Allocated}$).

3.2 *Empirical Analysis of Model Refinement*

The efficacy of the iterative approach is demonstrated by the incremental elimination of specific concurrency violations that were discovered during each verification run. In the transition from Iteration 1 to 2, we validated that structural decomposition effectively reduces component-level complexity, although it necessitates more robust synchronization. Subsequently, the shift from Iteration 3 to 4 proved that basic resource release logic is insufficient to prevent deadlock without a serialized state-based signaling protocol.

The growth in model complexity, as illustrated in Fig. 7, serves as an empirical metric for this design evolution. It shows that higher design maturity, reflected in the increased number of states and transitions, was necessary to achieve zero-deadlock guarantees. This systematic increase in complexity represents a controlled effort to harden the system's architecture against non-deterministic failures.

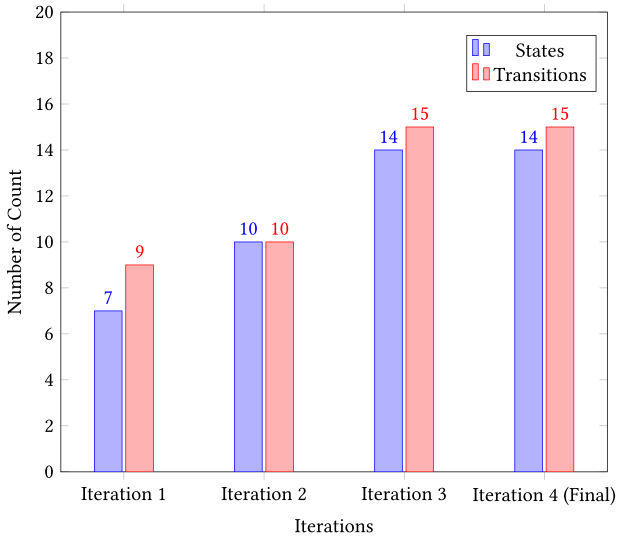


Figure 7: The growth of model complexity to ensure reliability.

To validate the practical utility of this formal approach, we performed a thorough traceability check. The iterative refinement process, detailed in Table 1, resulted in a final model proven to be deadlock free. As presented in Fig. 5, the key to this formal correctness was a significant restructuring of the *PatientAssigner* automaton into a distinct state-based signaling mechanism.

Table 1: Summary of verification results and model refinement per iteration.

Iter.	A □ ¬deadlock	Time Const.	Analysis → Corrective Action
1	Failed	Failed	Circular dependency → Decomposed the system
2	Failed	Satisfied	Unsynchronized signals → Redesigned synchronization
3	Failed	Satisfied	Resource exhaustion → Added release logic
4	Satisfied	Satisfied	All properties verified via state-based coordination

3.3 *Computational Feasibility and State-Space Analysis*

This section demonstrates the execution simulation of the final refined model under conditions where the number of concurrent patients (N) matches or exceeds the total number of available rooms or shared

resources (R, where $R = 3$ in this study), particulary evaluated for configurations of $N = 3, 4, 5$, and 6. To systematically evaluate the model during this exact verification phase, the analysis focuses on three foundational TCTL properties. First, the Deadlock property ($\phi_{deadlock}$ (For exact verification, ϕ_{deadlock} proves absolute deadlock freedom ($A\,\square\,\neg$deadlock). For SMC, it is adapted as ϕ_{live} to measure bounded-time completion probability, as stochastic simulation cannot guarantee absolute deadlock freedom)) ensures the system never reaches a halting state where no further transitions are possible, formally expressed as $A\,\square\,\neg$deadlock. Second, the Safety property (ϕ_{safety}) ensures strict resource matching, guaranteeing that rooms are solely allocated to patients of the corresponding medical category, formalized as $A\,\square\,(\forall i \in \{0, \ldots, \text{N\_PATIENTS} - 1\}.\ \text{allocation}[i] \neq 0 \implies \text{patient\_ category}[i] = \text{room\_ category}[\text{allocation}[i] - 1])$. Third, the Temporal Bound property (ϕ_{time}) guarantees strict temporal compliance, ensuring that any unallocated patient in the waiting state never exceeds the maximum allowed waiting time. Dynamically formulated for N patients, this is expressed as $A\,\square\,(\forall i \in \{0, \ldots, \text{N\_PATIENTS} - 1\}.\ \text{Patient}(i+1).\text{Waiting} \wedge \text{allocation}[i] = 0 \implies \text{clock\_p}[i] \leq 50)$. Although the model successfully satisfies these properties at lower scales, at $N = 6$, the system encounters a severe state-space explosion problem, rendering exact verification via UPPAAL computationally intractable on the designated workstation specifications. To mitigate this computational bottleneck, we pivot to a Statistical Model Checking (SMC) verification paradigm for expanded scaling configurations ($N = 3, 4, 5, 6, 10$ and 25).

It is important to note that the temporal threshold in ϕ_{time} was intentionally scaled from the initial formal specification of 5 time units to 50 time units during this advanced state-space and SMC analysis. This specific parameter adjustment ensures that the invariant bounding the maximum waiting time within the *Waiting* state strictly exceeds the required treatment duration (*clock_treat*) spent by a patient utilizing a room in the *Allocated* state. By guaranteeing that the timeout threshold is significantly larger than the treatment execution time, the verification framework realistically captures worst-case queueing dynamics without triggering premature logical failures.

The exact state-space exploration tasks were conducted on a computer equipped with an Intel Xeon E3-1245 v2 Central Processing Unit (CPU 4 cores, 8 threads) clocked at 3.40 GHz and 16 GB Random Access Memory (RAM). Every time a new process instance is introduced into the concurrent pool (i.e., an additional Patient automaton template), the computational overhead scales non-linearly rather than linearly. The system must exhaustively calculate all potential non-deterministic interactions and interleavings among the active patient entities and shared room states. Mathematically, the configuration state space grows exponentially following the scale of $O(k^N)$, where k represents the average state depth per process.

As a consequence of this exponential expansion, scaling from $N = 4$ to $N = 5$ triggers a massive surge in symbolic configurations, causing memory demands to spike drastically. When attempting exact verification at $N = 6$, the exponential multiplication of interleavings completely exhausts the available hardware resources, resulting in a state-space explosion. Consequently, the verifier encounters an Out-Of-Memory (OOM) error, leaving the system unable to resolve the safety and liveness properties exactly. However, as summarized in Table 2, the empirical results confirm that all specified properties are successfully satisfied and mathematically proven up to the computational limit of $N = 5$.

Table 2: Model checking performance statistics for varying configurations (N (N represents the number of concurrent patients competing for shared resources)).

