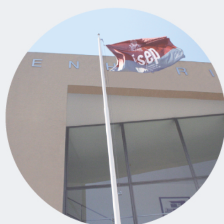
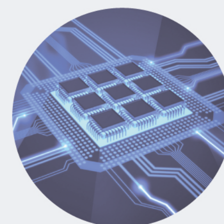
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS





Rumo a uma Estrutura de Verificação e Validação baseada em Arquiteturas de Microserviços

LIDIA GLORIA WILSON HERNÁNDEZ
outubro de 2025

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Towards a Verification and Validation Framework based on Microservices Architectures

Lidia Gloria Wilson Hernández

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: David Miguel Ramalho Pereira

Evaluation Committee
President:
Luís Miguel Pinho, Instituto Superior de Engenharia do Porto

Members:
David Pereira, Vortex Colab
Luis Nogueira, Instituto Superior de Engenharia do Porto

Porto, September 29, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

The work presented in this document is original and authored by me, and performed in the scope of the Master's degree in Critical Computing Systems Engineering.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration, all references have been acknowledged and fully cited, and all text was originally produced by me (except when duly noted).

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

Porto, Porto, September 29, 2025

Abstract

In recent years, microservices architectures have become a popular approach in software development to address the challenges of scalability, maintainability, and agility of complex systems. This thesis focuses on the intersection of microservices architectures with formal verification techniques, especially in the context of safety-critical applications such as Assisted and/or Automated Driving solutions.

The lack of research in this area requires a comprehensive investigation of how microservices architectures and formal verification techniques can interact effectively. The primary challenge is to seamlessly integrate microservices architectures with formal verification processes to ensure the safety properties typical of Cyber-Physical Systems, with a particular focus on autonomous driving applications. The importance of this work lies in the development of a robust framework that reconciles the decentralised nature of microservices with the requirements of formal verification.

Research objectives include analysing the benefits and limitations of microservices, identifying suitable formal verification tools and microservices frameworks, designing microservices APIs, and conducting experiments for validation. Successful implementations and a comprehensive overview of formal verification methods in safety-critical systems form the basis for this research.

As a proof of concept, this thesis presents the implementation of VVFramework, a microservices-based hybrid verification and validation framework that integrates NuSMV, Z3, and an automatic Python-SMV translation pipeline and demonstrates its applicability for small case studies in safety-critical domains.

Keywords: Microservices Architectures, Formal Verification, Verification Framework, Software Frameworks.

Resumo

Nos últimos anos, as arquiteturas de microsserviços tornaram-se uma abordagem popular no desenvolvimento de software para enfrentar os desafios de escalabilidade, manutenção e agilidade de sistemas complexos. Esta tese enfoca a interseção das arquiteturas de microsserviços com técnicas de verificação formal, especialmente no contexto de aplicações críticas para a segurança, como soluções de condução assistida e/ou automatizada.

A falta de investigação nesta área requer uma investigação abrangente sobre como as arquiteturas de microsserviços e as técnicas de verificação formal podem interagir de forma eficaz. O principal desafio é integrar perfeitamente as arquiteturas de microsserviços com os processos de verificação formal para garantir as propriedades de segurança típicas dos sistemas ciberfísicos, com um foco particular nas aplicações de condução autónoma. A importância deste trabalho reside no desenvolvimento de uma estrutura robusta que concilie a natureza descentralizada dos microsserviços com os requisitos da verificação formal.

Os objetivos da investigação incluem analisar os benefícios e limitações dos microsserviços, identificar ferramentas de verificação formal e estruturas de microsserviços adequadas, projetar APIs de microsserviços e realizar experiências para validação. Implementações bem-sucedidas e uma visão geral abrangente dos métodos de verificação formal em sistemas críticos para a segurança formam a base desta investigação.

Como prova de conceito, esta tese apresenta a implementação do VVFramework, uma estrutura híbrida de verificação e validação baseada em microsserviços que integra NuSMV, Z3 e um pipeline de tradução automática Python-SMV e demonstra a sua aplicabilidade para pequenos estudos de caso em domínios críticos para a segurança.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my supervisor, Prof David Pereira, for his unique ability to explain complex concepts clearly and for his patience throughout this journey. I would also like to thank Prof Eduardo Tovar, without whom my acceptance into the Master's programme would not have been possible. My deep appreciation goes to Prof. Luís Miguel Pinho for his valuable guidance and understanding and all the professors of the programme for their dedication and support.

I would like to thank my classmates for their academic support and encouragement and my colleagues at the CISTER Research Centre who contributed to this experience through their collaboration and guidance. I would also like to acknowledge the financial support from the Faculty of Engineering of the University of Porto (FEUP) and the CISTER Institute, which made this work possible.

On a personal note, I am deeply indebted to my parents, whose unwavering faith in me has been a source of strength throughout my life. I thank my husband for his unconditional help, trust and encouragement, which have given me the strength to carry on. Finally, I thank my dear friends, who have never doubted me, for their support and belief in this achievement.

Contents

List of Figures xiii

List of Tables xv

List of Source Code xvii

List of Abbreviations xix

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Background . . . 1
- 1.3 Problem Statement . . . 1
- 1.4 Objectives . . . 2
- 1.5 Thesis Organization . . . 3

2 State of the Art 5
- 2.1 Microservices Architectures . . . 5
  - 2.1.1 Architectural Principles and Patterns . . . 6
  - 2.1.2 Analysis of the strengths and weaknesses of microservices . . . 9
  - 2.1.3 Relevant technologies utilising microservices . . . 10
- 2.2 Formal Verification . . . 12
  - 2.2.1 Static Analysis: . . . 13
    - Model Checking . . . 13
    - Theorem Proving . . . 13
  - 2.2.2 Runtime Monitoring . . . 15
  - 2.2.3 Limitations of Formal Verification . . . 17
- 2.3 Concluding about Microservices and Formal Verification . . . 17

3 Structure of the VVFramework 19
- 3.1 Architectural Overview . . . 19
- 3.2 System Agents . . . 19
- 3.3 Architectural Scheme . . . 20
- 3.4 Organisation of the workflow . . . 22
- 3.5 Communication Model . . . 22
- 3.6 Summary . . . 23

4 Implementation of the VVFramework 25
- 4.1 Technology Stack . . . 25
- 4.2 User Interfaces Implementation . . . 26
  - 4.2.1 Web Interface Agent . . . 26
  - 4.2.2 The Verification Interface . . . 26

4.2.3 Underlying System Components . . . 29
4.3 Backend Service Implementation . . . 29
4.3.1 Python Service Agent (Orchestrator) . . . 29
4.3.2 Translation Agent (Python $\rightarrow$ SMV) . . . 29
4.3.3 NuSMV Agent . . . 30
4.3.4 Z3 Agent . . . 31
4.4 Communication Workflows and Data Exchange Model . . . 31
4.4.1 Verification workflows . . . 32
4.5 Microservices Architecture and Deployment . . . 32
4.5.1 Containers Deployed . . . 32
4.5.2 Docker Compose Configuration . . . 33
4.5.3 Dockerfile Configuration . . . 34
4.5.4 Python Dependencies . . . 37
4.5.5 Conclusions . . . 37
4.6 Shared Volume Architecture for Decoupling . . . 37
4.6.1 Auto-generated Files . . . 38
4.6.2 Manual Files: . . . 38
4.6.3 Benefits . . . 38

5 Validation of the VVFramework **39**
5.1 Evaluation Metrics . . . 39
5.2 Case studies for manual verification . . . 40
5.2.1 Case study: Traffic Light Control . . . 40
5.2.2 Case study: Login System . . . 43
5.2.3 Case study: Mutual Exclusion Protocol . . . 46
5.3 Validation of the automation pipeline . . . 49
5.3.1 Case study: A Finite State System . . . 49
5.4 Discussion of Results . . . 51
5.5 Interpretation of the Validation Results . . . 52

