Nottingham Trent University

</NTU>

SOFT40171 - Coursework March 2025

Design & Dev of Secure Systems

Name – Karunakar Reddy Machupalli

Student ID - N1334679

1. System Specification

1.1 Informal Specification

NivX is a web-based car rental platform engineered to deliver users a seamless, secure, and efficient experience for exploring, reserving, and managing car rentals. The platform seeks to fulfil consumer requirements and organizational goals by maintaining a balance among functionality, usability, and security. NivX integrates essential capabilities such user registration, vehicle search, reservation administration, and secure payment processing, while complying with rigorous security standards and best practices.

The informal specification of NivX delineates the functional and non-functional needs that characterize the system's activities. Functional requirements delineate the fundamental features and tasks the platform will execute, including user authentication, booking management, and transaction facilitation. Non-functional criteria pertain to overarching system attributes such as security, performance, dependability, and scalability, guaranteeing that the platform remains resilient, user-centric, and secure under diverse conditions.

The table below presents a systematic summary of these criteria, outlining the scope and aims of NivX's fundamental features and the secure methods that will be instituted to improve the platform's reliability, usefulness, and defense against potential cyber threats.

Category	Requirement
Functional Requirements	
User Registration and Authentication	**- Allow users to create accounts with name, email, and password. - Implement secure login with password hashing and session management. - Provide password recovery functionality.**
Browsing and Searching Cars	**- Enable browsing of cars by category (SUV, sedan, etc.) and searching with filters (e.g., price, availability, features). - Display essential car details like rental rates and specifications.**
Booking and Extras Selection	**- Allow users to book cars by specifying rental dates. - Provide options to add additional services (e.g., GPS, insurance). - Allow review of booking details before confirmation.**
Payment Processing	**- Implement secure payment processing with a payment gateway. - Support multiple payment methods and transaction logs. - Provide confirmation for successful transactions.**
Account and Booking Management	**- Allow users to manage profiles, update details, and change passwords. - Enable viewing, modifying, or canceling bookings based on policies.**
Non-Functional Requirements	

Security Requirements	**- Adhere to secure design principles (e.g., least privilege, input validation). - Implement protection against SQL injection, XSS, and CSRF vulnerabilities.**
Performance Requirements	**- Ensure the system supports concurrent users without performance degradation. - Optimize page load times and database query performance.**
Reliability and Availability	**- Minimize downtime with fault-tolerant mechanisms. - Implement error handling for system stability during unexpected issues.**
Usability Requirements	**- Design an intuitive and user-friendly interface. - Provide accessibility features for users with diverse needs.**
Maintainability and Scalability	**- Ensure modular design for easy maintenance and future updates. - Allow the system to scale to accommodate growing user demand.**

1.2 Importance of Formal Specification

Formal specification is essential for the development of the NivX car rental system, since it offers a mathematically rigorous framework for delineating system behaviors, validating security characteristics, and resolving ambiguities inherent in informal descriptions. In contrast to informal specifications, which may be subject to interpretation, formal approaches provide precision, facilitate early error identification, and validate the system's functionality and security compliance throughout distinct phases of the **“system development life cycle (SDLC).” (Centers for Medicare & Medicaid Services, n.d).**

1.2.1 Elimination of Ambiguity

Informal specifications may occasionally result in misinterpretations, particularly when articulating intricate system behaviors or security needs. Formal specification employs precise syntax and semantics to deliver a clear, unambiguous depiction of NivX’s operations, guaranteeing that all stakeholders, including developers, testers, and auditors, possess a unified understanding of the system's intended functionality.

1.2.2 Enhanced Precision and Accuracy

Formal approaches enhance the design process by quantitatively modelling system actions, transitions, and restrictions with precision. This accuracy facilitates precise modelling of essential NivX functions, like safe authentication, reservation processes, and payment management, thereby diminishing the probability of implementation problems.

1.2.3 Early Detection of Design Flaws

Formal specification facilitates the early identification of potential design faults, inconsistencies, and security vulnerabilities within the Software Development Life Cycle (SDLC). Problems like race situations, deadlocks, or incorrect state transitions can be detected and addressed during the specification phase, avoiding expensive corrections in subsequent development stages.

1.2.4 Verification of System Properties

The NivX system can be formally validated against specified properties to guarantee its intended behavior under all scenarios. Petri nets can model the system's concurrent and sequential processes, while verification techniques such as reachability analysis can prevent deadlocks and security vulnerabilities.

1.2.5 Adherence to Security Requirements

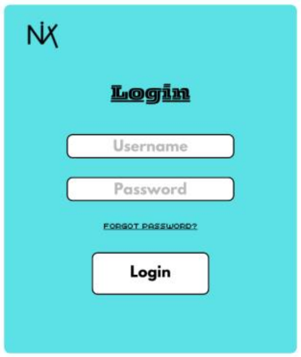
Formal methods are essential for proving that NivX complies with fundamental security concepts, including confidentiality, integrity, and availability (CIA). Formal specification aids in validating that the system adheres to security standards and mitigates dangers such as unauthorized access, data breaches, or process conflicts by modelling access control policies, data flow, and user interactions.

2. System Design

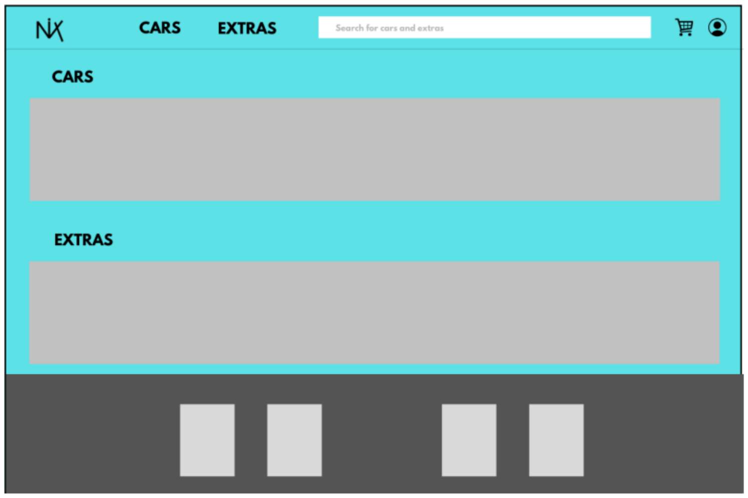
2.1 Wireframes

Wireframes show the structural configuration of essential pages, showing the arrangement of features like buttons, forms, and navigation links. Their emphasis is on human engagement, offering a visual roadmap for the implementation phase while highlighting usability, functionality, and security elements.