N	Property	Explored	Stored	Mem. (MB)	Time (ms)
3	ϕ_{deadlock}	1752	1546	21	16
	ϕ_{safety}	2054	1804	11	21

(Continued)

Table 2 (continued)

N	Property	Explored	Stored	Mem. (MB)	Time (ms)
	ϕ_{time}	1004	814	3	25
4	ϕ_{deadlock}	175,472	114,215	33	4287
	ϕ_{safety}	319,106	107,322	45	7053
	ϕ_{time}	134,688	86,460	45	1976
5	ϕ_{deadlock}	5,741,371	2,788,451	527	1,159,801
	ϕ_{safety}	16,226,248	3,673,148	1435	5,449,275
	ϕ_{time}	4,773,010	2,137,778	1189	562,200
6	ϕ_{deadlock}	40,375,266	15,181,595	3454	61,272,522
	ϕ_{safety}	OOM	OOM	OOM	OOM
	ϕ_{time}	OOM	OOM	OOM	OOM

To evaluate the architecture's scalability without exhausting physical memory, we transition from exact verification to Statistical Model Checking (SMC). Because exact verification suffers from state-space explosion at higher dimensions, SMC provides a highly effective alternative by estimating satisfaction probabilities through randomized simulation runs. Crucially, because SMC relies on finite simulation traces rather than exhaustive state-space traversal, it inherently cannot evaluate explicit deadlock queries. Therefore, it is an absolute prerequisite that the system's freedom from deadlocks is first rigorously proven using exact verification. Once structurally validated, SMC is utilized to evaluate three fundamental properties: Reachability (ϕ_{live}) ($Pr[\leq 100](\Diamond(\forall i \in \{1,\ldots,\text{N\_ROOMS}\}.\ \text{Patient}(i).\text{Allocated}))$), Safety ($\phi_{safety}$) ($Pr[\leq 100](\Box(\forall i \in \{0,\ldots,\text{N\_PATIENTS}-1\}.\ \text{Allocation}[i] \neq 0 \implies \text{patient\_category}[i] = \text{room\_category}[\text{allocation}[i]-1]))$), and Temporal Bound ($\phi_{time}$) ($Pr[\leq 100](\Box(\forall i \in \{0,\ldots,\text{N\_PATIENTS}-1\}.\ \text{Patient}(i+1).\text{Waiting} \wedge \text{allocation}[i] = 0 \implies \text{clock\_p}[i] \leq 50))$).

As summarized in Table 3, Statistical Model Checking (SMC) allows the framework to scale up to $N = 25$ while maintaining a perfectly stable physical memory footprint of precisely 14 MB across all evaluated dimensions. The experimental results confirm that the system consistently maintains a "satisfied" status for robust reachability (ϕ_{live}), safety (ϕ_{safety}), and temporal bound (ϕ_{time}) guarantees across all patient loads. Under tightened statistical parameters, all three properties evaluate to near-absolute certainty (estimated probability approaching 1.0 at a 99% confidence interval) and converge rapidly within exactly 528 simulation runs. The absence of rare-event probabilities confirms that the system's architectural design guarantees successful concurrent transitions to the *Finished* state without risking deadlocks or livelocks.

Table 3: Statistical model checking (SMC) performance statistics for the verified model (M^*) on varying patient loads (N).

N	Property	Mem. (MB)	Time (ms)
3	ϕ_{live}	14	13
	ϕ_{safety}	14	132
	ϕ_{time}	14	157

(Continued)

Table 3 (continued)

N	Property	Mem. (MB)	Time (ms)
4	ϕ_{live}	14	15
	ϕ_{safety}	14	171
	ϕ_{time}	14	231
5	ϕ_{live}	14	16
	ϕ_{safety}	14	202
	ϕ_{time}	14	284
6	ϕ_{live}	14	17
	ϕ_{safety}	14	235
	ϕ_{time}	14	366
10	ϕ_{live}	14	22
	ϕ_{safety}	14	376
	ϕ_{time}	14	672
25	ϕ_{live}	14	43
	ϕ_{safety}	14	878
	ϕ_{time}	14	97,059

Note: All verification properties were evaluated using 528 simulation runs, achieving an estimated probability of $[0.990, 1.0]$ with a 99% confidence interval (The number of simulation runs evaluates dynamically based on Sequential Hypothesis Testing under tightened statistical parameters ($\alpha = 0.01$, $\varepsilon = 0.01$). Because the estimated probabilities for all properties (ϕ_{live}, ϕ_{safety}, and ϕ_{time}) approach an extreme deterministic bound (near 1.0), the SMC engine rapidly achieves the required 99% confidence interval within exactly 528 runs across all evaluated patient loads)

A detailed analysis of the performance metrics reveals the computational nature of the system's verification bottleneck. CPU time scales gracefully and linearly for ϕ_{live} and ϕ_{safety} as the patient load (N) increases. However, verifying the strict temporal bounds (ϕ_{time}) introduces a significant computational hurdle at higher concurrency tiers. Specifically, at $N = 25$, the CPU time required to verify ϕ_{time} spikes dramatically to 97,059 ms. This exponential increase reflects the computational complexity of evaluating interleaved global clocks and concurrent patient timers within a deeply expanded stochastic state space. These figures empirically validate that the system is mathematically proven to be safe and perfectly responsive, but the verification of dense temporal logics inherently throttles the SMC engine's execution time at extreme concurrency levels.

3.4 Generalized Blueprint and Verification Prerequisites for EHR Workflows

To establish a concrete empirical baseline for our formal verification framework, we conducted an architectural workflow analysis of OpenMRS (https://github.com/openmrs/openmrs-core), a widely adopted open-source Electronic Health Record (EHR) system. This platform selection is directly motivated by its global prominence and superior user performance in executing core clinical tasks [35]. Because the OpenMRS core is inherently Java-based, its underlying Object-Oriented Programming (OOP) paradigm aligns seamlessly with the structural modeling principles of TA. In this context, software classes translate naturally into parameterized automata templates, while dynamic object interactions map directly to synchronization channels.

As detailed in Table 4, we systematically decompose several critical clinical workflows to demonstrate the broader extensibility of our approach. It is crucial to clarify that while the foundational operations listed in the first column are natively derived from the OpenMRS ecosystem, the subsequent mapping parameters (Columns 2 to 5) are not inherent features of the software. Rather, the defined Automata Candidates, Shared Resources, Matching Logic, and Timing Constraints represent our novel theoretical engineering—a conceptual construct deliberately formulated by the authors to translate standard EHR routines into verifiable mathematical models.

Table 4: Generalized mapping of the TA model architecture (OpenMRS core workflows).

Workflow	Automata	Shared Resources	Matching Logic	Timing Constraints
Medical Equipment	• *Patient* • *Equipment*	Availability of specific medical machines (Ventilators, Intensive Care Unit (ICU) Beds) (k).	Patient severity vs. equipment priority: *patient_urgency* ≥ *equip_priority*	• Usage limit: *clock_use* ≤ *T_use* • Urgency response
Laboratory Orders	• *Physician* • *Analyzer*	Laboratory machine capacity and chemical reagent availability (L).	Lab specimen type vs. machine capability: *order_type* == *machine_cap*	• Turnaround: *clock_tat* ≤ *T_max* • Sample expiration
Pharmacy Dispense	• *Pharmacist* • *Dispenser*	Physical drug stock in the pharmacy warehouse (S).	Prescribed drug vs. stock availability: *drug_id* == *stock_id* && *stock* > *0*	• Dispensing time: *clock_prep* ≤ *T_prep* • Dosage interval
Program Enrollment	• *Patient* • *Coordinator*	Capacity quota for public health programs (e.g., Human Immunodeficiency Virus (HIV)/Tuberculosis (TB) (Q).	Patient clinical status vs. inclusion criteria: *diag_status* == *elig_criteria*	• Status duration: *clock_state* ≤*Max_State* • Review schedule

This mapping illustrates that defining explicit candidates for concurrent automata, heavily contended shared resources, strict logical matching constraints, and critical temporal invariants constitutes the fundamental prerequisites for executing the verification process. While the Room Allocation workflow serves as the primary empirical focus of this study, this generalized blueprint explicitly confirms that our proposed methodology is highly capable of modeling and verifying a wide spectrum of other complex EHR operations. Ultimately, fulfilling these structural requirements provides the foundational raw materials necessary to construct robust TA models, enabling rigorous formal verification and temporal analysis natively within the UPPAAL environment.