6 Conclusions and Future work **55**
6.1 Comparison with existing approaches . . . 56
6.2 Future work and strategic expansion . . . 57
6.3 Concluding remarks . . . 58

Bibliography **59**
A Traffic Light Control with CTL Specifications in Manual SMV . . . 63
B Python Program . . . 64
C Python Program . . . 65
D NuSMV Program . . . 67
E Python Program . . . 68

List of Figures

2.1 Microservices Architecture (Hannousse and Yahiouche 2021). 6
2.2 Docker client-server architecture (Al-Rakhami et al. 2018). 12
2.3 A sample NuSMV program (Choi and Heimdahl 2002). 14
2.4 Control flow graph representation of an ST code in NuSMV (Darvas et al. 2014). 14
2.5 The principle of theorem proving (Fakhfakh, Kallel, and Cheikhrouhou 2021). 16
2.6 Isabelle Proof Assistant (Jacobsen and Villadsen 2023). 16

3.1 Framework Architecture Diagram 21

4.1 VVFramework Principal View 26
4.2 VVFramework Manual Verifications Tools Selection View 27
4.3 VVFramework Manual Verifications Tab NuSMV 27
4.4 VVFramework Manual Verifications Tab Z3 28
4.5 VVFramework Automatic Verifications Tab View 28
4.6 Microservices-based Architecture 33
4.7 Running Containers Status 33

5.1 Traffic Light Control with CTL Specifications in Manual SMV Verification . 41
5.2 Traffic Light Control in Manual SMV Verification Result 42
5.3 Traffic Light Control in Manual Z3 Verification 42
5.4 Traffic Light Control in Manual Z3 SAT Result 44
5.5 Traffic Light Control in Manual Z3 UNSAT Result 45
5.6 Login System with CTL Specifications in Manual SMV Verification 47
5.7 Login System with CTL Specifications in Manual SMV Verification Result . 48
5.8 Mutual Exclusion Protocol in Manual SMV Verification 49
5.9 Mutual Exclusion Protocol in Manual SMV Verification Result 50
5.10 Mutual Exclusion Protocol in Manual SMV Verification Result 51

1 Traffic Light Control with CTL Specifications in Manual SMV Verification Result 63

List of Tables

4.1 Description of containers and their functions 32

6.1 Comparison of VVFramework with related approaches 57

List of Source Code

4.1 Input Python Code . . . 30
4.2 Generated SMV model . . . 30
4.3 API call . . . 31
4.4 Verification output . . . 31
4.5 SMT-LIB2 input . . . 31
4.6 JSON Result . . . 31
4.7 Docker Compose File for VVFramework . . . 33
4.8 Python Orchestrator Dockerfile . . . 35
4.9 NuSMV Agent Dockerfile . . . 35
4.10 Z3 Agent Dockerfile . . . 36
4.11 Requirements File . . . 37
5.1 Traffic Light in NuSMV model . . . 40
5.2 CTLSPEC properties evaluated in Traffic Light . . . 40
5.3 Login System in NuSMV model . . . 43
5.4 CTLSPEC properties evaluated in Login System . . . 45
5.5 Mutual Exclusion Protocol in NuSMV model . . . 46
5.6 CTLSPEC properties evaluated in Mutual Exclusion Protocol . . . 48
1 Python Program Example (example_program.py) . . . 64
2 Python Program Example (py2smv.py) . . . 65
3 Auto Model NuSMV . . . 67
4 Python Program Example (contracts_example.py) . . . 68

List of Abbreviations

DevOps	**Dev**elopment and **Op**eration**s**
DSP	**D**omain-**S**pecific **P**rotocol
ST	**S**tructured **T**ext
CFG	**C**ontrol **F**low **G**raph
HOL	**H**igher **O**rder **L**ogic
ACL2	**A C**omputational **L**ogic for Applicative Common Lisp
PVS	**P**rototype **V**erification **S**ystem

Chapter 1

Introduction

This chapter gives an overview of the thesis subject matter, background, problem statements, and objectives. It also details the development structure of this work.

1.1 Overview

Microservices architectures have become a popular approach in software development, providing a decentralized and modular structure that enables scalability and flexibility. However, integrating microservices architectures with formal verification tools and processes in a single framework for verification & validation of safety properties in safety-critical systems, such as autonomous driving applications, poses unique challenges. This thesis aims to address these challenges and develop a robust framework that aligns the decentralized nature of microservices with the challenging requirements of formal verification.

1.2 Background

Autonomous driving, a transformative technology poised to revolutionize transportation, demands absolute safety and reliability. Ensuring the safety of autonomous vehicles depends on meticulous verification and validation of their software systems.

Formal verification, a rigorous technique that employs mathematical methods to prove software correctness, holds great promise for achieving this goal. However, it is challenging to make use of formal verification tools in a non-integrated manner since typically a verification problem requires the usage of more than one single method to be used.

One way to design and build a flexible and scalable framework that incorporates multiple formal verification tools is to design it following the principles of microservices. As opposed to traditional monolithic architectures, microservices architecture promotes a distributed system composed of lightweight, independent services. While this modularity offers numerous advantages, it also introduces complexities when it comes to integrating it with formal verification. The decentralized nature of microservices poses challenges in managing interactions between services, ensuring the correctness of the system as a whole, and handling the immense volume of data generated by these interactions.

1.3 Problem Statement

Seamless integration of microservices architectures with formal verification processes is still a major challenge for ensuring safety properties in autonomous driving applications. The

decentralized nature of microservices, coupled with the complexity of their interactions and the huge amount of data they generate, presents obstacles to effectively applying formal verification techniques.

The crux of the problem lies in developing a framework that effectively manages these challenges and enables rigorous verification of microservices-based systems. Such a framework should:

- Effectively manage interactions between microservices, ensuring their coordination and compliance with security constraints.
- Perfectly integrates with formal verification tools, enabling analysis of overall system behavior and identification of potential security risks.
- Handle the large volume of data generated by microservices, providing efficient and scalable verification techniques.

The development of such a framework is crucial not only to advance the safety of autonomous driving systems but also to establish a deeper understanding of the compatibility between microservices architectures and formal verification in safety-critical domains.

1.4 Objectives

The primary objectives of this research align with the outlined goals in the proposal. These objectives are crafted to delve into the complexities of safety verification in microservices architectures. The specific aims of this research include:

- Conduct an in-depth analysis to elucidate the specific advantages and limitations of microservice architectures, with a focus on their feasibility in supporting formal verification services.
- Research and compile a list of software frameworks suitable for constructing microservice architectures, emphasizing those compatible with embedded systems.
- Proposing effective methodologies and tools tailored to address the challenges microservices pose in ensuring the formal verification services.
- Design and implement Application Programming Interfaces (APIs) for the activation and coordination of microservices within the framework.
- Evaluating the feasibility and efficacy of the proposed methodologies through practical implementation and experimentation.

1.5 Thesis Organization

The current work is composed of six chapters. Chapter 1 presents the thesis theme by describing the problem and enumerating the objectives this work intends to achieve. Chapter 2 provides an in-depth analysis of the fundamental characteristics of microservices architecture and formal verification. A list of software frameworks suitable for building microservices architectures is also investigated and shown. Chapter 3 presents the conceptual design of the proposed VVframework and describes its agents, workflows and overall architecture. Chapter 4 describes the implementation, including the technology stack, translation pipeline and deployment using Docker and Docker Compose. 5. validates the framework using case studies that illustrate manual and automated verification scenarios. Finally, Chapter 6 concludes the thesis by summarising the key contributions, highlighting limitations and outlining future research directions.

Chapter 2

State of the Art

In this chapter, we will discuss the current state of the art that is most relevant to our work and serves as the baseline for our thesis. We will begin by explaining the key concepts related to microservices, followed by the most relevant formal verification approaches and tools. These will be taken into consideration when developing the integrated framework that we envision for this thesis work.