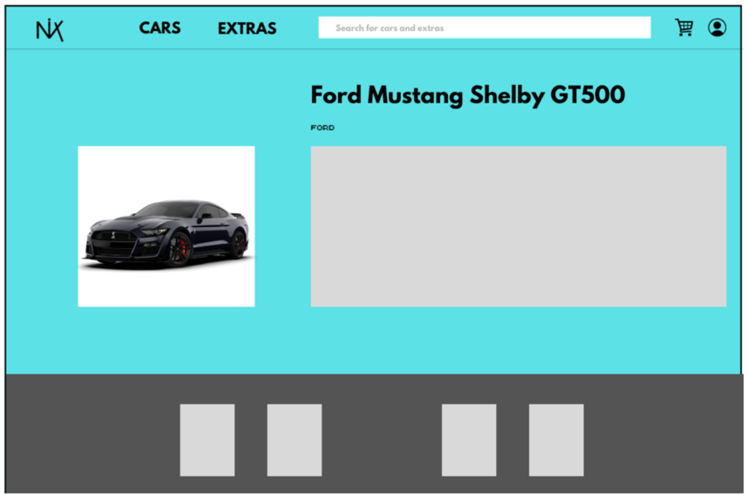
2.1.1 Login Page



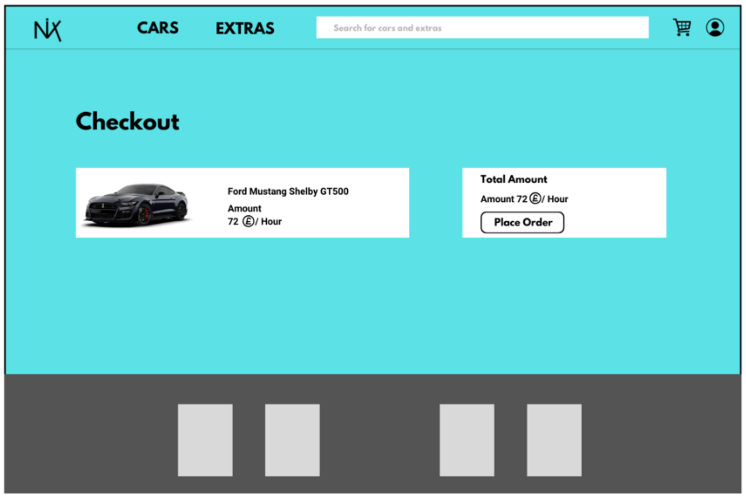
2.1.2 Car Search and Browse Page



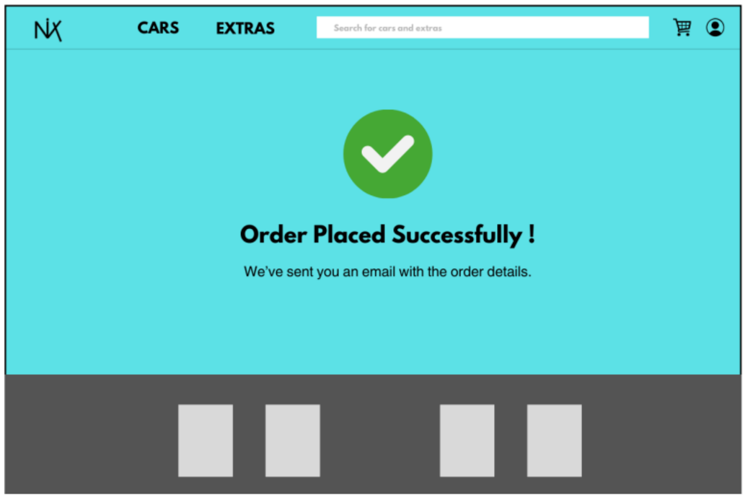
2.1.3 Content Description page



2.1.4 Ordered List page



2.1.5 Order confirm page

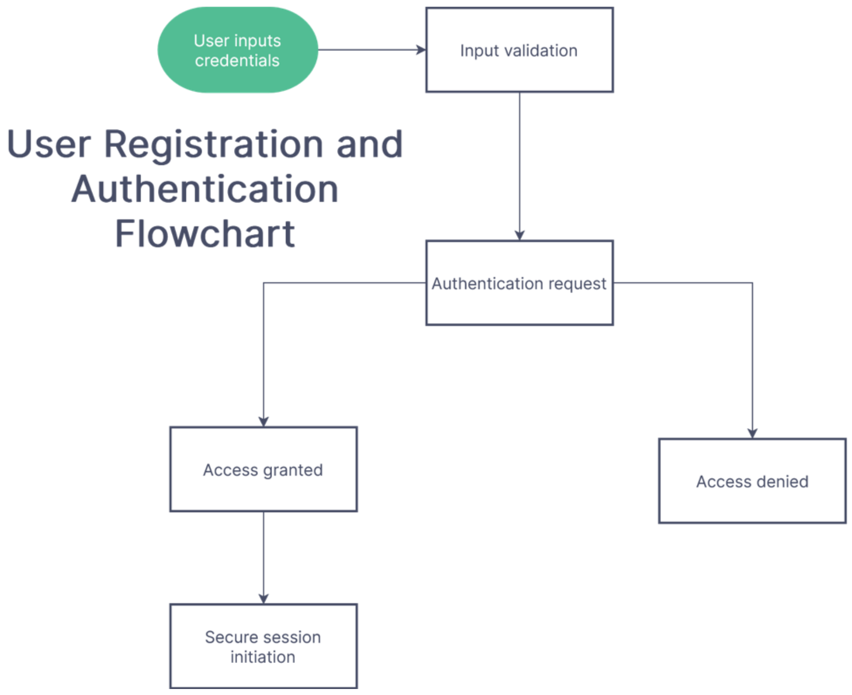


Order Placed Successfully !

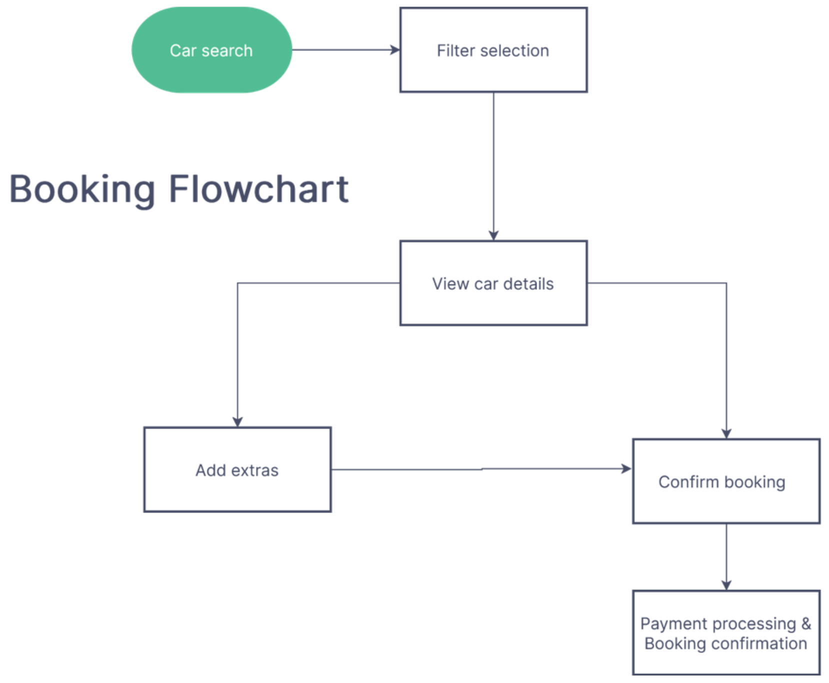
We've sent you an email with the order details.

2.2 Flowcharts

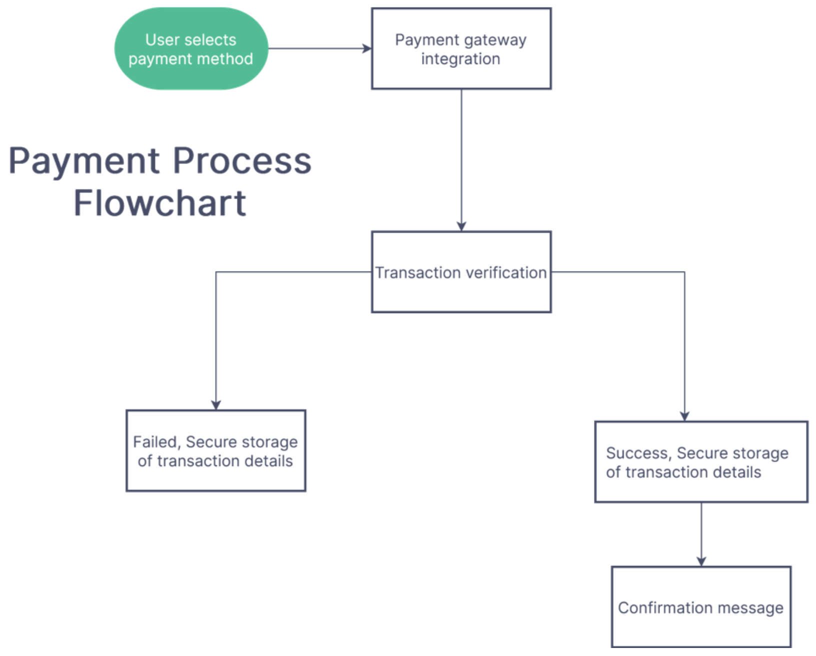
2.2.1 User Registration and Authentication Flowchart



2.2.2 Booking Flowchart



2.2.3 Payment Process Flowchart


User selects payment method
Payment gateway integration
Payment Process Flowchart
Transaction verification
Failed, Secure storage of transaction details
Success, Secure storage of transaction details
Confirmation message

2.2 Secure Design Principles

The NivX car rental system has been carefully crafted to comply with essential secure design principles that improve the platform's overall security posture. NivX seeks to reduce risks, minimize vulnerabilities, and guarantee the confidentiality, integrity, and availability (CIA) of user data and system capabilities by embedding these principles into the fundamental architecture of the system.

"The notion of least privilege has been used to limit access permissions for users" (Saltzer and Schroeder, 1975)., processes, and systems, ensuring they can only access the information and functions essential for their responsibilities. This mitigates the potential consequences of compromised accounts or insider threats by diminishing the attack surface. Administrative powers are restricted to authorized people, while regular users are permitted access just to customer-facing functionalities, like browsing vehicles, making reservations, and managing their profiles. This meticulous access control implementation mitigates privilege escalation attacks and fortifies system security.

Alongside the idea of least privilege, the concept of defense in depth has been integrated to establish a layered security strategy. This entails implementing numerous security measures at various system levels to guarantee redundancy and robustness against diverse cyber-attacks. Even if a single security layer is breached, the subsequent layers offer supplementary protection. NivX does this by the implementation of multi-factor authentication (MFA) for user access, the use of input validation to avert injection attempts, the encryption of vital information, and the safeguarding of network connection over HTTPS. This comprehensive technique enhances the platform's resilience against complex and multi-faceted attacks.

The principle of fail-safe defaults guarantees that the system maintains security during failures or unforeseen occurrences. Consistent with this principle, NivX is designed to immediately limit access upon authentication failure, rather than bestowing undesired privileges. User sessions are configured to expire after a duration of inactivity, thus mitigating the risk of session hijacking and unauthorized access. NivX mitigates the risk of security breaches stemming from system misconfigurations or failures by defaulting to a secure state.

Another fundamental idea applied to the system is secure by design, which underscores the incorporation of security issues from the initial phases of the development life cycle. NivX mitigates the risk of introducing vulnerabilities during implementation by proactively incorporating security measures in the design phase. This is accomplished by following secure code standards, crafting secure database queries to avert SQL injection, and executing secure session management to safeguard user data and uphold session secrecy.

Moreover, input validation and sanitization are employed across the platform to avert harmful input from undermining the system's security. All user inputs, including those entered through login forms, search filters, and booking pages, are checked to verify adherence to prescribed formats and to reject potentially malicious scripts. This reduces the possibility of injection attacks, including SQL injection and cross-site scripting (XSS), and preserves the integrity and reliability of the data processed by the system.

The notion of security by default guarantees that NivX's initial configurations emphasize security, necessitating users or administrators to intentionally modify settings to diminish security levels. HTTPS is enabled by default for all user sessions, and superfluous services, APIs, and open ports are disabled to minimize the attack surface and avert inadvertent exposure of sensitive information.

Incorporating these secure design concepts into NivX's architecture establishes a formidable security framework that safeguards user data, deters unauthorized access, and improves overall system reliability. This thorough, principle-based strategy guarantees that NivX remains resilient against advancing cyber threats while preserving a fluid, user-friendly experience.