3.5 Synthesis and Traceability of Design Artifacts

By applying the mapping rules established in Section 2.5, the PRISMA Translator successfully generated the complete UML suite. As summarized in Table 5, the formal-to-design synthesis ensures a deterministic mapping from TA to Class, Sequence, and State Machine diagrams.

Table 5: Mapping rules for formal-to-design synthesis.

UPPAAL (Formal Model)	UML Artifact (Design)	Software Implementation Context
Automata Template	Class Diagram	Object structure and class attributes
Location	State (State Diagram)	Current execution context or lifecycle phase
Edge/Transition	Transition (State Diagram)	Logic or method call triggering state change
Guard	Pre-condition	*if* statements or constraints for execution
Invariant	State Constraint	Real-time timeout logic or safety bounds
Channel (Sync)	Message (Sequence)	Inter-process communication and synchronization

While Class and Sequence diagrams are successfully generated to capture the static object structures and message-passing protocols, this paper focuses its visual demonstration and in-depth analysis on the behavioral modeling component. The internal logic of the verified system is documented through the State Machine Diagram, as illustrated in Fig. 8. This high-fidelity translation allows the development team to implement the internal state-machine logic of the system components with mathematical certainty.

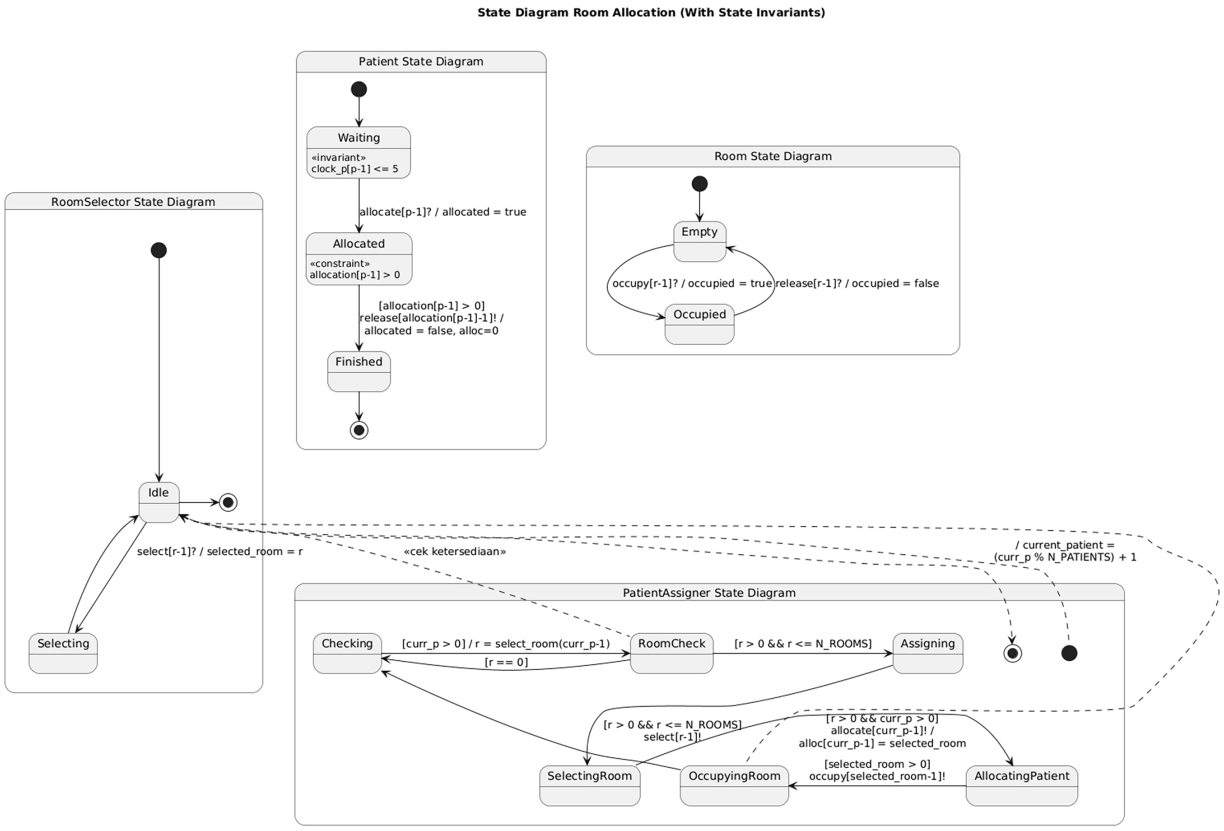


Figure 8: Integrated state machine diagram generated from the final UPPAAL model.

Among all the synthesized UML artifacts, the State Machine Diagram serves as the most critical architectural blueprint, owing to its direct isomorphism with the underlying TA model. This one-to-one mapping allows for a lossless translation of the verified triple-constraint specification into practical software design. In particular, the TA's location invariants (e.g., strict temporal bounds such as $clock \leq 5$) are directly embedded as internal state constraints, while the meticulously refined logical guards and synchronization

channels are represented as transition triggers. By explicitly codifying these verified boundaries within the State Machine Diagram, the resulting design achieves a "Correctness-by-Construction" status. It inherently inherits the mathematical guarantees of the UPPAAL verification—most notably the absolute prevention of circular waits ($A \square \neg$deadlock) governed by the *PatientAssigner* orchestrator—thereby providing developers with an unambiguously safe, deadlock-immune implementation guide.

3.6 Case Study Execution and Expert Validation: Radiotherapy and Room Allocation Workflow

We selected two critical EHR workflows: the previously discussed room allocation and a newly adopted radiotherapy model from a hospital in Purwokerto, Indonesia. While representing local best practices, the current operational workflow lacks strict time constraints. Expert physicians clinically validated that adding temporal constraints and state invariants is crucial, confirming the logical feasibility of implementing this enhanced workflow.

The radiotherapy UPPAAL TA model emphasizes safety and resource efficiency via an automatic preemption mechanism, allowing emergency (statim or STAT) patients to preempt regular patients when the Computed Tomography (CT) machine, Treatment Planning System(TPS) computer, and Linear Accelerator (LINAC) are at full capacity. To prevent state-space explosion in large-scale simulations, a global semaphore-based Counting Abstraction is synchronized with local clocks and safety timeouts, guaranteeing a deadlock-free system. The topology, preemption transitions, and guard functions follow Algorithm 1. As shown in Fig. 9, the verified final automaton is deadlock-free and satisfies all safety and temporal constraints.