2.1 Microservices Architectures

In recent years, Microservices Architecture has gained prominence as a modern approach to software design. Microservices are a style of architecture that emphasizes dividing the system into small and lightweight services that are purposely built to perform a very cohesive business function and are an evolution of the traditional service-oriented architecture style (Alshuqayran, N. Ali, and Evans 2016; Lewis and Fowler 2014). It involves decomposing applications into independently deployable services that communicate through lightweight protocols, enhancing scalability and robustness (Söylemez, Tekinerdogan, and Tarhan 2023; Wang et al. 2021).

Over the last decade, leading software consultancy firms and product design companies have found the microservices approach to be an appealing architecture that allows teams and software organizations to be more productive in general, and build frequently more successful software products. Many organizations outside of the traditional software business have also tried and tested this style of architecture and have found it to be very beneficial. Their efficient loose coupling enables their development using different programming languages, use different database technologies, and be tested in isolation concerning the rest of the underlying systems (Alshuqayran, N. Ali, and Evans 2016; Söylemez, Tekinerdogan, and Tarhan 2023; Wang et al. 2021).

Microservices can communicate with one another either directly via an HTTP resource API or indirectly through message brokers. These microservices can be deployed in either virtual machines or lightweight containers. It is preferred to use containers for deploying microservices due to their simplicity, lower cost, and faster initialization and execution (Hannousse and Yahiouche 2021). Please refer to Figure 2.1 for a visualization of this concept.

Microservices are also considered an appropriate architecture for systems deployed on cloud infrastructures, as they can take advantage of the elasticity and on-demand provisioning features of the cloud model. Companies such as Uber, Netflix, and SoundCloud have adopted the microservices style in the cloud and gained many benefits from it (Alshuqayran, N. Ali, and Evans 2016; Calçado 2014; Tonse 2014).

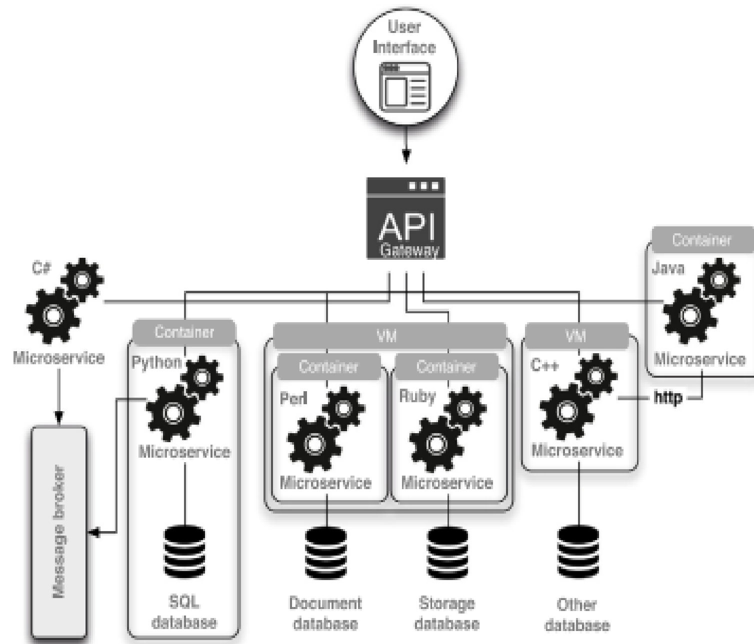


Figure 2.1: Microservices Architecture (Hannousse and Yahiouche 2021).

Notable tech giants like Microsoft, Google, and Amazon have adopted microservices to improve their systems. However, applying microservices in resource-constrained embedded systems, such as IoT devices, poses unique challenges (Wang et al. 2021).

The adoption of microservices architectures in automotive systems is a dynamic area of research. Existing studies showcase benefits such as enhanced flexibility, scalability, and ease of maintenance (Lotz et al. 2019).

Knowing a little about what microservices are all about, the next step is to go deep into the fundamental patterns and principles that govern their design and implementation. The next sections aim to reveal the complexities of microservices by delving into the principles and patterns that compose them.

2.1.1 Architectural Principles and Patterns

An architectural style is a collection of essential principles and patterns that work together to create recognizable designs and support consistent design practices. These principles guide the design decisions and express the purpose behind the design, helping to define the vision of the design. Patterns, on the other hand, are common, tested solutions to design issues frequently encountered in architectural practice. Therefore the principles and patterns are coordinated and harmonized ensuring the design follows a consistent direction, and design activities can be efficiently repeated (Taibi, Lenarduzzi, and Pahl 2018).

Microservices architecture is based on the idea of breaking down complex software systems into individual services that can be deployed and managed independently. Each microservice is designed to perform a specific function, which fosters modularity and autonomy. Communication between microservices is usually facilitated via well-defined APIs and messaging protocols, allowing for flexibility and scalability. These assertions have been discussed by authors such as (Jamshidi et al. 2018; Söylemez, Tekinerdogan, and Tarhan 2023).

Authors (Velepucha and Flores 2023) provide a detailed exploration of the key principles guiding the development of microservices architecture as identified by Martin Fowler and Sam Newman. Other authors such as (Taibi, Lenarduzzi, and Pahl 2018) also mention many of these principles, which are *Integration via Services*, *Organization*, *Decentralized Governance*, *Design for Failure*, *Multifunctional Teams*, *Decentralized Data Management*, *Evolutionary Design*, *Unlimited Application size*, *Independent Deployability*, *High Observability*, and *Ability to Code in Multiple Languages* presented below:

- *Integration via Services.* The application is divided into microservices, emphasizing their autonomy and self-contained nature.
- *Organization.* The microservices organization is structured based on business functions rather than technical layers, promoting alignment with business goals.
- *Decentralized Governance.* Decentralized governance is implemented through autonomous microservices, with specific teams responsible for maintenance.
- *Design for Failure.* Microservices are built to ensure that individual failures do not disrupt the entire application. Other microservices can continue operating without problems, making the system resilient to failures.
- *Multifunctional Teams.* Multifunctional teams are assembled with experts in software development, testing, deployment, monitoring, and issue resolution to handle microservices' complexities.
- *Decentralized Data Management.* Decentralized data management is implemented such that each microservice manages its data, ensuring independence and flexibility.
- *Evolutionary Design.* Microservices are designed with the flexibility to evolve and replace individual components without impacting the entire application.
- *Unlimited Application size.* By using microservices architecture, it is possible to develop a system that is scalable without any size limitations.
- *Independent Deployability.* The objective is to deploy microservices independently, enabling modifications or upgrades to specific microservices without affecting others or the entire system.
- *High Observability.* The importance of monitoring tools for microservices is emphasized as they facilitate proactive issue detection and resolution.
- *Ability to Code in Multiple Languages.* Each service can be implemented in a different programming language, allowing for the selection of the most appropriate language for the specific service.

When exploring microservice architecture patterns, researchers typically group them based on their objectives. Some patterns are particularly well-known and will be examined in this study. The focus will be on three categories that are commonly used to classify processes and technologies: *data storage*, *communication*, and *deployment*. The following is a summary of the most important aspects of each category.

- *Data storage patterns.* As described in (Taibi, Lenarduzzi, and Pahl 2018), are usually managed through two methods: database per service, and shared database.

  - In the case of database per service, each service has its own dedicated database, providing a high degree of isolation, autonomy, and security. However, this approach can lead to increased complexity in data management and consistency across services.
  - On the other hand, with a shared database, multiple services use the same database, simplifying data management and consistency. However, this approach can also lead to performance issues and potential data conflicts.