2.3 Additional Security Mechanisms

To bolster the security of NivX and ensure compliance with the principles of confidentiality, integrity, and availability (CIA), various supplementary security methods are instituted, as outlined in the assignment checklist. These techniques guarantee data protection, safeguard user interactions, and maintain system reliability. Access control implements role-based restrictions to limit users to permitted functionality and data. Customers can oversee their reservations and profiles, whilst administrative users possess access to backend management functionalities, guaranteeing that unauthorized individuals cannot elevate their capabilities. This restricts the attack surface and maintains data confidentiality. Encryption safeguards sensitive information during transmission and storage. TLS (Transport Layer Security) encrypts user credentials, payment information, and other data exchanged between users and the server to prevent eavesdropping and man-in-the-middle (MITM) attacks. Sensitive information, including hashed passwords, is encrypted to mitigate the danger of data breaches, thereby safeguarding confidentiality and integrity. Session management guarantees safe user authentication through the encryption of session tokens and the use of secure cookie attributes (HTTP Only and safe). Sessions are equipped with timeouts that automatically log out users after a period of inactivity to mitigate the risk of session hijacking. These steps improve session secrecy and reduce the dangers of unauthorized access. Moreover, input validation and sanitization safeguard the system against injection-based assaults such as SQL injection and cross-site scripting (XSS). By checking user inputs at each stage, the platform guarantees data integrity and prevents malicious payloads from undermining system functionality. These strategies mitigate critical vulnerabilities by implementing various security measures, consistent with defense-in-depth concepts. They enhance access control, safeguard critical data, and ensure secure session management, consequently augmenting the system's overall security, reliability, and adherence to secure development best practices.

3. Development Process

3.1 Implementation

The NivX car rental system is developed using a systematic procedure that complies with secure coding standards and norms. The development process commences with planning, during which the project's scope and objectives are outlined. Essential features and functionalities have been established, and a project timeline has been established to deploy resources efficiently.

In the requirements analysis phase, user needs are collected and examined to delineate both functional and non-functional requirements. These standards are guaranteed to be explicit, succinct, and verifiable. The design phase entails the development of wireframes and flowcharts to illustrate the system's architecture and progression. Principles of secure design, including least privilege, defense in depth, and fail-safe defaults, are implemented. The database schema is structured to guarantee secure data management.

Input validation is an essential component of secure programming. All user inputs undergo validation through regular expressions to mitigate injection attacks, including SQL injection and cross-site scripting (XSS). Error management is executed to address unforeseen problems tactfully, preventing the disclosure of sensitive information in error messages.

Authentication and authorization procedures are employed to provide secure user verification. Password hashing and multi-factor authentication are employed to augment security. Role-based access control is utilized to limit access to sensitive capabilities, guaranteeing that only authorized individuals may execute administrative tasks.

Session management constitutes a vital component of security. Secure session tokens are utilized and communicated via HTTPS to guarantee data encryption. Session timeouts are instituted to reduce the danger of session hijacking.

Data encryption is utilized to safeguard sensitive information during transmission and storage. Robust encryption techniques, such as AES, are employed, and encryption keys are controlled with security. Logging and monitoring are established to record security-related events, including login attempts and data access, for auditing reasons. Logs are systematically monitored to identify and address security incidents.

Access control measures are implemented to restrict user permissions according to their roles. Input validation and sanitization are implemented for all user inputs to avert injection attacks. Secure communication is achieved through the utilization of HTTPS to encrypt data exchanged between the client and server. Secure APIs are established for intercommunication among various system components.

Database security is upheld by the utilization of parameterized queries to avert SQL injection. Access to the database is limited to authorized users and applications. Systematic code evaluations are performed to detect and rectify security issues. Thorough testing, encompassing unit tests, integration tests, and security tests, is conducted to guarantee the system's robustness and reliability.

By implementing these measures and conforming to safe coding standards, the NivX automobile rental system is designed to be resilient, secure, and dependable. This methodical approach guarantees that the system adheres to the necessary security standards while delivering a seamless, safe, and efficient user experience.

3.2 Implementation in Code

3.2.1 Input Validation

To avert injection threats like SQL injection and cross-site scripting (XSS), all user inputs must be checked with regular expressions. This guarantees that only permissible input formats are processed by the system.

Example in **contentDetails.js**

```
function validateInput(input) {
    const regex = /^[a-zA-Z0-9\s]+$/;
    return regex.test(input);
}
```

3.2.2 Error Handling

Comprehensive error handling must be established to address unforeseen problems effectively. This mitigates the disclosure of sensitive information in error messages.

Example in **orderPlaced.js**

```
httpRequest.onreadystatechange = function() {
    if (httpRequest.readyState == 4) {
        if (httpRequest.status == 200) {
            // Handle success
        } else {
            console.error("Error: " + httpRequest.statusText);
        }
    }
};
```

3.2.3 Authentication and Authorization

Robust user authentication protocols must be established, including password hashing and multi-factor authentication. Role-based access control must be implemented to limit access to sensitive functionalities.

Example in **orderPlaced.js**

```
document.cookie = "orderId=" + 0 + ",counter=" + 0;
```

3.2.4 Data Encryption

Sensitive data must be encrypted during transmission and storage utilizing robust encryption methods such as AES. Encryption keys must be managed with utmost security.

Example in **orderPlaced.js**

```
function encryptData(data) {
    // Use AES encryption
    return CryptoJS.AES.encrypt(data, "secretKey").toString();
}
```

3.2.5 Access Control

Role-based access control must be established to restrict user permissions according to designated roles. This guarantees that administrative functions are limited to authorized users.

Example in **contentDetails.js**

```
function checkUserRole(role) {
    if (role !== "admin") {
        alert("Access denied");
        return false;
    }
    return true;
}
```

3.2.6 Secure Communication

HTTPS must be employed to encrypt data exchanged between the client and server. Secure APIs must be established for communication among various system components.

Example in **orderPlaced.js**

```
httpRequest.open("POST", "https://secure-api.example.com/order", true);
```

3.2.7 Database Security

Utilize parameterized queries to mitigate SQL injection risks. Access to the database must be limited to authorized users and applications.

Example in **orderPlaced.js**

```
httpRequest.open("POST", jsonRequestURL, true);
httpRequest.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

3.2.8 Code Review and Testing

Systematic code reviews must be performed to detect and rectify security flaws. Thorough testing, encompassing unit tests, integration tests, and security tests, must be conducted to guarantee the system's robustness and reliability.

Example in **content.js**