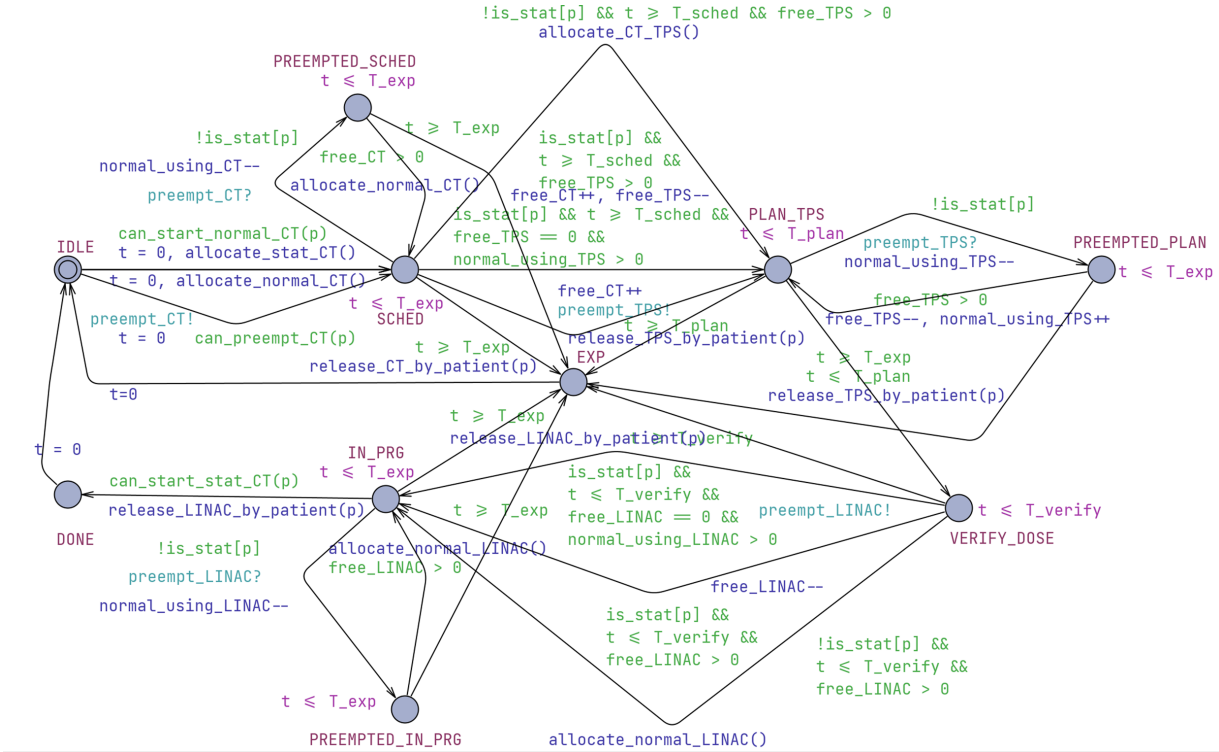
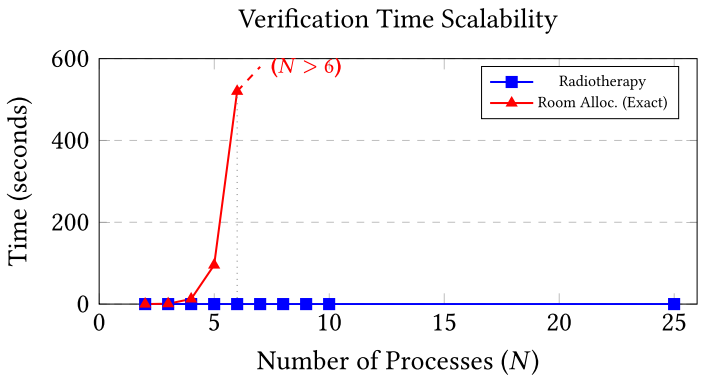


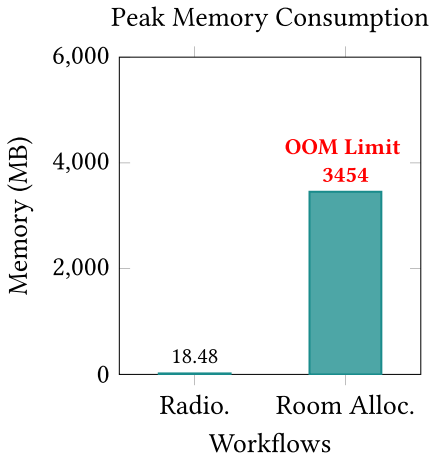
Figure 9: The final radiotherapy automata.

The state-space explosion stems from the automata's design architecture, not workflow bias. Using exact UPPAAL verification, both the four-automata room allocation and the single-modular radiotherapy models satisfy deadlock-freedom, safety, and temporal guarantees. However, scaling the resource requesters

(N) reveals stark computational contrasts. The room allocation design hits an Out-Of-Memory (OOM) error at $N = 6$. Conversely, the radiotherapy design remains highly stable from $N = 2$ to $N = 25$, constantly storing exactly 81 states with minimal memory fluctuations (17.16 to 18.48 MB) and rapid execution (1 to 7 ms). Fig. 10a illustrates the exact method's state-space explosion and OOM trigger at $N > 6$, while Fig. 10b compares the extreme peak memory consumption at $N = 6$.



(**a**) Exact Time Scalability of 2 Workflows



(**b**) Peak System Memory Consumption

Figure 10: Comparative visualization of verification performance: (**a**) The line graph illustrates failures due to a state-space explosion, triggering an OOM when $N > 6$, (**b**) The bar chart shows extreme peak memory consumption exceeding the system's RAM capacity limit.

Finally, the verified Radiotherapy Automaton satisfying the triple constraints is translated into class, sequence, and state diagrams (e.g., the PlantUML script in Listing 1) using PRISMA translator. Transforming these UML blueprints into object-oriented skeleton code (e.g., Java, C++) falls under the SDLC [36,37]. Thus, this study limits its scope to artifact generation, as indicated by arrow 3 in Fig. 1.