- *Communication pattern.* In classification some communication patterns have been identified such as publish/subscribe communication, synchronous communication, asynchronous communication, point-to-point communication, binary protocol patterns for communication, API Gateway, and event-driven architecture.
  - The publish/subscribe communication pattern is based on event-driven architecture. All events published on a topic are broadcast to subscribers. It provides instant event notifications for microservice-based applications (Karabey Aksakalli et al. 2021).
  - In synchronous communication mechanisms like HTTP-based REST or request/ response-based Apache, clients must wait for a service response before they can proceed. Once the client sends a request, it waits for the response. However, if the response doesn't arrive within a specified time limit, the client will receive a failure notification (Karabey Aksakalli et al. 2021).
  - In contrast to synchronous communication, asynchronous communication involves the client sending a request through a synchronous medium like a messaging queue. The client cannot block the communication while waiting for the response. Instead, the service response is delivered to the client at a later time. Asynchronous communication using message brokers is discussed in (Bakshi 2017). Message brokers are asynchronous communication mediums where a message-producing service sends messages to a queue, and consumers receive messages from the queue asynchronously. This type of communication separates message generators from message consumers. An intermediate message broker stores messages until consumers process them. In this way, the producer and consumer microservices are completely unaware of each other, and there is asynchronous communication between them (Karabey Aksakalli et al. 2021).
  - In a point-to-point communication architecture, the logic for message routing is contained within each endpoint. Each service instance can directly communicate with other services through APIs like REST. However, as the number of requests and the complexity of services increase, the performance of systems based on point-to-point communication can decline, which is a disadvantage of this architecture. Additionally, as the number of services grows, point-to-point communication patterns become increasingly complex in an exponential manner (Karabey Aksakalli et al. 2021).
  - Binary protocol patterns for communication can be more efficient than other protocols. This pattern allows having multiple microservices that expose more than one API. This also allows for efficient and reliable communication across different programming languages such as Java, Python, C++, and JavaScript.

Therefore, it may be the preferred choice when dealing with many services created using different programming languages (Karabey Aksakalli et al. 2021).

    - API Gateway is a crucial tool that ensures secure and efficient communication between clients and backend services. It serves as an entry point that manages API requests, routing them correctly, authenticating and authorizing them, and processing them efficiently (S. A. Ali and Zafar 2021). This is a very effective pattern as it allows for easy modification of services based on market needs without the need to modify the entire system (Taibi, Lenarduzzi, and Pahl 2018). However, relying on a single API Gateway instance can be problematic as it creates a single point of failure. To avoid this, it is recommended to replicate the API Gateway instance (Karabey Aksakalli et al. 2021).
    - Event-driven architecture is a design pattern that relies on events to trigger communication between microservices. This approach enables high levels of decoupling and distribution, as an event creator only needs to know that an event has occurred, without having to know who might be interested in the event or how it will be processed. This makes it a highly flexible and scalable architecture for modern applications (Theorin et al. 2017).

- *Deployment architecture.* In microservices deployment architecture, two patterns have been identified: the multiple service per host pattern and the single service per host pattern.
    - In the multiple services per host pattern, several services run on the same host (node). This deployment pattern has an advantage as it allows for easy scaling to deploy multiple service instances on the same host.
    - On the other hand, each service is deployed on its host in the single service per host pattern. This approach ensures complete isolation of services, thereby reducing the possibility of conflicting resources. However, it significantly reduces performance and scalability. Assigning a single server to each microservice is inefficient and contradicts the basic idea of microservices architecture (Taibi, Lenarduzzi, and Pahl 2018).

2.1.2 Analysis of the strengths and weaknesses of microservices

Microservices architecture provides several benefits. It allows the freedom to select the most appropriate technologies and programming languages (Jamshidi et al. 2018; Söylemez, Tekinerdogan, and Tarhan 2023).

Microservices are a software architecture approach that offers high maintainability and testability, according to (Richardson 2018). This approach provides great agility to a software system (Jamshidi et al. 2018) by enabling a modular approach, allowing teams to develop, deploy, and scale individual services independently. This modularity enhances agility and responsiveness to evolving business needs. In addition, the microservices architecture also provides the ability to scale and maintain availability, as mentioned in (Söylemez, Tekinerdogan, and Tarhan 2023).

The decentralized nature of microservices ensures that if a fault occurs in one service, it won't affect the others, see Figure. 2.1 above. This contributes to the reliability and resilience of the system. Microservices architecture includes loosely coupled services, which

makes the system more fault-tolerant. When a failure occurs, only specific and relevant services are affected, preventing the entire system from becoming unavailable.

Microservices support continuous integration and deployment, enabling frequent and efficient releases. This results in faster time-to-market and the ability to adapt quickly to changing requirements. The author (Shadija, Rezai, and Hill 2017; Söylemez, Tekinerdogan, and Tarhan 2023) claims that it is easy to adapt to new changes when built with a good microservices architecture.

When built with a good microservices architecture, microservices are independent and do not impact each other. Therefore, if one microservice fails, it does not cause the entire application to crash (Prasandy et al. 2020; Velepucha and Flores 2023). Adopting a microservices architecture can be challenging, mainly due to the steep learning curve at the beginning. One of the main challenges is organizational restructuring, which involves creating autonomous teams responsible for developing new microservices. These teams may use different programming languages and new tools to automate compilation, deployment, and monitoring. Additionally, training may be necessary to fully understand the complexities of deploying containerized microservices to cloud computing environments.

One disadvantage of incorporating a new architecture like microservices is the lack of experience from the organization or development team. They may not know if they will be successful due to factors such as limited experience in working with this new architecture (Fan et al. 2018; Velepucha and Flores 2023).

In a microservices architecture, it is common for each microservice to have its own dedicated database. However, this approach requires duplicating data across multiple new databases. Consequently, additional programming mechanisms are necessary to ensure data consistency, which increases the overall complexity. As the number of microservices increases, some services become more difficult to orchestrate, making the challenge even more intense. (Velepucha and Flores 2023).

Microservices typically result in a higher frequency of calls to executed functions, leading to communication delays and increased network overhead (Velepucha and Flores 2023). Therefore, a thorough examination of the network architecture may be necessary, which could require architectural adjustments. It is also crucial to consider programming-level optimizations such as enabling asynchronous calls and implementing parallel processing to alleviate the impact of network overhead.

Although modularity can be beneficial, managing a large number of microservices can introduce complexity in terms of service discovery, communication, and data consistency.

2.1.3 Relevant technologies utilising microservices

To build microservices architectures, various software frameworks, technologies, and cloud services have gained prominence, particularly in the context of embedded systems. The literature shows a landscape characterized by the following:

- Flask is a Python web framework that is lightweight, flexible, and extensible. It emphasizes simplicity and can be installed and configured quickly without requiring many external dependencies. Flask's components are modular and easy to extend using third-party libraries and extensions, making it an excellent choice for developers who need complete control over their application's architecture. Flask is highly customizable, making it suitable for tailored solutions that meet the specific requirements of

a project. It is a micro-framework that provides only the essential components for building web applications, leaving most of the configuration and implementation to the developer. Flask is ideal for smaller projects that require a quick start and easy customization. Its lightweight nature makes it well-suited for building RESTful APIs that handle data exchange between applications. Flask can also be used to create lightweight and modular microservices that can be easily deployed and scaled (Ghimire 2020).

- Django is a full-stack Python web framework that comes with a wide range of built-in features. These features include database abstraction, form handling, template rendering, and authentication. This makes it an ideal choice for fast-paced development and reduces the need for developers to create custom code, but it can also make it more difficult to customize. Django's architecture is designed for performance and scalability. It can handle high volumes of traffic, supports multiple databases, and can be easily extended with third-party libraries. Additionally, Django has extensive documentation and a large community of users who provide abundant resources and support to developers. Django's comprehensive features and robust architecture make it well-suited for developing large-scale web applications with complex requirements (Ghimire 2020).

- Spring Boot is a Java framework that uses the Controller View Model design pattern. It offers a simple and easy way to develop and deploy REST services using various components and libraries. This framework is widely popular due to its simplicity and convention-over-configuration approach. It allows developers to build standalone, production-grade Spring-based applications with ease. Additionally, it can provide stable data services, making it simple to maintain and extend functionality (F. Zhang et al. 2021).