```
function runTests() {
    // Unit tests
    console.assert(validateInput("test"), "Input validation failed");
    // Integration tests
    console.assert(httpRequest.status === 200, "AJAX request failed");
}
```

3.3 Compliance with Standards

It is essential that the development of the NivX car rental system adheres to recognized secure development life cycle (SDLC) standards and procedures to mitigate vulnerabilities and improve the system's overall security posture. This section delineates the principal criteria and methods adhered to during the development process.

In the planning and requirements analysis phase, security requirements are defined and documented, encompassing access restrictions, data protection measures, and adherence to pertinent standards such as **"GDPR" (European Union, 2016).** and **"CCPA" (State of California, 2018).** A risk assessment is performed to detect potential security threats and vulnerabilities, prioritizing risks according to their impact and probability. The system design phase entails the implementation of secure design concepts, including least privilege, defense in depth, and fail-safe defaults, so enabling the integration of security from the beginning of the design process. Threat modelling is conducted to discover potential attack routes and devise countermeasures, utilizing technologies such as the Microsoft Threat Modelling Tool to visualize and analyze threats.

During the implementation phase, secure coding practices are adhered to, encompassing the validation and sanitization of all user inputs to avert injection attacks. Systematic code reviews are performed to uncover and rectify security vulnerabilities. Automated testing is utilized within continuous integration and continuous deployment (CI/CD) pipelines, guaranteeing that security assessments are included in the automated test suite.

During deployment, the system is configured securely by establishing secure defaults, deactivating superfluous services, and implementing security fixes. The deployment environment is fortified by the implementation of network security measures, including firewalls and intrusion detection systems, as well as the safeguarding of cloud infrastructure. Maintenance and monitoring entail executing a patch management protocol to guarantee the timely application of security patches, vigilance for emerging vulnerabilities, and subsequent system updates. A comprehensive incident response strategy is established and updated to manage security issues efficiently, with periodic drills executed to guarantee the team's readiness to address security breaches.

Secure programming practices encompass input validation and sanitization, comprehensive error handling, secure user authentication protocols, role-based access control, secure session management, data encryption, logging and monitoring, secure communication, database security, and routine code review and testing. By conforming to these secure development life cycle standards and safe programming methodologies, the NivX automobile rental system is designed to be resilient, secure, and dependable. This methodical approach guarantees that the system adheres to the necessary security standards while delivering a seamless, safe, and efficient user experience.

3.4 Documentation

3.4.1 Development Process Documentation

The creation of the NivX automobile rental system utilized JavaScript, HTML, and CSS to construct a safe, efficient, and user-friendly website. The system was engineered to offer consumers a smooth experience for discovering, reserving, and managing car rentals. A SQL database and **“MockAPI” (MockAPI, 2025).** were employed to replicate backend interactions, facilitating the testing and validation of many functionalities.

3.4.2 Rationale for Coding Decisions

The coding choices during the development of the NivX automobile rental system were motivated by the necessity to provide a secure, efficient, and user-friendly platform. Every option was meticulously evaluated to guarantee that the system satisfies both functional and non-functional criteria while complying with secure coding standards.

Input Validation

To avert injection threats, including SQL injection and cross-site scripting (XSS), it is imperative to validate all user inputs. This guarantees that only permissible input formats are processed by the system. Regular expressions were employed to delineate permissible input formats, and input validation procedures were incorporated into the code.

Error Handling

Effective error handling is essential to address unforeseen problems tactfully and safeguard sensitive information from being revealed in error messages. Error handling techniques were incorporated into the code, especially in AJAX requests, to track problems and deliver constructive feedback while safeguarding sensitive information.

Authentication and Authorization

Robust user authentication methods, like password hashing and multi-factor authentication, are essential for safeguarding user accounts. Role-based access control guarantees that only authorized individuals are permitted to execute sensitive operations. Authentication and authorization protocols were executed via cookies and session management strategies.

Data Encryption

Encrypting sensitive data during transmission and storage safeguards it from unauthorized access and maintains data integrity. Robust encryption algorithms, including AES, were employed to secure data, and encryption keys were kept with utmost security.

Secure Communication

Employing HTTPS to encrypt data exchanged between the client and server safeguards against interception and tampering. The httpRequest.open function was altered to utilize HTTPS for secure communication.

Database Security

Utilizing parameterized queries mitigates SQL injection vulnerabilities and guarantees that database access is confined to authorized users and applications. The httpRequest.open and httpRequest.setRequestHeader methods were revised to utilize parameterized queries and establish suitable content types.

Code Review and Testing

Systematic code evaluations and thorough testing facilitate the detection and rectification of security flaws, hence maintaining the system's resilience and dependability. Functions for code review and testing were incorporated to validate input and assess AJAX queries.

Phase/Practice	Description

Input Validation and Sanitization	All user inputs validated using regular expressions; sanitized to remove malicious content.
Error Handling	Robust mechanisms are implemented to handle errors gracefully and avoid exposing sensitive information.
Authentication and Authorization	Secure password hashing, multi-factor authentication, and role-based access control used to restrict access.
Session Management	Secure session tokens transmitted over HTTPS; session timeouts implemented to prevent hijacking.
Data Encryption	Sensitive data encrypted in transit and at rest using AES; encryption keys securely managed.
Logging and Monitoring	Security events logged for auditing; logs monitored regularly to detect and respond to incidents.
Secure Communication	HTTPS is used for client-server communication; secure APIs implemented for system components.
Database Security	Parameterized queries to prevent SQL injection; database access restricted to authorized users/applications.
Code Review and Testing	Regular code reviews and comprehensive testing (unit, integration, security) to ensure robustness and reliability.
Planning and Requirements Analysis	Security requirements documented (GDPR, CCPA compliance); risk assessment conducted to prioritize threats.
System Design	Applied secure design principles (least privilege, defense in depth, fail-safe defaults); threat modeling with tools like Microsoft Threat Modeling Tool.
Implementation	Secure coding practices followed; inputs validated/sanitized; static analysis with tools.
Testing	Vulnerability scanning, penetration testing
Deployment	Secure configuration with defaults, disabled unnecessary services, applied patches; hardened environment with firewalls and IDS.
Maintenance and Monitoring	Patch management process; system monitored for vulnerabilities; incident response plan with regular drills.

4. Testing and Analysis

4.1 Testing Techniques

A comprehensive testing approach is undertaken to assure the security, reliability, and functionality of the NivX automobile rental system. This encompasses the application of both static and dynamic analytic methodologies to detect vulnerabilities, enhance performance, and guarantee adherence to security requirements.

4.1.1 Static Analysis

Static analysis is employed to evaluate the code without its execution. This facilitates the early detection of potential defects in the development process and improves code quality.

The subsequent static analysis methodologies are employed:

Code Inspection

Manual code reviews are performed to identify vulnerabilities, including unsecured API utilization, inadequate input validation, and possible SQL injection issues. Tools are employed to automate the identification of coding errors, poor practices, and security vulnerabilities.

Data Flow Analysis

This technique examines the flow of data throughout the system, guaranteeing the secure management of critical information during storage, processing, and transmission. This inhibits the disclosure of sensitive information.

Control Flow Analysis

Control flow analysis assesses the logical execution paths within the code, verifying that all potential branches (conditional statements, loops) function as intended. This assists in identifying inaccessible code, infinite loops, and inadequate exception handling.

4.1.2 Dynamic Analysis

Dynamic analysis is performed to assess the system's behavior during execution. It reveals vulnerabilities that may not be apparent during static analysis.

The subsequent dynamic testing methodologies are implemented:

Vulnerability Scanning

Instruments that include OWASP ZAP and Burp Suite are employed to examine the system for prevalent vulnerabilities, including XSS, SQL injection, and CSRF. These scans assist in identifying vulnerabilities that adversaries may exploit.

Penetration Testing

Penetration testing emulates actual attack scenarios to evaluate the system's robustness. It addresses vulnerabilities at both the application and infrastructure levels, hence enhancing the platform's defenses.

Unit and Integration Testing

Automated unit and integration tests are performed within a CI/CD pipeline to verify that individual components and their interactions function as intended. This assessment encompasses both functional and security dimensions.

Load and Stress Testing

The system undergoes testing under diverse load situations to assess its performance, scalability, and stability. This ensures that NivX can manage substantial volumes of simultaneous users without deterioration.

4.2 Coverage

Ensuring thorough test coverage is crucial to verify the functionality, security, and performance of the NivX car rental system. A systematic methodology is utilized to attain extensive coverage via path testing, edge testing, and statement testing, which together guarantee that all essential system functionalities are comprehensively assessed and verified.

Path Testing

Path testing emphasizes the analysis of all potential paths within the system's control flow to identify problems across various execution sequences. This guarantees that each path is traversed at least once during testing. Evaluates the authentication procedure, encompassing valid, invalid, and null input scenarios. Verifies the automobile selection, supplementary additions, and payment processing sequence; ensures the search capability operates effectively with diverse filters, varying input lengths, and edge cases.

Edge Testing

Edge testing emphasizes boundary conditions to assess the system's performance when handling data at or near its limitations. This technique reveals problems associated with inadequate management of edge cases. Evaluating the maximum and minimum lengths of input fields. Assessing the functionality of adding the maximum quantity of products to the cart. Ensuring the appropriate management of empty or null input values to prevent system failure. Verifying system responses for boundary circumstances, including dates adjacent to rental cutoffs and price thresholds.

Statement Testing

Statement testing guarantees that each line of code is performed a minimum of once to identify unreachable code, dead code, and flawed logic implementation. Automated instruments are employed to monitor and enhance statement coverage. Evaluating both affirmative and negative branches in conditional assessments. Verifying correct loop execution, encompassing zero iterations and numerous iterations in product listing, cart management, and order processing. Ensuring that API requests accommodate various response types.

4.3 Problem Resolution

The testing phase of the NivX car rental system revealed multiple issues, such as functional faults, security vulnerabilities, and performance bottlenecks. Every issue was meticulously examined, and solutions were executed to improve system security, functionality, and efficiency. The subsequent table delineates the principal difficulties, their ramifications, and the implemented fixes, which conform to secure coding techniques to alleviate vulnerabilities and enhance the overall resilience of the platform.

Issue	Impact	Solution	Aligned Secure Practice
Cross-Site Scripting (XSS)	**Unvalidated and unsensitized user-generated data left the system vulnerable to XSS attacks.**	**Implemented input and output sanitization using DOMPurify to prevent injection of malicious scripts.**	**Output sanitization to prevent XSS and ensure secure DOM updates.**
Insecure Cookie Handling	**Cookies storing sensitive data (e.g., cart information) were set without secure attributes, making them vulnerable to theft or tampering.**	**Set cookies with Secure, HttpOnly, and SameSite attributes to enhance cookie security.**	**Secure cookie handling to prevent session hijacking and CSRF attacks.**
Synchronous XMLHttpRequest Calls	**Synchronous API calls caused UI blocking and potential race conditions in asynchronous workflows.**	**Replaced with asynchronous requests using fetch API to improve efficiency and prevent UI freezing.**	**Asynchronous processing to avoid race conditions and improve performance.**
Improper API Response Handling	**Unvalidated API responses increased the risk of injection vulnerabilities and unexpected errors.**	**Added API response validation to ensure that only properly formatted and expected data is processed.**	**Input validation to ensure data integrity and prevent injection risks.**

Lack of Content Security Policy (CSP)	**Absence of CSP left the system open to XSS and resource injection attacks.**	**Added a strict CSP to control the sources of scripts, images, and other resources.**	**CSP implementation to reduce XSS risk and control external resources.**
Improper Query Parameter Handling	**Query parameters were parsed without validation, risking open redirects and injection attacks.**	**Sanitized and validated query parameters to ensure they are safe and correctly formatted.**	**Input validation to prevent open redirects and injection vulnerabilities.**
Hardcoded API URLs	**Hardcoded API URLs reduced flexibility and risked exposure of sensitive data in source code.**	**Moved API URLs to environment variables to support secure configurations and prevent accidental exposure.**	**Secure API configuration to enhance deployment flexibility.**
Lack of Authentication on API Requests	**API requests lacked proper authentication, risking unauthorized access to backend resources.**	**Implemented API key authentication to secure API requests and prevent unauthorized access.**	**Authentication to secure backend communications.**
Unvalidated User Inputs in Search Bar	**User inputs were processed without validation, leaving the search functionality vulnerable to injection attacks.**	**Added input validation to ensure only valid inputs are accepted and processed.**	**Input validation to prevent injection attacks and improve security.**

5. Formal Methods Application

5.1 Behavioral Modeling with Petri Nets

5.1.1 Petri Nets

“Petri Nets are a mathematical tool utilized for modelling, analyzing, and verifying systems,” (Heiner et al., 2012). especially those that encompass concurrency, synchronization, and intricate control flows. Petri Nets offer a formal framework for the NivX car rental system to depict sequential and concurrent processes, guaranteeing that system behavior complies with security and functionality requirements. This part formulates a Petri Net model to encapsulate essential activities inside the NivX system, emphasizing state transitions, synchronization points, and safe data management.

5.1.2 Model-1 Cookie State Management for Cart Updates

Objective

Modelling the flow of cookie-based cart updates within the NivX system to ensure appropriate session management and mitigate risks such as cookie manipulation or invalid counters.

Relevant Code