Listing 1: PlantUML source code for the radiotherapy workflow state machine (Snippet).

```
@startuml State_Machine_Diagram
skinparam monochrome false
state "RadiotherapyWorkflow" as RadiotherapyWorkflow {
  state "IDLE" as id0
  state "SCHED" as id1 : Invariant: t <= T_exp
  state "PLAN_TPS" as id2 : Invariant: t <= T_plan
```

```
  state "VERIFY_DOSE" as id3 : Invariant: t <= T_verify
  state "IN_PRG" as id4 : Invariant: t <= T_exp
  state "DONE" as id5
  state "EXP" as id6
  state "PREEMPTED_SCHED" as id7 : Invariant: t <= T_exp
  state "PREEMPTED_PLAN" as id8 : Invariant: t <= T_exp
  state "PREEMPTED_IN_PRG" as id9 : Invariant: t <= T_exp
  [*] --> id0
  id6 --> id0 : Assignment: t=0
 id0 --> id1 : Guard: !is_stat[p] && free_CT > 0  && laterality_validated[p]
                && clinical_history_validated[p] \n\ Assignment: t=0, free_CT--,
                    ↪ normal_using_CT++
  id0 --> id1 : Guard: is_stat[p] && free_CT > 0 && laterality_validated[p]
                && clinical_history_validated[p] Assignment: t=0, free_CT--
  id0 --> id1 : Guard: is_stat[p] && free_CT == 0 && normal_using_CT > 0 &&
      ↪ laterality_validated[p]
                && clinical_history_validated[p] Sync: preempt_CT! Assignment: t =
                    ↪ 0
  'more detail.....................'
  id4 --> id5 : Assignment: free_LINAC++, normal_using_LINAC = normal_using_LINAC -
                (is_stat[p] ? 0 : 1)
 'more detail.....................'
  id4 --> id6 : Guard: t >= T_exp\nAssignment: free_LINAC++, normal_using_LINAC =
                normal_using_LINAC - (is_stat[p] ? 0 : 1)
  id9 --> id6 : Guard: t >= T_exp
  id5 --> id0 : Assignment: t = 0
}
@enduml
```

4 Discussion

The architectural evolution of our room allocation model demonstrates that intuitive design cannot safeguard concurrent EHR workflows against non-deterministic bugs. To articulate the significance of these findings, this section synthesizes the framework's theoretical and practical implications. We first analyze how hierarchical orchestration breaks core concurrency deadlocks, followed by a comparative benchmarking that underscores the necessity of dense-time verification and proactive blueprint synthesis. Finally, we assess practical deployment feasibility, focusing on queueing scalability under dynamic loads. Within this context, UPPAAL's exact verification serves as the primary deterministic frontline—successfully scaled via its integrated SMC extension—while the future roadmap integrates external symbolic SMT to handle infinite-state hospital-wide complexities.

4.1 Resolving Concurrency via Orchestration

The "Heisenbugs" in early iterations were driven by Circular Wait and Hold and Wait conditions [38]. As depicted in Fig. 11, we resolved this by transitioning from a decentralized resource-grab approach to a hierarchical orchestration using the *PatientAssigner*. This structural change enforced a strictly ordered, predictable protocol. The verified property $A\,\square\,\neg deadlock$ provides mathematical certainty that no combination of events can lead to a system freeze. This orchestration eliminates the Circular Wait condition without relying on heavy-duty database locks that degrade EHR performance, significantly reducing operational risks before deployment.

4.2 Comparative Analysis: Beyond Structural Logic

A key contribution of this work is its temporal granularity. While Mukhlash et al. [15] successfully modeled workflows using Coloured Petri Nets (CPN) focusing on logical reachability, their approach

abstracts away continuous time (Fig. 12). Although this foundational CPN methodology was later enhanced by Medina-Garcia et al. [16] through architectural simulations and conformance checking to evaluate system performance, such simulation-based techniques fundamentally evaluate specific execution traces rather than providing exhaustive mathematical proofs. Our use of Timed Automata (TA) addresses this exact gap by introducing dense-time TCTL invariants, which are crucial for safety-critical triage where minor delays can be catastrophic.

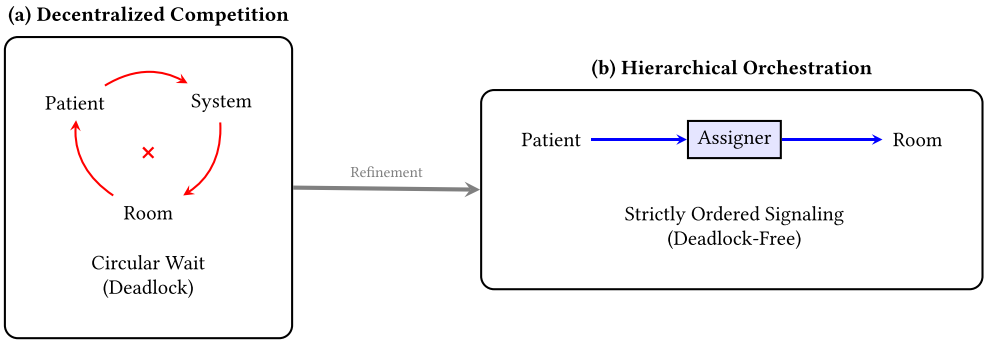


Figure 11: Architectural transition from (**a**) decentralized competition to (**b**) hierarchical orchestration. By introducing the *Assigner* as a central orchestrator, the circular wait identified in previous designs is eliminated, ensuring liveness.

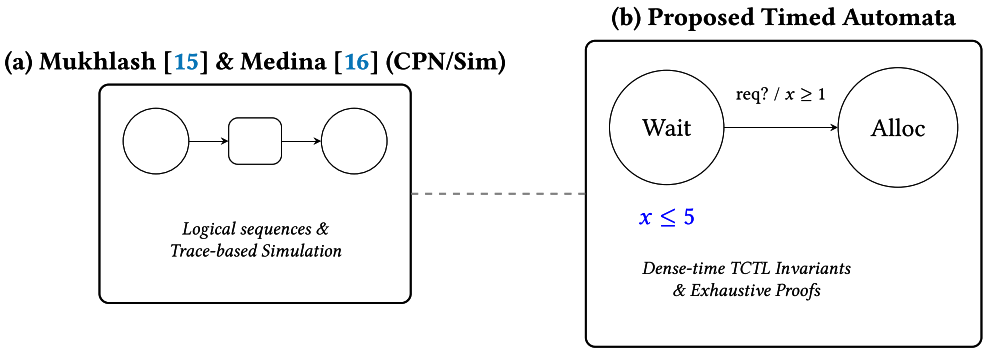


Figure 12: Comparison of modeling granularity. (**a**) Mukhlash et al. [15] and Medina-Garcia et al. [16] focus on discrete process flows and simulations. (**b**) Our proposed TA model integrates explicit clock variables (x) and invariants to guarantee real-time safety exhaustively.

Unlike static analysis tools, our methodology verifies safety within specific temporal bounds. The property $A \square \, Patient.Waiting \rightarrow Patient.clock \leq 5$ proves the system is both logically correct and operationally efficient. Our model navigates a significantly denser state-space than typical discrete CPN models. This added complexity is a necessary trade-off to mathematically guarantee real-time safety.

4.3 Performance Evaluation and Trade-Off Analysis

To ensure a fair comparison, we stress-tested the benchmarked frameworks by replicating their mathematical structures up to $N = 25$ processes. We evaluated Medina et al.'s simulation-based Petri Net [16] via a *simpy* Python replica, modeling transition delays with Gaussian distributions based on their empirical logs (e.g., arrival: 2547.972 s, reading: 30.066 s, invoicing: 17.380 s). Profiled using *time* and *tracemalloc*, Medina et al.'s parameter N scales seamlessly due to independent, non-interacting loop iterations. However,

applying this to our model introduces severe synchronization overhead, as concurrent processes compete for bounded room resources.