- Docker is a container technology that enables the automatic deployment of applications with their dependencies in a self-contained environment, known as containers (Boettiger 2015). It's a lightweight application that facilitates rapid container deployment. The containers are isolated from each other and are given individual network interfaces (Al-Rakhami et al. 2018). Docker containers are more efficient than virtual machines in running different applications and are ideal for building dynamic cloud applications that can scale with load or add and remove features on demand (Al-Rakhami et al. 2018). In summary, Docker provides containerization for microservices, ensuring consistency across different environments (see Figure 2.2).

- Kubernetes is an open-source tool for orchestration that simplifies the deployment, management, and execution of containerized applications. It allows for the replication of containers within a pod, which improves resource usage, load distribution, and fault tolerance (Rossi, Cardellini, and Presti 2020). The use of software containers and orchestration tools such as Kubernetes has made the development and management of microservices much easier (Rossi, Cardellini, and Presti 2020). In summary, Kubernetes is an orchestration platform that streamlines the deployment and scaling of containerized applications.

- Services such as AWS Lambda (Villamizar, Garcés, Ochoa, Castro, Salamanca, Verano Merino, et al. 2017) enable the implementation of microservice architectures without the need to manage servers. This makes it easier to create functions (i.e. microservices) that can be deployed easily and scaled automatically, while also helping to reduce

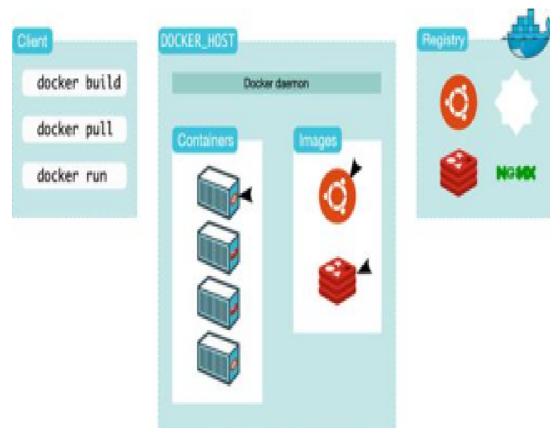


Figure 2.2: Docker client-server architecture (Al-Rakhami et al. 2018).

infrastructure and operational costs (Villamizar, Garcés, Ochoa, Castro, Salamanca, Verano, et al. 2016). AWS Lambda is ideal for microservices with sporadic workloads, providing serverless computing that allows developers to focus on code without having to manage the underlying infrastructure.

- Azure Service Fabric is a microservice architecture implementation by Microsoft. It is a distributed systems platform that simplifies the process of packaging, deploying, and managing scalable and reliable microservices and containers. The platform also tackles the significant challenges involved in developing and managing cloud-native applications. Service Fabric is designed to build and manage enterprise-class, tier-one, cloud-scale applications running in containers (Sahay and Sahay 2020). Azure Service Fabric offers the necessary infrastructure and tools to develop scalable and resilient services.

2.2 Formal Verification

Formal verification is a rigorous approach based on mathematical foundations that are used to prove or disprove the correctness of a system/component concerning a given specification or formal property. Formal verification is a powerful approach for ensuring the reliability and safety of critical systems, such as those in the aerospace, automotive, medical, and financial industries. In safety-critical systems such as autonomous driving, where human lives are at stake, formal verification becomes imperative for identifying and eliminating potential hazards.

Formal verification is an analytical approach within the broader domain of formal methods, which encompasses a collection of precisely defined mathematical languages and methodologies for articulating, modeling, and validating systems. Specifications are elaborated to coordinate desired system properties using rigorous and unambiguous notations. Systems are represented using mathematically based languages that support established theoretical frameworks such as finite-state automata, directed graphs, Büchi automata, and Petri nets, among other mathematical theories, models, and languages. The verification process involves a mathematical proof to determine whether the model conforms to the specified requirements. Formal verification has demonstrated success in various applications, particularly in the domain of computer hardware, where ensuring performance, and correctness is

crucial. Two specific technologies, model checking and automated theorem proving, have proven useful in the formal verification of expansive systems (Bolton, Bass, and Siminiceanu 2013; Grimm, Lettnin, and Hübner 2018; Wing 1990).

2.2.1 Static Analysis:

One of the methods developed for formal verification techniques in safety-critical systems is static analysis. This examines the source code without executing it, identifying potential issues through code analysis Sözer 2015. It can be argued that this type of verification can decrease the overall cost of the final product and its maintenance since fewer resources are spent on fixing errors. It can also be argued that it tends to be complex and can increase the initial cost of a project. One of the main advantages of static analysis is that it Can detect errors early in the development process and ensure the final product is correct and reliable (Krichen 2023; Todorov, Boulanger, and Taha 2018).

Model Checking

Analyses finite-state models of systems to exhaustively verify properties and detect potential issues. Model checking is a highly automated approach to verify that a formal system model satisfies a set of desired properties. A formal model describes a system as a set of variables and transitions between variable states (Wing 1990). One of the main advantages of model checking is that it can provide complete coverage of all possible behaviors, making it a powerful tool to ensure the correctness of critical systems (Krichen 2023). Some of the most famous model-checkers in the context of real-time systems are UPPAAL (Larsen, Pettersson, and Yi 1997), KRONOS (Bozga et al. 1998), and NuSMV (Cimatti et al. 2000).

- NuSMV is a symbolic model checker that emerged from the revision, reimplementation, and extension of SMV, the original BDD-based model checker developed at Carnegie Mellon University [15]. The goal of the NuSMV project is to provide a state-of-the-art symbolic model checking tool suitable for technology transfer initiatives. It provides a well-structured, open, flexible, and carefully documented platform that is both robust and closely aligned with industrial system standards. Figure 2.3 illustrates a small NuSMV model. The state variables and input variables, which determine the state space, are declared under the VAR keyword. Since NuSMV is limited to describing finite state machines, only finite data types are provided, such as Boolean, enumerations, integer subrange, and fixed-length array. NuSMV uses several keywords to structure its language, including MODULE, VAR, IVAR, DEFINE, ASSIGN, TRANS, INVAR, and SPEC. These keywords serve various purposes: MODULE represents a reusable component, VAR and IVAR declare variables, DEFINE provides a symbolic representation of expressions, ASSIGN and TRANS specify transition relations, INVAR specifies system invariants, and SPEC specifies system properties in temporal logic. Figure 2.4 illustrates an extract from a simple ST code, the corresponding CFG, and its NuSMV representation.

Theorem Proving

This is a specific type of deductive method. Involves using mathematical logic to prove the correctness properties of a system. Theorem proving is a deductive technique that is similar to traditional pencil-and-paper proofs. From a set of axioms, using a set of inference rules,

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
  counter : 0..10;
  m_check_busy : busy_status(request, state);

ASSIGN
  init(state):=ready;
  next(state):=
     case
        (state=ready & request=1)  | (m_check_busy.status=1)  : busy;
         1                                                     :ready;
     esac;
  ...

SPEC
  AG(request -> AF state=busy)

MODULE busy_status(request, state)
VAR
  status : boolean;