```
if (document.cookie.indexOf(",counter=") >= 0) {
  let counter = document.cookie.split(",")[1].split("=")[1];
  document.getElementById("badge").innerHTML = counter;
}

document.cookie = "orderId=" + order + ",counter=" + counter;
```

Petri Net Design

Places:

- P1: Cart is Empty (counter = 0)
- P2: Item Added to Cart (Counter incremented)
- P3: Cart with Items
- P4: Checkout Complete (Counter reset to 0)

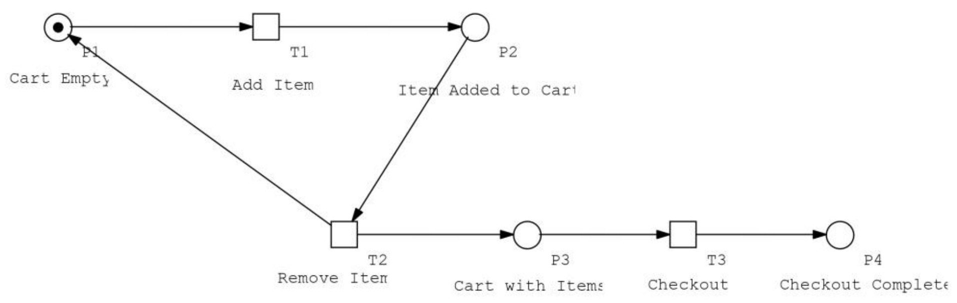
Transitions:

- T1: Add item to cart (Increments the cookie counter)
- T2: Remove item from cart (Decrements the counter)
- T3: Checkout (Counter reset and items removed)

Tokens Flow Explanation

The token transitions from P1 to P2 upon the user adding an item to the cart (T1: Add Item is triggered). Upon acquiring a token, subsequent actions (such as adding additional products) can transfer the token to P3, symbolizing a cart with increased contents. The token transitions from P3 to P4 upon the activation of T3 (Checkout), signifying the completion of the cart clearance and checkout process.

Petri Net Model



5.1.3 Model-2 Asynchronous API Call Handling and Response Processing

Objective

Designing the API call flow and error management to guarantee valid request-response cycles without obstructing the user interface.

Relevant Code

```
let httpRequest = new XMLHttpRequest();
httpRequest.onreadystatechange = function () {
  if (this.readyState === 4) {
    if (this.status == 200) {
      contentTitle = JSON.parse(this.responseText);
    }
  }
};
httpRequest.open("GET", "https://67dfd9577635238f9aab54df.mockapi.io/api/product/products",
true);
httpRequest.send();
```

Petri Net Design

Places:

- P1: Request Not Sent – Initial state where no API request has been sent.
- P2: Request Sent and Awaiting Response – The system has sent an API request and is waiting for a response.
- P3: Response Received (Success) – The system has received a successful response (200 status).
- P4: Response Received (Error) – The system has received an error response (404, 500).
- P5: Retry Request – The system retries the API call after receiving an error.
- P6: Request Completed – The request process is finished (after success or retry completion).

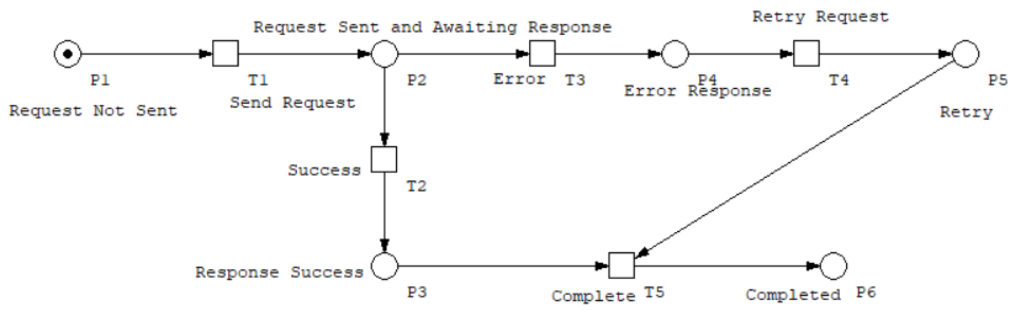
Transitions:

- T1: Send API Request – Triggers the API request and moves the token from P1 to P2.
- T2: Receive Successful Response – Moves the token from P2 to P3 after receiving a successful response.
- T3: Receive Error Response – Moves the token from P2 to P4 if the response indicates an error.
- T4: Retry API Request – Moves the token from P4 to P5 and then back to P2 to retry the request.
- T5: Complete Request Process – Moves the token from P3 or P5 to P6, indicating the API call is complete.

Tokens Flow Explanation

The token initiates in P1 to indicate that no request has been dispatched. Upon the activation of T1, the token transitions from P1 to P2, signifying a request dispatched and pending a response. Upon successful request, T2 activates, and the token transitions from P2 to P3, signifying a successful API invocation. Upon request failure, T3 activates, transferring the token from P2 to P4 (Error Response). The system retries the request from P4. T4 initiates, transferring the token from P4 to P5 and subsequently returning it to P2 to reattempt the operation. Upon the completion of the request procedure (after success or retries), T5 is activated, and the token transitions to P6 (Completed).

Petri Net Model



5.1.4 Model-3 Cart Item Selection and DOM Updates

Objective

Modelling the cart DOM update process to provide secure, race-free DOM manipulations during the addition or modification of goods in the cart.

Relevant Code