Similarly, we modeled Mukhlash et al.'s Coloured Petri Net (CPN) [15] using an object-oriented replica. Tokens encapsulate three financial variables (*SO Cost*, *PCP Cost*, *PO Cost*) as transition guards over macro-level timelines, incorporating empirical delays like ≈6.78 days for compliance checks and ≈166-day bottlenecks for weight reviews. Stress-testing this architecture across 16 foundational places revealed that explicitly constructing reachability graphs triggers an untamable state-space explosion. Driven by the 16 structural positions, token color variations (2–5 per attribute), and unbounded recalculation (*RE*) loops, the state space aggressively expands from ~128 states at $N = 1$ to over 4.65×10^{12} states at $N = 5$, causing hardware Out-of-Memory (OOM) failures. Our exact TA verification faces a similar exponential halt ($O(k^N)$) at $N \geq 6$. Conversely, Medina et al.'s sequential probabilistic evaluation avoids graph construction—yielding extremely fast execution (3.774 ms at $N = 25$) and negligible memory (<0.018 MB)—but fundamentally sacrifices formal safety verification.

Ultimately, Table 6 highlights a crucial trade-off between computational cost and verification rigor. Medina et al.'s simulation minimizes overhead but lacks the formal safety guarantees vital for healthcare. Conversely, exact state-space exploration (Mukhlash et al. and our exact TA) provides mathematical certainty but is practically unusable under heavy concurrency due to catastrophic memory limits. The proposed SMC methodology emerges as the optimal pragmatic solution. By leveraging statistical trace simulations within defined (ε, α) confidence bounds, it delivers rigorous formal safety validation (≥99% confidence) while completely neutralizing state scaling. Our TA-SMC framework locks peak memory at a strictly constant ≈ 14 MB with competitive execution times up to $N = 25$, proving to be a robust, feasible architecture for verifying complex clinical workflows without state-space explosion.

Table 6: Comprehensive comparison of state-transition based business process modeling.

Aspect/Feature	Mukhlash et al. [15] (CPN)	Medina-Garcia et al. [16] (PN Sim)	Proposed (TA + SMC)
Paradigm & Tool	Reactive/Historical. Extracts logs for past bottlenecks (ProM).	Stochastic/Probabilistic. Simulates future behavior (Python).	Proactive/Deterministic. Mathematical proofs pre-implementation (UPPAAL).
Time Granularity	Discrete timelines. Captures macro-level cycles (165–192 days).	Stochastic distributions. Micro-services via average logs.	Dense-time continuum. Strict boundaries via clock invariants.
Safety Guarantee	Reachability Graph. Prone to state-space explosion.	No formal guarantee. Relies on random path generation.	Exact: 100% mathematical proof. SMC: ≥99% statistical confidence.
Time Complexity	$O(\|E\|)$ where $\|E\|$ is log events. Variable parse time.	$O(T \times L)$ where T is traces. Extremely fast (~0.7 to 3.7 ms).	Exact: Exponential $O(k^N)$. SMC: Scales linearly $O(N)$ (max ~97 s at $N = 25$).
Space & Memory	Intractable for exact proofs. ≈7.7 GB at $N = 4$; OOM at $N \geq 5$.	Minimal (single path evaluation). <0.018 MB at $N = 25$.	Exact: Exponential (OOM at $N \geq 6$). SMC: Strictly constant ≈ 14 MB up to $N = 25$.

4.4 The Proactive Paradigm vs. Reactive Validation

Our methodology represents a fundamental shift from the reactive "UML-to-UPPAAL" validation proposed by Arfi et al. [27]. Their framework uses formal verification as a post-hoc diagnostic tool, often causing a "Redundancy Penalty" as engineers repeatedly oscillate between design and verification layers to fix late-stage bugs. By contrast, we adopt a proactive synthesis strategy (UPPAAL → UML). By verifying the formal model against all triple-constraint properties before generating any UML artifacts, we achieve a "Correctness-by-Construction" status. This proactive flow (Fig. 13) minimizes human error during the architectural phase, ensuring the resulting blueprints—anchored by the verified State Machine Diagram—are inherently deadlock-immune. While currently efficient, our reliance on explicit state-space exploration may face bottlenecks with high-dimensional continuous variables. Shifting toward symbolic constraint satisfaction will be a promising path for scaling this verification to more complex, hospital-wide EHR dependencies.

4.5 Practical Deployment and Integration Feasibility

It is crucial to delineate that this framework focuses strictly on design-level formal verification, culminating in mathematically verified Object-Oriented UML blueprints that remarkably narrow the gap between theoretical design and practical implementation. By providing exact structural data via Class Diagrams for ORM-based persistence and dynamic behavioral logic via State Machine Diagrams, developers are equipped with a zero-defect architectural baseline before coding begins. This intentional boundary underscores a "prevention-by-design" paradigm, systematically eliminating structural flaws early rather than treating verification as a supplementary testing burden at the development's tail end.

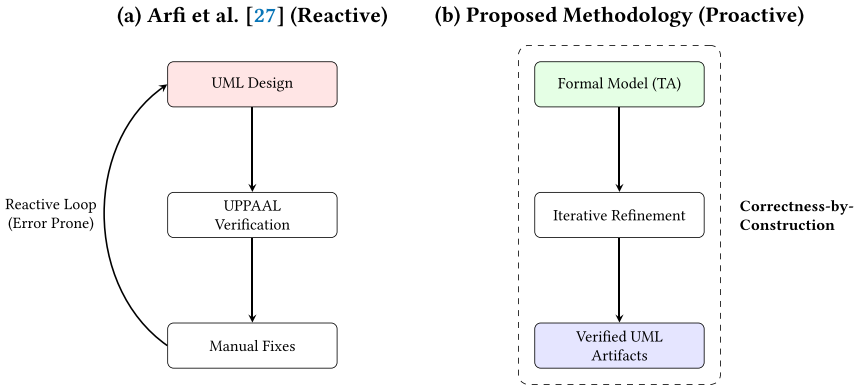


Figure 13: Paradigm shift in workflow. (**a**) The reactive validation loop by Arfi et al. [27] compared to (**b**) our proactive approach, which derives verified UML blueprints directly from a mathematically proven state-space.

In addition, the core mathematical architecture of the proposed triple-constraint framework is inherently workflow-agnostic and extensible. Utilizing generic TA primitives, the methodology can be systematically replicated for other safety-critical modules—such as medication prescribing or surgical scheduling—while relying on SMC to manage state-space explosion in larger heterogeneous environments. Although formal methods do not directly evaluate human-centric attributes, the "UPPAAL → UML" synthesis pipeline indirectly enhances usability and maintainability by providing intuitive, mathematically guaranteed visual blueprints and strict structural traceability. Finally, this scalable model facilitates safe offline adaptation in evolving healthcare ecosystems; software architects can re-verify parameter or guard changes ($A \Box \neg deadlock$) before systematically re-synthesizing the updated architecture into code, ensuring safe evolution without introducing non-deterministic concurrency bugs.