ASSGIN
  init(status):= 0;
  next(status):= (state=busy & next(request)=1);
```

Figure 2.3: A sample NuSMV program (Choi and Heimdahl 2002).

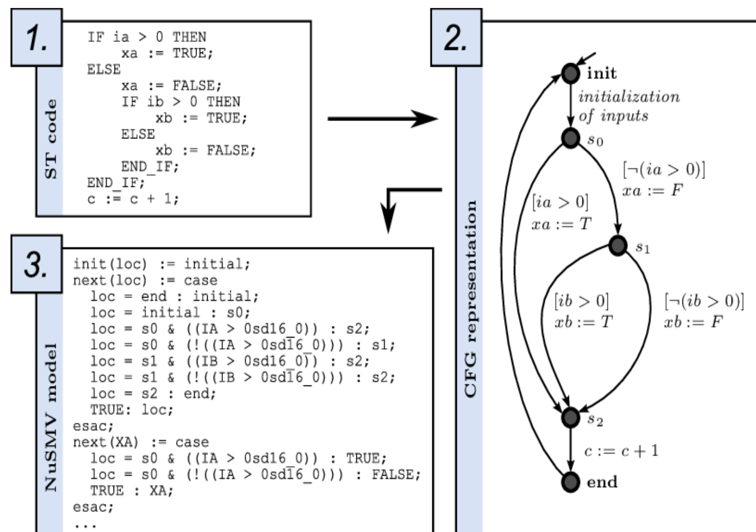


Figure 2.4: Control flow graph representation of an ST code in NuSMV (Darvas et al. 2014).

one builds theories and proves theorems to verify correctness claims about the system under investigation with the help of a proof assistant program (Kern and Greenstreet 1999; Wing 1990). In Figure. 2.5, the basic idea behind the theorem-proving method can be seen. This method involves describing how a system behaves using mathematical equations and generating a set of proof obligations with the help of a proof assistant. These proof obligations can be checked automatically or with some human involvement. If a particular proof obligation cannot be confirmed, it is important to make changes to the formal specification. The theorem-proving method is especially useful for ensuring the correctness of safety-critical systems, such as those used in aviation and medical devices. One of its main advantages is that it can provide a high level of assurance that the software is correct, and can ensure the correctness of safety-critical systems (Krichen 2023). Some of the most well-known theorem provers are Z3(Moura and Bjørner 2008), Coq, Isabelle, HOL, ACL2, and PVS (Fakhfakh, Kallel, and Cheikhrouhou 2021; Kern and Greenstreet 1999). Figure 2.6 shows an image obtained using a method to assess students' understanding of Isabelle/HOL Isar language structured tests. The question given is in declarative style and includes a lemma and a comment with a "proof" of the lemma.

- Z3, developed by Microsoft Research, it has established itself as one of the most influential SMT (Satisfiability Modulo Theories) solvers in the field of formal verification. Since its presentation in international forums, it has been adopted in both research and industry for its efficiency in automatically reasoning about complex logical properties. Z3 stands out because it supports a wide range of theories, including integer and real arithmetic, bit-vectors, arrays, and quantifiers, making it a versatile tool for modelling software and hardware systems. In Romanowicz 2010 explores applications of the solver in automatic program verification. It argues that traditional methods based on manual testing or interactive theorem systems are costly and prone to human error. By automating the process, Z3 enables the derivation of loop invariants, the verification of sorting algorithms and the analysis of concurrent programs in the style of Owicki-Gries. Although the study acknowledges limitations- for example in the verification of complex recursive programs- it shows that Z3 significantly reduces manual intervention and thus comes closer to the vision of a more practical and accessible verification. The use of Z3 is not limited to software. Q. Zhang et al. 2020 demonstrates its potential in the hardware domain. In it, the authors address the problem of detecting hardware Trojans in gate-level designs, a critical threat in the integrated circuit supply chain. They propose an automated framework that combines Gate-Level Information Flow Tracking (GLIFT) with Z3 to verify confidentiality policies. The solver is used as an engine to propagate security labels, enabling the detection of sensitive information leaks without requiring extensive manual efforts that typically slow down theorem proving- based techniques. This work demonstrates how Z3 helps bridge the gap between scalability and accuracy in hardware verification.

2.2.2 Runtime Monitoring

This is a dynamic verification technique that involves observing the behavior of a system during execution to detect deviations from expected norms (Cassar et al. 2017; Chen et al. 2015; Kern and Greenstreet 1999; Sözer 2015). It is a lightweight and scalable approach that can be used to monitor a wide range of systems, including software, hardware, and cyber-physical systems. Some of the frameworks used to generate monitors are Copilot (Pike, Niller, and Wegmann 2012) and MARS (G. Nandi et al. 2022; G. S. Nandi et al. 2020).

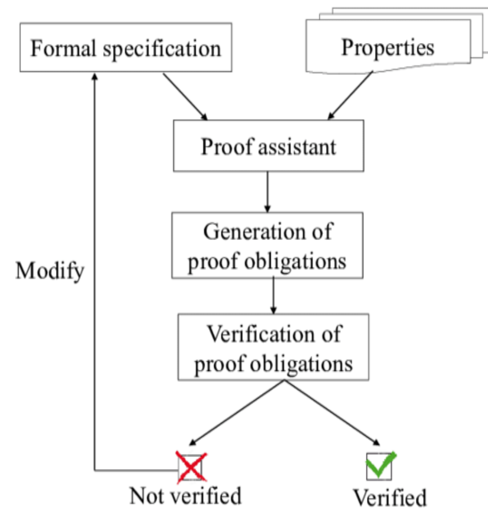


Figure 2.5: The principle of theorem proving (Fakhfakh, Kallel, and Cheikhrouhou 2021).

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹¬ (∃x. ¬ p x)›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p a›
    proof (rule ccontr)
      assume ‹¬ p a›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› show ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊥ ..
qed
```

Figure 2.6: Isabelle Proof Assistant (Jacobsen and Villadsen 2023).

2.2.3 Limitations of Formal Verification

Although formal verification techniques provide trusted and valid mathematical proofs, they have certain limitations (Bolton, Bass, and Siminiceanu 2013).

As systems become more complex, one of the major challenges faced by model checking is the state explosion problem, which leads to difficulties in managing memory and time resources. (Bolton, Bass, and Siminiceanu 2013).

The process of theorem proving is not completely automated, although it is theoretically free from some limitations faced by model checking. Analyst involvement is necessary, and the degree of automation decreases as the complexity of the logic used increases. Highly skilled experts are needed to oversee the theorem-proving process, which can be labor-intensive besides the fact that most experts in theorem proving remain in the academy so they are a scarce resource for integrating industrial teams (Bolton, Bass, and Siminiceanu 2013).

One of the main challenges in formal verification methods is ensuring that the models used are valid. To gain accurate insights into real systems, the models used in formal verification must be valid. If the models are not valid, the verification process may lead to false problems or overlook actual issues in the real system (Bolton, Bass, and Siminiceanu 2013).

2.3 Concluding about Microservices and Formal Verification

The use of microservices is highly advantageous for supporting formal verification services due to several important architectural features.

Microservices enable continuous integration and deployment, allowing for rapid, isolated updates to formal verification tools (such as NuSMV or Z3) without disrupting the entire system. Essential to maintaining system integrity is error isolation; an error in one verification agent does not affect other services.

However, in order to utilise these advantages, communication must be carefully planned. Secure, well-defined protocols and data exchange methods are required for effective communication between services. Ultimately, while microservices offer many advantages, their design and the choice of underlying tools must be carefully matched to the mathematical and computational requirements of the formal verification process.

Chapter 3

Structure of the VVFramework

This chapter presents the design of the proposed VV framework – a hybrid web-based system that combines manual and automated formal verification processes. The framework follows a microservices-oriented architecture, where each verification tool and core service is encapsulated in an independent container, enabling modularity, scalability, and interoperability.

The design focuses on the integration of classical model checking tools such as NuSMV, theorem proving, and SAT solving via Z3. This approach supports both manual verification, where the user explicitly specifies formal properties, and automatic verification, where the framework independently derives models and invokes verification tools.

3.1 Architectural Overview

The architecture of the VVFramework is based on a microservices model in which each functional unit is encapsulated as a service that runs in an isolated Docker container. The services communicate via RESTful APIs and exchange intermediate artefacts, such as translated models, via connected volumes.

At the top level, the architecture is divided into the following four layers:

1. User interaction layer: The web interface that allows users to upload Python code, define temporal logic properties (CTL/LTL), and view validation results.
2. Orchestration layer: The Python Service Agent, which is responsible for coordinating requests, invoking translation services, distributing tasks to verification agents, and summarising results.
3. Verification layer: A set of independent agents that encapsulate verification systems such as NuSMV and Z3.
4. Shared file system: A dedicated file system that can be accessed by all agents and is used to transfer intermediate files between services.

3.2 System Agents

The framework is structured around four main agents, each encapsulated as a microservice:

- Web Interface Agent
  - Built with Flask as backend and Vue.js as frontend.