```
function dynamicCartSection(ob, itemCounter) {
  let boxDiv = document.createElement("div");
  boxDiv.id = "box";
  boxImg.src = ob.preview;
  boxh3.innerHTML = ob.name + " × " + itemCounter;
}
```

Petri Net Design

Places:

- P1: No items rendered in the DOM
- P2: Item selected by user
- P3: Item added to cart
- P4: DOM updated after cart changes

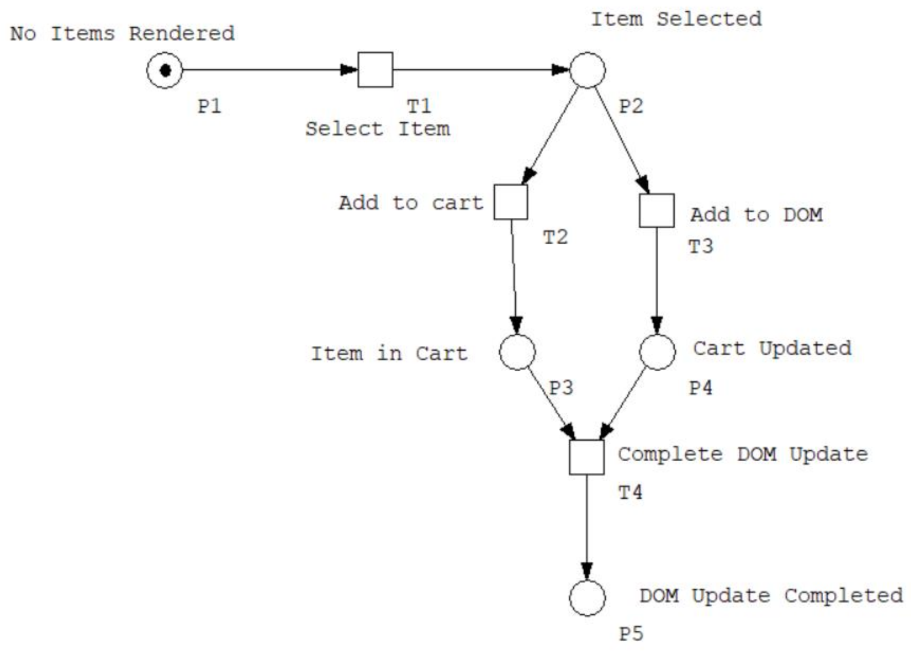
Transitions:

- T1: Select item (Triggers cart update)
- T2: Add item to cart (Update cart DOM)
- T3: Update DOM to reflect new cart state

Tokens Flow Explanation

One token starts in P1 (No Items Rendered). Upon the activation of T1 (Select Item), the token transitions from P1 to P2, signifying that an item has been selected. Upon the activation of T2, the token transitions from P2 to P3, signifying that the item has been incorporated into the cart and shown in the DOM. Upon the activation of T3, the token transitions from P3 to P4, indicating that the cart has been modified. Upon the activation of T4, the token transitions from P4 to P5, signifying the completion of the DOM change.

Petri Net Model



5.2 Verification

5.2.1 Verification Tool

“Charlie” (Heiner, Schwarick and Wegener, 2015). is a robust instrument intended for the formal verification and analysis of Petri Net models. It facilitates the creation and analysis of the Reachability Graph (RG), which delineates all potential states and transitions inside a system. This facilitates the analysis of concurrent system behavior and the verification of critical features such as reachability, safety, and liveness. We utilized Charlie to validate the three Petri Net models of the NivX system: Cookie State Management, Asynchronous API Call Handling, and Cart Item Selection with DOM Updates.

We initially exported the Petri Net models from Snoopy and subsequently imported them to Charlie for verification. We constructed the Reachability Graph (RG), which illustrates all possible sequences of state transitions (places and tokens) within the system. This graph enabled us to identify potential system pathways, detect deadlocks, and verify that all critical states were accessible. After computing the reachability graph, we employed Computation Tree Logic (CTL) to evaluate several features of the system.

CTL is a temporal logic framework that facilitates the formal specification of attributes in dynamic systems. It employs operators such as EF (there exists a path where a state is ultimately attained), AG (on all paths, a state is perpetually true), and AF (on all paths, a state will finally be achieved) to articulate system needs.

We established and validated 10 CTL features for each of the three models. These properties evaluated essential elements, including the accessibility of pivotal states, the accuracy of transitions, and the avoidance of enduring errors or stalemate conditions.

For example, we validated features such as EF(P2 >= 1) to ascertain that the system might ultimately attain the "Item Selected" state and AG(P4 == 0) to guarantee that the cart error state would not endure endlessly. We similarly evaluated liveness criteria, such as AF(P4 >= 1), to confirm that a system process, such as the completion of the DOM update, would occur. The verification results indicated that all specified properties were fulfilled, affirming that the NivX system models operate safely, prevent deadlocks, and uphold accuracy in concurrent settings.

By verifying with Charlie and CTL, we confirmed that the NivX system complies with essential safety and liveness standards, hence enhancing its overall reliability and security.

5.2.2 System Properties to Verify

Property Type	Explanation	CTL Syntax Example
Reachability	**Ensures that critical states can be reached.**	**EF(P3) – Eventually, P3 (Item in Cart) is reached.**
Safety	**Ensures that unsafe states are never reached.**	**AG(¬P4) – Always avoid P4 if it represents an error state.**
Liveness	**Ensures that certain actions or states will eventually occur.**	**AF(P5) – Always eventually reach P5 (DOM Updated).**
Deadlock-Free	**Verifies that no deadlock states exist.**	**¬EF(deadlock) – No state where the system gets stuck.**
Fairness	**Ensures that all processes get fair access to resources.**	**A(P3 → AF(P5)) – If the cart updates, the DOM update will eventually complete.**

5.2.3 Defining CTL Properties for Verification

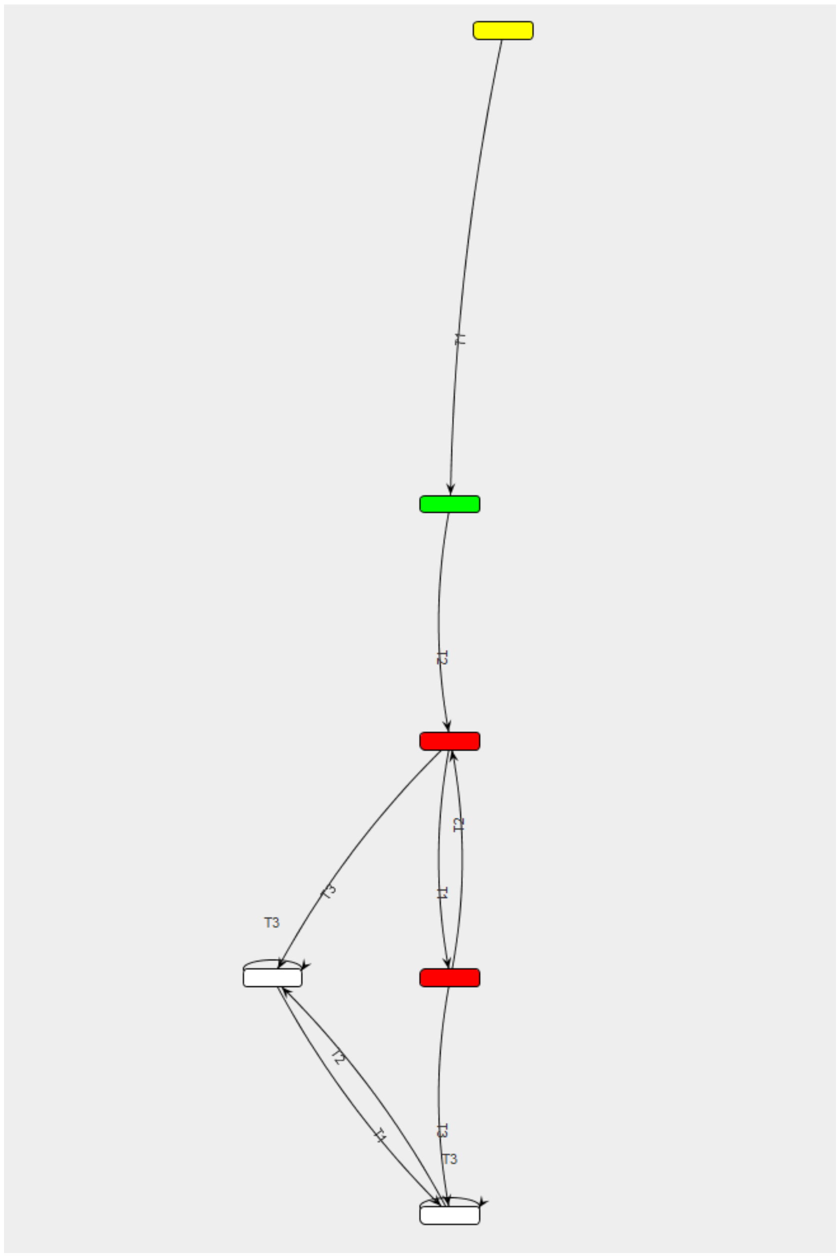
CTL Property	Description
1. EF(P2)	**Verify that the state where an item is selected is reachable.**
2. EF(P5)	**Ensure that the DOM update completion state is reachable.**
3. AG(¬P4)	**Ensure that the system does not get stuck in P4 (preventing an error state from persisting).**
4. ¬EF(deadlock)	**Verify that the system is free from deadlocks (no state where all tokens get stuck).**
5. A(P1 → AF(P2))	**If the cart is empty, it must eventually transition to the state where an item is selected.**
6. A(P2 → AF(P3))	**If an item is selected, the system must eventually add it to the cart.**

7. AF(P5)	**Ensure that the state where the DOM update is completed is always eventually reached.**
8. A(P3 → AF(P5))	**If the cart is updated, the DOM update must eventually complete.**
9. AG(P1 → ¬P4)	**Ensure that an empty cart state (P1) does not lead directly to an error state (P4).**
10. EF(T4)	**Verify that the transition for completing the DOM update (T4) is reachable.**

5.3 Reachability Analysis

5.3.1 Model-1 verification Cookie State Management for Cart Updates

Reachability Graph

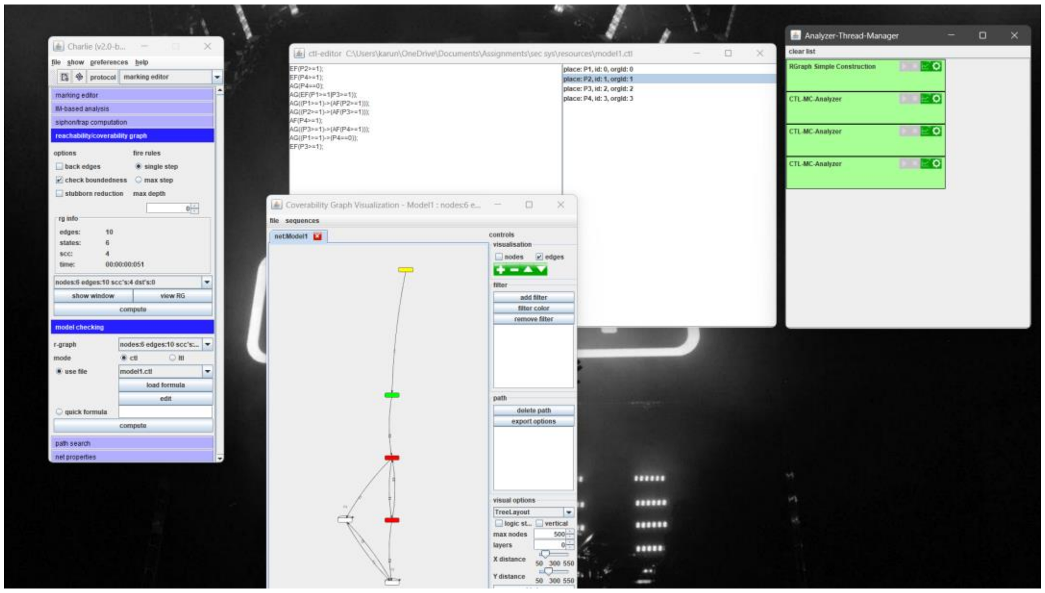


Test Analysis

CTL Property	Description	Result
EF(P2 >= 1)	**Item selection state is reachable.**	**Success**
EF(P4 >= 1)	**Cart cleared state (after checkout) is reachable.**	**Success**
AG(P4 == 0)	**Ensures that the cart error state is never persistently reached.**	**Success**
AG(EF(P1 >= 1))	**It is always possible to eventually reach the empty cart state, indicating safe cart handling.**	**Success**

AG((P1 >= 1) -> (AF(P2 >= 1)))	**If the cart is empty, the item selection state is eventually reached.**	**Success**
AG((P2 >= 1) -> (AF(P3 >= 1)))	**If an item is selected, it must eventually be added to the cart.**	**Success**
AF(P4 >= 1)	**Always eventually reach the cart cleared state after checkout.**	**Success**
AG((P3 >= 1) -> (AF(P4 >= 1)))	**If the cart is updated, it must eventually be cleared (after checkout).**	**Success**
AG((P1 >= 1) -> (P4 == 0))	**Ensure that the cart does not transition to an error state directly from the empty state.**	**Success**
EF(P3 >= 1)	**Cart updated state is reachable.**	**Success**

Screenshot

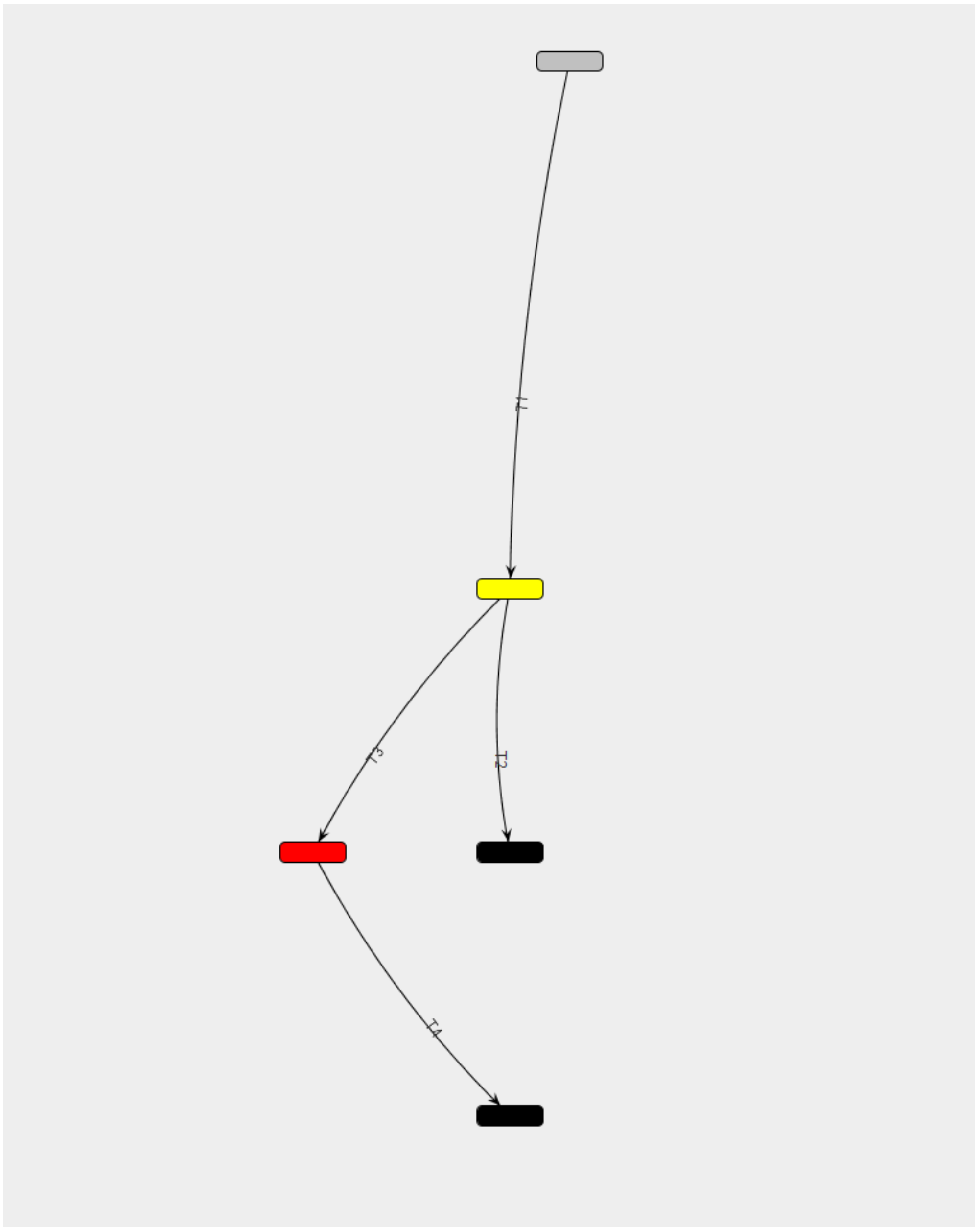