4.6 Addressing State-Space Explosion via Statistical Model Checking (SMC)

Although TA via UPPAAL is strategically maintained as the primary modeling gateway to provide absolute deterministic guarantees against deadlocks and timing violations, this exhaustive verification empirically faces inherent computational boundaries—specifically, state-space explosion—as the number of concurrent processes scales up ($N \geq 6$). It is precisely at this computational threshold that the probabilistic verification framework becomes crucial. Rather than being employed as the initial design baseline, which would only yield stochastic reliability, SMC is applied specifically as a scalability strategy. By relying on randomized simulation traces that bypass the construction of the full state graph, SMC successfully validates system operability under high loads ($N = 25$) with stable memory efficiency. This approach provides robust reliability estimations only after the core architecture has been deterministically proven safe and deadlock-free at a smaller scale. While SMC successfully provides stochastic scalability under hardware constraints, future work will explore SMT to achieve absolute proofs in infinite-state domains.

4.7 Timing Calibration and Queueing Scalability

Within the exact verification framework, evaluating the timing property (ϕ_{time}) requires careful calibration when the system transitions from ideal conditions to resource-constrained scenarios ($N > R$). In the initial specification, the waiting time threshold was strictly set to 5 time units; however, when testing the model under queueing loads ($N \geq 4$), this threshold was intentionally adjusted to 50 time units. This adjustment is based on operational reality, where each patient occupying a room (in the *Allocated* state) requires a treatment duration between 20 and 30 time units, governed by the guard $clock\_treat \geq 20$ and the invariant $clock\_treat \leq 30$. If the waiting time limit in the *Waiting* state is forced to remain smaller than the treatment duration of the preceding queue, the model checker will detect a timing violation ($\neg\phi_{time}$) purely due to resource exhaustion, rather than a concurrent architectural flaw such as a deadlock. Hence, establishing a waiting invariant of 50 time units ($clock\_p \leq 50$) becomes crucial for the proposed final model to pass the verification thoroughly. When this verified architecture is further evaluated under extreme scenarios where the number of concurrent patients exceeds room capacity, all formal properties are absolutely satisfied for every tested value of N, as empirically evidenced by the performance data in Tables 2 and 3. This consistency in absence of deadlock ($\phi_{deadlock}, \phi_{liveness}$), safety ($\phi_{safety}$), and temporal bounds ($\phi_{time}$) mathematically proves that the *PatientAssigner*-based coordination mechanics are not only effective in eliminating circular dependencies at an ideal scale but are also robust and reliable in maintaining the system's logical integrity under high clinical queueing pressure.

In evaluating this queueing scalability, the resource capacity was deliberately fixed ($R = 3$) while only the concurrent patient load (N) was scaled. This methodological choice strictly isolates queueing stress as the primary independent variable, forcing the model to verify worst-case concurrency interleavings without triggering redundant state-space explosion from symmetrical shared resources. Ultimately, this configuration accurately mirrors real-world clinical environments, where physical hospital infrastructure remains highly static against dynamically fluctuating patient influxes.

5 Conclusion and Future Work

This study confirms that formal verification utilizing UPPAAL model checking robustly guarantees the reliability of patient room allocation workflows. Applying the proposed iterative refinement algorithm systematically stripped the system design of critical flaws to satisfy the triple-constraint specification. To illustrate, the final Timed Automata model is mathematically proven free from deadlocks and timelocks ($\phi_{deadlock}$), strictly compliant with calibrated temporal upper-bounds under high queueing loads (ϕ_{time}), and logically sound in preventing illegal state transitions like patient-room category mismatches (ϕ_{safety}).

Furthermore, synthesizing these verified results into standard UML artifacts ensures subsequent software engineering phases inherently preserve these temporal, safety, and liveness guarantees, successfully bridging the gap between abstract mathematics and safety-critical software implementation.

Building upon these results, this methodology can effectively adapt to other critical healthcare workflows demanding high-integrity concurrency management, such as medical equipment, medication prescribing, surgery scheduling, laboratory orders, pharmacy dispensing, and program enrollment. However, while establishing a mathematically sound design, verification boundaries are strictly confined to deterministic Functional Requirements (FRs); human-centric usability, design-driven maintainability, network latencies, and asynchronous infrastructure-level Non-Functional Requirements (NFRs) were abstracted away to isolate core architectural flaws. To address these boundaries, future iterations will incorporate Timed Game Automata (TGA) and Stochastic Timed Automata (STA) to natively model triage interruptions and stochastic medical events. Moreover, future work will integrate SMT Solvers as a backend decision procedure to evaluate exact computation latencies in infinite-state domains, combining this symbolic integration with advanced abstraction and symmetry reduction to maintain feasibility at hospital-wide scales.

Acknowledgement: Not applicable.

Funding Statement: This research was funded by Directorate of Research, Technology, and Community Service, Ministry of Education, Culture, Research, and Technology, Indonesian Government, through the Doctoral Dissertation Research scheme (main contract number 112/E5/PG.02.00.PL/2023 and researcher contract number 1926/PKS/ITS/2023).

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Acep Taryana, Dieky Adzkiya and Imam Mukhlash; methodology, Acep Taryana; software, Acep Taryana; validation, Acep Taryana, Dieky Adzkiya, Muhammad Syifa'ul Mufid and Imam Mukhlash; formal analysis, Acep Taryana; investigation, Muhammad Syifa'ul Mufid; resources, Imam Mukhlash; data curation, Dieky Adzkiya; writing—original draft preparation, Acep Taryana; writing—review and editing, Dieky Adzkiya; visualization, MS Mufid; supervision, Imam Mukhlash and Muhammad Syifa'ul Mufid; project administration, Dieky Adzkiya; funding acquisition, Dieky Adzkiya. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Arcaini P, Bonfanti S, Gargantini A, Mashkoor A, Riccobene E. Integrating formal methods into medical software development: the ASM approach. Sci Comput Program. 2018;158(6):148–67. doi:10.1016/j.scico.2017.07.003.
2. Sittig DF, Wright A, Coiera E, Magrabi F, Ratwani R, Bates DW, et al. Current challenges in health information technology–related patient safety. Health Inform J. 2020;26(1):181–9. doi:10.1177/1460458218814893.
3. Khan S, Akhtar N, Mushtaq MF, Abdel Samee N, Mahmoud NF, Ashraf I. Towards robust electronic health record systems: integrating formal verification and process modeling techniques. BMC Med Res Methodol. 2025;25(1):215. doi:10.1186/s12874-025-02637-8.
4. Parveen R, Ruchkin I, Laurel A, Goveas N, Blouin D. Modelling and verification of resource allocation for healthcare systems. Res Sq. 2024. doi:10.21203/rs.3.rs-4650586/v1.