  - Provides an interactive environment for users to write Python programmes, define temporal properties, and check verification results.
  - Communicates with the Orchestration Layer via Axios and REST APIs.
- Python Service Agent (Orchestrator)
  - Acts as the central coordinator of the framework.
  - Contains the integrated Translation module (py2smv.py) for parsing Python code and generating SMV models.
  - Responsible for receiving user input, forwarding it to the appropriate verification agent, collecting results, and normalizing them for the web interface.
  - Implements the automated pipeline for: Python → SMV → Verification → Results.
- NuSMV Agent
  - Encapsulates NuSMV 2.6.0 as a model checking service.
  - Supports CTL and LTL specifications.
  - Provides formal verification results, including proofs or counterexamples when properties are violated.
- Z3 Agent
  - Encapsulates Z3 4.15.3.0 as a SAT/SMT solver service.
  - Handles proof obligations, invariance checking, and constraint solving.
  - Complements NuSMV by providing counterexamples or satisfiability results in cases where model checking alone is not sufficient.

3.3 Architectural Scheme

Figure 3.1 illustrates the conceptual architecture of the VVFramework. The system is designed to support three different verification workflows, all of which converge in a unified web interface that displays the results in a standardised format:

- **Manual SMV Verification**: Users enter SMV models and CTL/LTL specifications directly, which are processed by the **NuSMV model checking** programme.
- **Manual Z3 Verification**: Users enter Z3 conditions and properties, which are processed by the **Z3 SMT solver** for satisfiability checking.
- Automatic Python → SMV Pipeline: Python programmes are automatically translated into SMV models using the py2smv.py module. The resulting models are then verified with both **NuSMV** and **Z3** (via contracts_example.py) to ensure consistency between the model checking and theorem proving approaches.

This unified approach allows the user to compare and analyse the verification results of different formal methods.

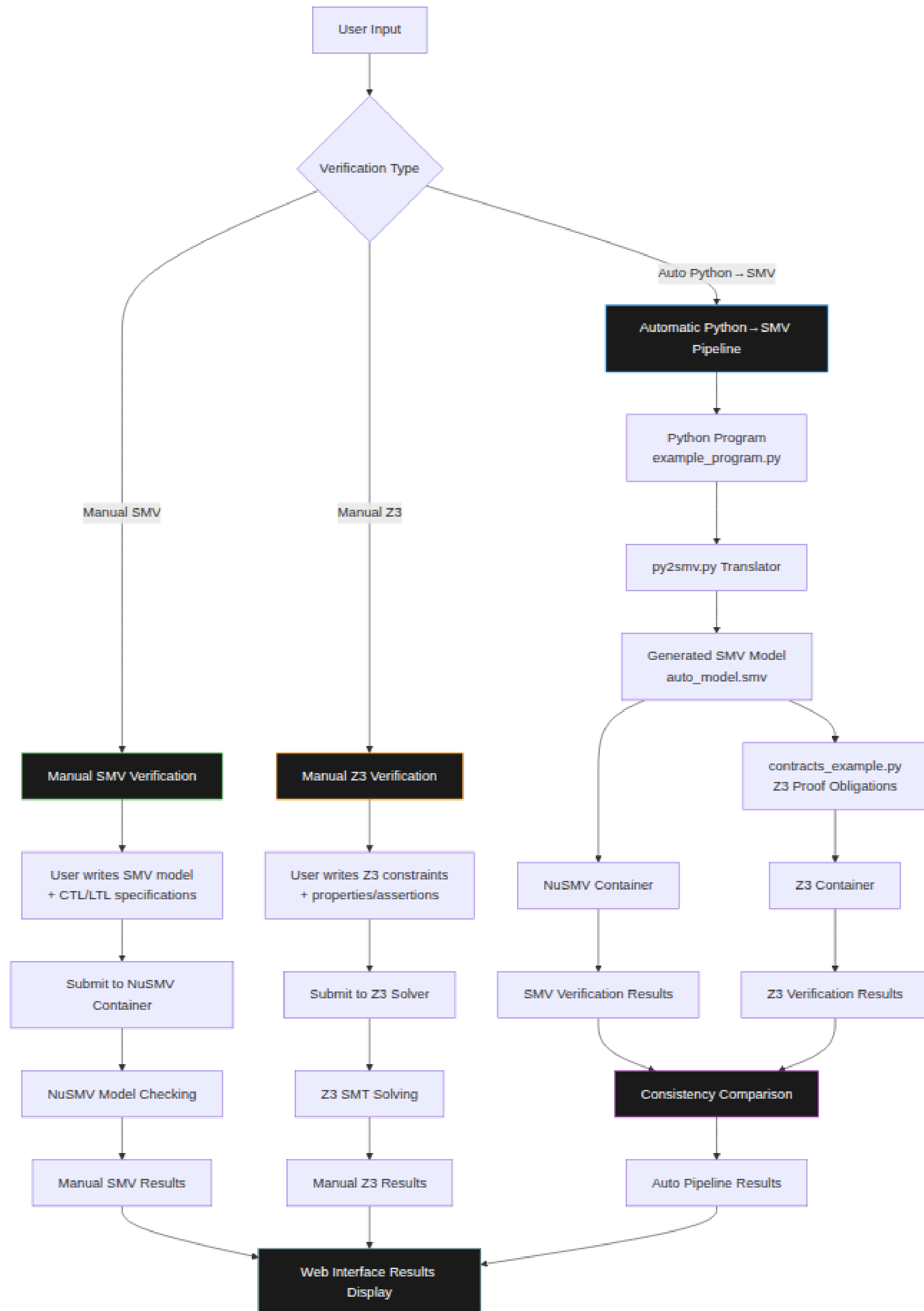


Figure 3.1: Framework Architecture Diagram

3.4 Organisation of the workflow

The VVFramework is designed to support both manual and automatic verification through different user experiences and backend workflows.

Manual Verification Workflow: This workflow is initiated when the user selects the manual verification tab on the web interface. In this mode, the user has direct control over the formal tools. The user explicitly writes or uploads an SMV model or Z3 code and defines the properties they want to check. The Python Service Agent receives this direct input and forwards it to the NuSMV or Z3 agents without prior translation. This enables a fine-grained specification of security and liveness properties.

Automatic verification process: This process is triggered by a single user action on the automated pipeline tab. The user simply deploys a Python programme without explicitly defining any formal properties. The Python Service Agent acts as an orchestrator and automatically invokes its internal translation module to generate an SMV model. It then applies a standard set of security properties (e.g. invariants, deadlock-free) and distributes the verification tasks to both the NuSMV and Z3 agents. The results are summarised and any security violations or counterexamples are returned to the user.

3.5 Communication Model

Communication between the services is based on a REST-based request/response model. The main Flask application acts as an API gateway and forwards requests from the web interface to the corresponding internal services.

The API endpoints are structured in such a way that a clear distinction can be made between manual and automated workflows.

- */api/v1/auto/validate-approach*: Manages the automated pipeline, from translation to parallel verification and aggregation of results.
- */api/v1/auto/check-smv*: Is called internally to initiate NuSMV checks in the automated workflow.
- */api/v1/manual/nusmv*: Receives and processes direct SMV/CTL/LTL input from the user.
- /api/v1/manual/z3: Receives and processes the user's direct Z3 code.
- */z3-proof*: Is called internally to initiate Z3 checks for the automated process.
- */py2smv*: Internal endpoint for the translation module of the Python Service Agent.

Intermediate artefacts, such as the .smv files generated by the translation module, are stored in a shared Docker volume. This ensures that the results of the translation are accessible to the verification agents, decoupling the generation process from consumption and allowing the services to work independently.

3.6 Summary

The design of VVFramework integrates microservices principles with formal verification methods, enabling a flexible, scalable, and extensible environment for hybrid verification. Its modular architecture enables for seamless interaction between user input, translation mechanisms, and multiple verification engines. By supporting manual and automatic workflows, VVFramework reduces the barrier to entry into formal verification while maintaining mathematical rigour.