```
ctl-editor C:\Users\karun\OneDrive\Documents\Assignments\sec sys\resources\model1.ctl
EF(P2>=1);
EF(P4>=1);
AG(P4==0);
AG(EF(P1>=1|P3>=1));
AG((P1>=1)->(AF(P2>=1)));
AG((P2>=1)->(AF(P3>=1)));
AF(P4>=1);
AG((P3>=1)->(AF(P4>=1)));
AG((P1>=1)->(P4==0));
EF(P3>=1);

place: P1, id: 0, orgId: 0
place: P2, id: 1, orgId: 1
place: P3, id: 2, orgId: 2
place: P4, id: 3, orgId: 3
```

```
rg info
edges:   10
states:  6
scc:     4
time:    00:00:00:051
nodes:6 edges:10 scc's:4 dst's:0
```

5.3.2 Model-2 verification Asynchronous API Call Handling

Reachability Graph

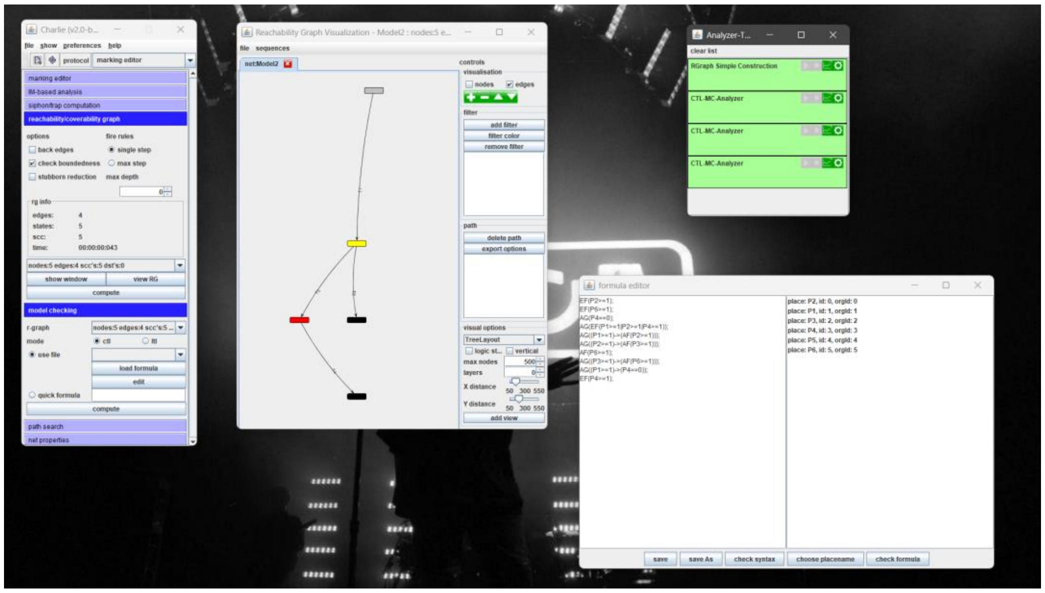


Test Analysis

CTL Property	Description	Result
EF(P2 >= 1)	Request sent state is reachable.	Success
EF(P6 >= 1)	Request completed state is reachable.	Success
AG(P4 == 0)	Ensures that the error response state is not reached persistently.	Success

AG(EF(P1 >= 1 | P2>= 1))	**It is always possible to reach a request initialization or retry state.**	**Success**
AG((P1 >= 1) -> (AF(P2 >= 1)))	**If the request has not been sent, it must eventually transition to the “sent” state.**	**Success**
AG((P2 >= 1) -> (AF(P3 >= 1)))	**If a request is sent, a successful response should eventually be received.**	**Success**
AF(P6 >= 1)	**Always eventually reach the request completion state.**	**Success**
AG((P3 >= 1) -> (AF(P6 >= 1)))	**If a successful response is received, the request must eventually be completed.**	**Success**
AG((P1 >= 1) -> (P4 == 0))	**Prevent transitioning to an error state directly from the initial state.**	**Success**
EF(P4 >= 1)	**Error response state is reachable.**	**Success**

Screenshots



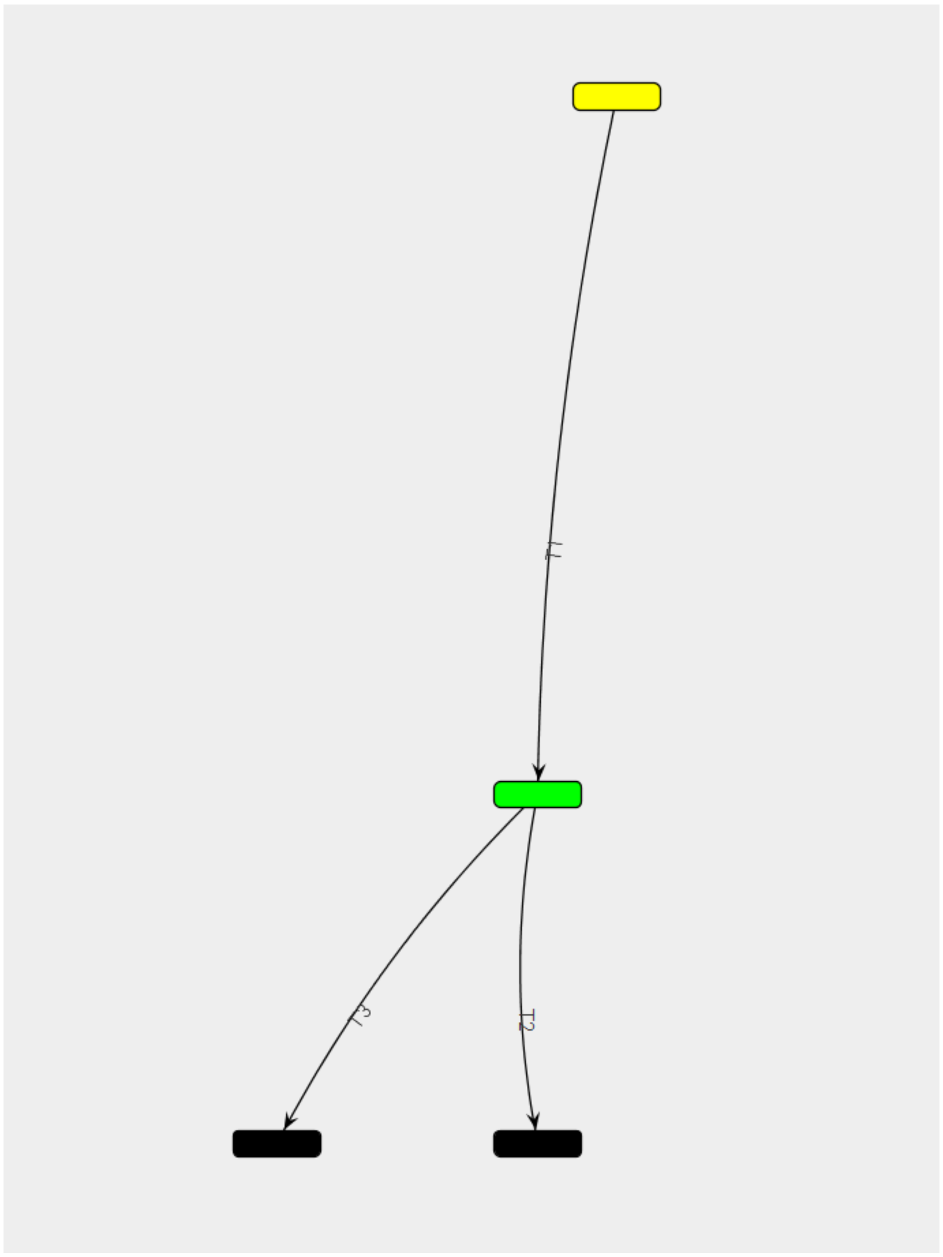
```
edges:  4
states: 5
scc:    5
time:   00:00:00:043
nodes:5 edges:4 scc's:5 dst's:0
```

```
EF(P2>=1);
EF(P6>=1);
AG(P4==0);
AG(EF(P1>=1|P2>=1|P4>=1));
AG((P1>=1)->(AF(P2>=1)));
AG((P2>=1)->(AF(P3>=1)));
AF(P6>=1);
AG((P3>=1)->(AF(P6>=1)));
AG((P1>=1)->(P4==0));
EF(P4>=1);
```