5. Hassan MH, Darwish SM, Elkaffas SM. An efficient deadlock handling model based on neutrosophic logic: case study on real time healthcare database systems. IEEE Access. 2022;10(6):76607–21. doi:10.1109/ACCESS.2022.3192414.
6. Mejia JMR, Rawal A, Rawat DB. Finite state automata for real-time health electronic record update: a survey. In: Proceedings of the 2022 IEEE Global Humanitarian Technology Conference (GHTC); 2022 Sep 8–11; Santa Clara, CA, USA. p. 154–61. doi:10.1109/GHTC55712.2022.9911007.
7. Coffmann Coffman EG, Elphick MJ, Soshani A. System deadlocks. Computing. 1971;3(2):67–78. doi:10.1145/356586.356588.
8. Dong Z, Wang Z, Yi C, Xu X, Zhang J, Li J, et al. Database deadlock diagnosis for large-scale ORM-based web applications. In: Proceedings of the International Conference on Data Engineering; 2023 Apr 3–7; Anaheim, CA, USA. p. 2864–77. doi:10.1109/ICDE55515.2023.00219.
9. Cray J. Why do computers stop and what can be done about it? In: Proceedings of the Symposium on Reliability in Distributed Software and Database Systems; 1986 Jan 13–15; Los Angeles, CA, USA. p. 3–12.
10. Fu H, Wang Z, Chen X, Fan X. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. Softw Qual J. 2018 sep;26(3):855–89. doi:10.1007/s11219-017-9385-3.
11. Jula H, Tralamazza D, Zamfir C, Candea G. Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008; 2008 Dec 8–10; San Diego, CA, USA. p. 295–308.
12. Kheradmand A, Kasikci B, Candea G. Lockout: efficient testing for deadlock bugs. In: Proceedings of the Workshop on Determinism and Correctness in Parallel Programming (WoDet) 2014; 2014 Mar 2; Salt Lake City, UT, USA.
13. Xu Z, Zhu J, Yang L, Zuo C. Mining the relationship between object-relational mapping performance anti-patterns and code clones. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE 2023; 2023 Jul 1–10; South San Francisco, CA, USA. p. 136–41. doi:10.18293/SEKE2023-161.
14. Khanna D, Purandare R, Sharma S. Synthesizing multi-threaded tests from sequential traces to detect communication deadlocks. In: Proceedings of the 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021; 2021 Apr 12–16; Porto de Galinhas, Brazil. p. 1–12. doi:10.1109/ICST49551.2021.00013.
15. Mukhlash I, Rumana WN, Adzkiya D, Sarno R. Business process improvement of production systems using coloured petri nets. Bull Electr Eng Inform. 2018;7(1):102–12. doi:10.11591/eei.v7i1.845.
16. Medina-Garcia S, Medina-Marin J, Montaño-Arango O, Gonzalez-Hernandez M, Hernandez-Gress ES. A petri net approach for business process modeling and simulation. Appl Sci. 2023;13(20):11192. doi:10.3390/app132011192.
17. Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J. Formal methods: practice and experience. ACM Comput Surv. 2009;41(4):19. doi:10.1145/1592434.1592436.
18. Baier C, Katoen JP. Principles of model checking. Vol. 950. Cambridge, MA, USA: MIT Press; 2008.
19. Kurshan RP. Model checking and abstraction. In: Koenig S, Holte RC, editors. Abstraction, reformulation, and approximation. Berlin/Heidelberg, Germany: Springer; 2002. p. 1–17. doi:10.1007/3-540-45622-8_1.
20. Grumberg O. Abstraction and refinement in model checking. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP, editors. Formal methods for components and objects. Berlin/Heidelberg, Germany: Springer; 2006. p. 219–42. doi:10.1007/11804192_11.
21. De Moura L, Bjørner N. Satisfiability modulo theories: an appetizer. In: Formal methods: foundations and applications. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 5902. Berlin/Heidelberg, Germany: Springer; 2009. p. 23–36. doi:10.1007/978-3-642-10452-7_3.
22. Biere A, Cimatti A, Clarke E, Zhu Y. Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99. Berlin/Heidelberg, Germany: Springer; 1999. p. 193–207. doi:10.1007/3-540-49059-0_14.
23. Kwiatkowska M, Norman G, Parker D. PRISM 4.0: verification of probabilistic real-time systems. In: Proceedings of the Computer Aided Verification: 23rd International Conference, CAV 2011; 2011 Jul 14–20; Snowbird, UT, USA. Berlin/Heidelberg, Germany: Springer; 2011. p. 585–91. doi:10.1007/978-3-642-22110-1_47.

24. Nigro L, Cicirelli F. Correctness verification of mutual exclusion algorithms by model checking. Modelling. 2024;5(3):694–719. doi:10.3390/modelling5030037.
25. Bengtsson J, Larsen KG, Larsson F, Pettersson P, Yi W. UPPAAL—a tool suite for automatic verification of real-time systems. BRICS Rep Ser. 1996;3(58). doi:10.7146/brics.v3i58.18769.
26. Alur R, Courcoubetis C, Dill D. Model-checking in dense real-time. Inf Comput. 1993;104(1):2–34. doi:10.1006/inco.1993.1024.
27. Arfi F, Courbis AL, Lambolais T, Bughin F, Hayot M. Formal verification of a telerehabilitation system through an abstraction and refinement approach using Uppaal. IET Softw. 2023;17(4):582–99. doi:10.1049/sfw2.12128.
28. Taryana A, Adzkiya D, Mufid MS, Mukhlash I, Abate A. LTL model checking for verification of electronic medical record (EMR) design. Springer Proc Math Stat. 2024;455(2):283–97. doi:10.1007/978-981-97-2136-8_21.
29. Famelis M, Salay R, Chechik M. Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of the 34th International Conference on Software Engineering; 2012 Jun 2–9; Zurich, Switzerland. p. 573–83. doi:10.1109/ICSE.2012.6227159.
30. Behrmann G, David A, Larsen KG. A tutorial on UPPAAL. In: Formal methods for the design of real-time systems. SFM-RT 2004. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 3185. Berlin/Heidelberg, Germany: Springer; 2004. p. 200–36. doi:10.1007/978-3-540-30080-9_7.
31. Alur R, Dill DL. A theory of timed automata. Theor Comput Sci. 1994;126(2):183–235. doi:10.1016/0304-3975(94)90010-8.
32. Duangmalai S, Dechsupa C. Transforming of the sequence diagram into time-automata network. In: Proceedings of the International Multi Conference of Engineers and Computer Scientists IMECS 2023; 2023 Mar 15–17; Hong Kong, China. p. 147–52.
33. Zorin D, Podymov V. Translation of UML statecharts to UPPAAL automata for verification of real-time systems. In: Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering; 2012 May 30–31; Perm, Russia. doi:10.15514/syrcose-2012-6-13.
34. Peres F, Ghazel M. A proven translation from a UML state machine subset to timed automata. ACM Trans Embed Comput Syst. 2024;23(5):72. doi:10.1145/3581771.
35. Purkayastha S, Allam R, Maity P, Gichoya JW. Comparison of open-source electronic health record systems based on functional and user performance criteria. Healthc Inform Res. 2019;25(2):89–98. doi:10.4258/hir.2019.25.2.89.
36. Sunitha EV, Samuel P. Automatic code generation from UML state chart diagrams. IEEE Access. 2019;7:8591–608. doi:10.1109/ACCESS.2018.2890791.
37. Apostol D, Rusovan P, Marcu M. UML to code, and code to UML, a view inside implementation challenges and cost. In: Proceedings of the 2022 26th International Conference on System Theory, Control and Computing (ICSTCC); 2022 Oct 19–21; Sinaia, Romania. p. 140–5. doi:10.1109/ICSTCC55426.2022.9931871.
38. Silberschatzh A, Galvin PB, Gagne G. Operating system concepts. 10th ed. Vol. 11. Hoboken, NJ, USA: Laurie Rosatone; 2018.