```
place: P2, id: 0, orgId: 0
place: P1, id: 1, orgId: 1
place: P3, id: 2, orgId: 2
place: P4, id: 3, orgId: 3
place: P5, id: 4, orgId: 4
place: P6, id: 5, orgId: 5
```

5.3.3 Model-3 verification Cart Item Selection and DOM Updates

Reachability Graph

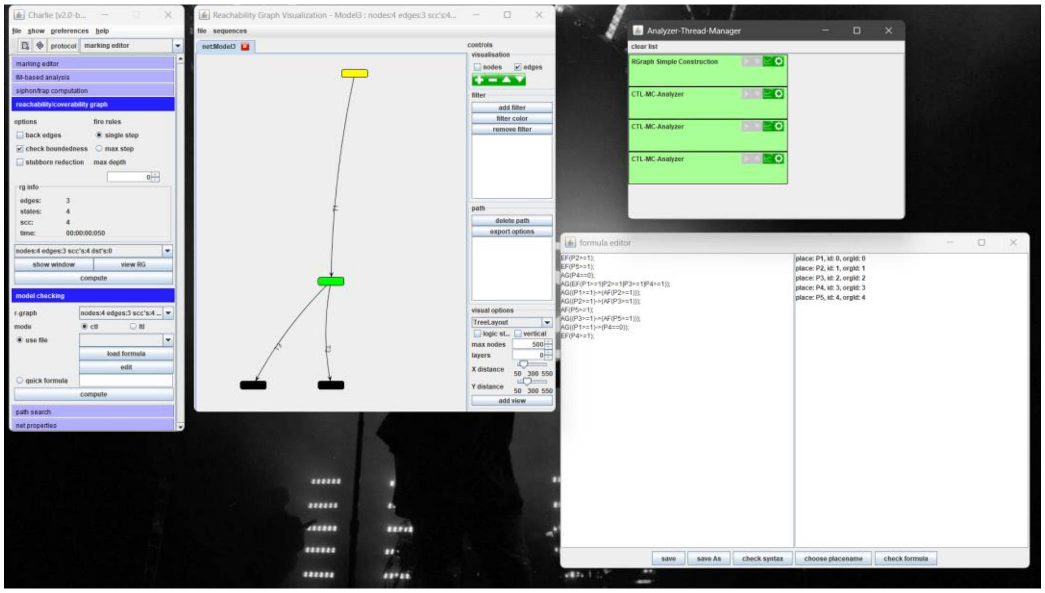


Test Analysis

CTL Property	Description	Result
EF(P2 >= 1)	**Item selection state is reachable.**	**Success**
EF(P5 >= 1)	**DOM update completed state is reachable.**	**Success**
AG(P4 == 0)	**Prevent transitioning into an error state persistently.**	**Success**

AG(EF(P1>= 1|P2>=1))	**It is always possible to reach a valid item selection or cart state.**	**Success**
AG((P1 >= 1) -> (AF(P2 >= 1)))	**If no items are rendered, it must eventually transition to the item selection state.**	**Success**
AG((P2 >= 1) -> (AF(P3 >= 1)))	**If an item is selected, it must eventually be rendered in the DOM.**	**Success**
AF(P5 >= 1)	**Always eventually reach the DOM update completion state.**	**Success**
AG((P3 >= 1) -> (AF(P5 >= 1)))	**If the cart is updated, the DOM update must eventually complete.**	**Success**
AG((P1 >= 1) -> (P4 == 0))	**Prevent transitioning to an error state directly from the initial state.**	**Success**
EF(P4 >= 1)	**Error state is reachable.**	**Success**

Screenshots



```
EF(P2>=1);
EF(P5>=1);
AG(P4==0);
AG(EF(P1>=1|P2>=1|P3>=1|P4>=1));
AG((P1>=1)->(AF(P2>=1)));
AG((P2>=1)->(AF(P3>=1)));
AF(P5>=1);
AG((P3>=1)->(AF(P5>=1)));
AG((P1>=1)->(P4==0));
EF(P4>=1);
```

```
place: P1, id: 0, orgId: 0
place: P2, id: 1, orgId: 1
place: P3, id: 2, orgId: 2
place: P4, id: 3, orgId: 3
place: P5, id: 4, orgId: 4
```

6. Strengths, Limitations, and Reflection

6.1 Strengths

The methodology employed for the NivX system, incorporating Petri Net modelling, secure design principles, and formal verification via Charlie, has shown significant efficacy in fulfilling system requirements and security standards. This approach's strengths reside in its capacity to meticulously model and evaluate both the functional and security dimensions of the system. Moreover, the application of Petri Nets and formal verification effectively represented the system's behaviors, concurrency, and crucial transitions. This enabled us to confirm that critical system attributes, including reachability, safety, and liveness, were validated. The application of Computation Tree Logic (CTL) features validated that the system is deadlock-free, prevents severe failures, and ensures the eventual completion of processes, hence enhancing overall reliability and resilience.

The methodology's focus on both static and dynamic analysis throughout testing significantly enhanced its efficiency. Thorough testing addressed edge situations and possible weaknesses, whereas the application of CTL verification in Charlie proved that both anticipated and unanticipated scenarios could be managed securely. This iterative methodology for modelling, verifying, and refining the system enhanced its consistency, concurrency management, and resilience to threats, ensuring the final system closely adheres to security best practices and functional specifications.

6.2 Limitations

Notwithstanding its efficacy, the methodology possessed many drawbacks. A key problem faced was the intrinsic complexity of Petri Net modelling in systems characterized by several concurrent operations and dynamic interactions. Developing comprehensive Petri Net models for the NivX system necessitated considerable time and effort, as it entailed precisely modelling both individual system components and their interdependencies and transitions. The intricacy occasionally hindered the management of bigger reachability graphs, which might become vast and more challenging to analyze, particularly when addressing retries, concurrent operations, and race situations.

A further restriction was the dependence on tool-specific limits. Although Charlie is an effective instrument for validating Petri Net models, it necessitates a degree of expertise to comprehend counterexamples, reachability graphs, and liveness features. The verification procedure presupposes that all potential system routes are represented in the Petri Net, which may not be true if minor design or modelling mistakes arise. Moreover, although the system fulfilled all CTL criteria, practical implementation may present unanticipated difficulties, such as managing extensive concurrency or unforeseen API errors, which may not be entirely represented in a static Petri Net.

6.3 Reflection

The application of formal methods and safe design principles in the development and verification of the NivX system was essential for ensuring functional correctness and security. Formal approaches offer a stringent, mathematical basis for modelling and validating system behaviors, aiding in the eradication of ambiguities and the early detection of design faults during the development process. Utilizing Petri Nets to model the system's concurrent processes and validating essential characteristics with Charlie enabled us to tackle significant security concerns, including the prevention of deadlocks, the preservation of safe state transitions, and the assurance of process completion under diverse scenarios. This significantly improved the system's overall robustness and reliability. This experience underscored the importance of integrating formal verification with secure design to establish a dependable and secure system. Nonetheless, it emphasized the necessity of reconciling complexity with practicality, as formal approaches may prove difficult to implement in big or highly dynamic systems. Advancing, a hybrid methodology that combines formal verification with real-time monitoring and automated security testing may significantly improve system security and reliability.

7. References

1. **Clarke, E.M., Emerson, E.A. and Sistla, A.P., 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), pp.244-263. Available at: https://doi.org/10.1145/5397.5399.**
2. **European Union, 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data. *Official Journal of the European Union*, L 119, pp.1-88. Available at: https://eur-lex.europa.eu/eli/reg/2016/679/oj [Accessed: 24 March 2025].**
3. **Internet Engineering Task Force, 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Available at: https://doi.org/10.17487/RFC8446 [Accessed: 24 March 2025].**
4. **Murata, T., 1989. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4), pp.541-580. Available at: https://doi.org/10.1109/5.24143.**
5. **National Institute of Standards and Technology, 2001. *Advanced Encryption Standard (AES)*. FIPS PUB 197. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf [Accessed: 21 March 2025].**
6. **National Institute of Standards and Technology, 2022. *Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities*. NIST SP 800-218. Available at: https://doi.org/10.6028/NIST.SP.800-218 [Accessed: 21 March 2025].**
7. **OWASP, 2021. *OWASP Testing Guide v4*. [Online] Available at: https://owasp.org/www-project-web-security-testing-guide/ [Accessed: 23 March 2025].**
8. **OWASP, 2023. *Zed Attack Proxy (ZAP)*. [Online] Available at: https://www.zaproxy.org/docs/ [Accessed: 22 March 2025].**
9. **Saltzer, J.H. and Schroeder, M.D., 1975. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), pp.1278-1308. Available at: https://doi.org/10.1109/PROC.1975.9939.**
10. **State of California, 2018. *California Consumer Privacy Act of 2018*. California Civil Code, Title 1.81.5. Available at: https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?lawCode=CIV&division=3.&title=1.81.5 [Accessed: 24 March 2025].**
11. **Snoopy: Heiner, M., Rohr, C. and Schwarick, M., n.d. *Snoopy – A tool to design and animate Petri Nets*. [Online] Available at: https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy [Accessed: 20 March 2025]. (Note: Authorship and exact URL may vary; verify from the official site.)**
12. **Heiner, M., Schwarick, M. and Wegener, J., 2015. Charlie – an extensible Petri net analysis tool. In: *Proc. PETRI NETS 2015, Brussels*. Springer, LNCS, volume 9115, pp. 200–211. Available at: https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie [Accessed: 21 March 2025].**
13. **Heiner, M., 2011. *Petri Nets with Snoopy and Charlie*. [Online] Available at: https://www-dssz.informatik.tu-cottbus.de/publications/tutorial_heidelberg_2011/Tutorial_English.pdf [Accessed: 20 March 2025].**
14. **MockAPI, n.d. *MockAPI*. [Online] Available at: https://mockapi.io/ [Accessed: 24 March 2025].**