Chapter 4

Implementation of the VVFramework

This chapter describes the practical implementation of the proposed VVFramework. This describes in detail the technologies used, development decisions, translation mechanisms, and implementation strategies. The implementation follows a microservices-based approach, where each component is encapsulated in a Docker container and communicates with others via REST APIs.

The chapter is organised as follows. Section 4.1 introduces the technology stack. Section 4.2 describes the implementation of user interfaces. Section 4.3 describes the implementation of backend services where explains the translation process from Python to SMV, and others. Section 4.4 presents the topic of communication and data exchange. Section 4.5 the microservices architecture and deployment. Section 4.6 is about shared volume structure. Finally, section **??** summarises the most important findings.

4.1 Technology Stack

The implementation of the VVFramework uses a specific technology stack, chosen for its balance of ease of development, community support, and compatibility with formal verification tools.

The back-end is mainly built using Python 3.12, which is the core language for the orchestration and translation services. The front-end interface uses JavaScript.

The VVFramework utilises several important libraries and frameworks:

- Flask and FastAPI manage the RESTful back-end services.
- Vue.js 2, paired with Axios, provides a reactive front-end interface and processes HTTP client requests.

The VVFramework integrates two powerful tools for formal verification:

- NuSMV 2.6.0 is used for model checking.
- Z3 4.15.3.0 is used to solve SAT/SMT.

To ensure modularity and easy deployment, the services are encapsulated with Docker and orchestrated with Docker Compose, which simplifies the management of the various containers that make up the system.

4.2 User Interfaces Implementation

The architecture of the VVFramework is based on a series of microservices, each of which has a specific function. This modularity simplifies development and management and allows each component to work independently.

4.2.1 Web Interface Agent

The Web Interface Agent, implemented as a Vue.js application with a Flask backend, is the primary point of user interaction. Its main function is to provide a unified interface for all verification tasks, i.e., the home page (index.html), as shown in Figure 4.1, serves as the landing page and entry point for the system. It contains a welcome message and a description of the framework, as well as navigation buttons that lead users to the most important functions, such as the verification and dashboard pages. It includes a code editor for uploading Python programmes and an editor for setting CTL/LTL properties. It also has a results window that displays the verification results, including counterexamples and proof obligations. The frontend communicates with the backend via REST calls using Axios.

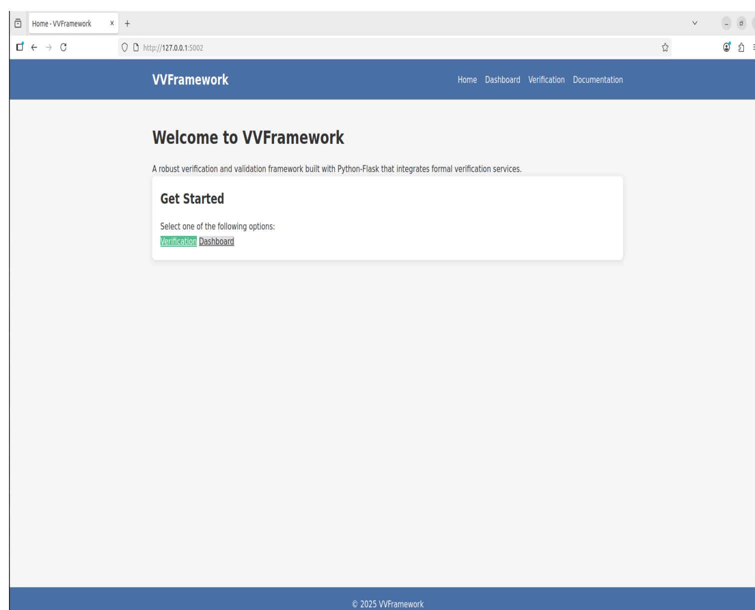


Figure 4.1: VVFramework Principal View

The dashboard interface (dashboard.html) acts as the central control panel. It has a sidebar that allows you to navigate between the different tools, as well as cards that display key system metrics such as active checks and completed validations. A recent activity section provides a log of the framework's recent tasks and provides a general overview of the system. This interface is currently under development, as it was not relevant to the research.

4.2.2 The Verification Interface

The core of the VVFramework is the Verification Interface (verification.html), in which the entire formal analysis takes place. This interface is structured with a tabbed architecture to enable both manual and automated workflows.

The Manual SMV System tab allows the user to work directly with formal tools, which is shown in Figure 4.2. It contains separate sections for NuSMV and Z3. For NuSMV, a text area is available for the SMV model, along with dynamic fields for specifying CTL/LTL properties, 4.3. The Z3 section, see Figure 4.4, provides similar fields for entering code and assertions. This tab is intended for users who want direct control over their models and properties.

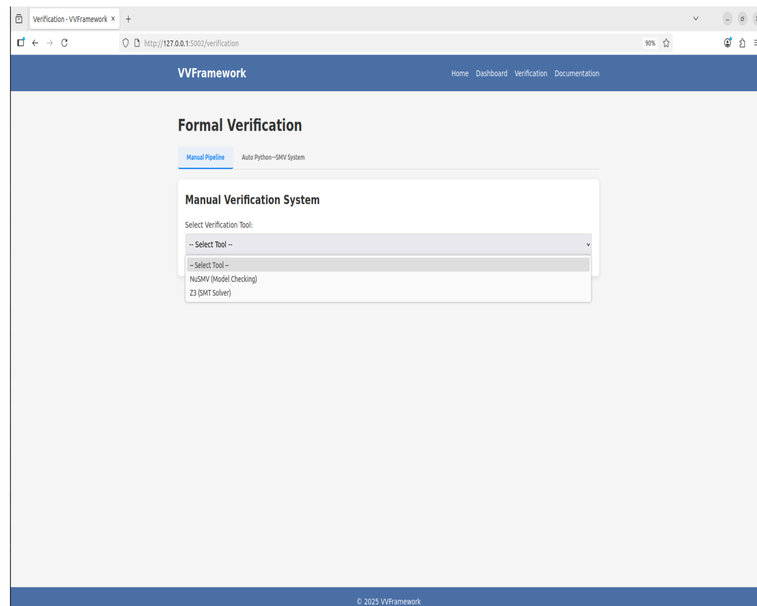


Figure 4.2: VVFramework Manual Verifications Tools Selection View

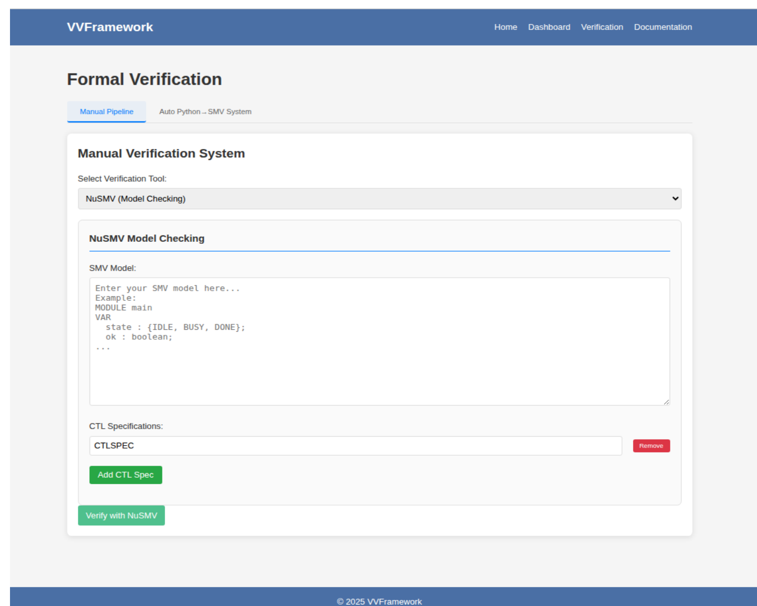


Figure 4.3: VVFramework Manual Verifications Tab NuSMV

The Auto Python → SMV System tab activates the automatic pipeline, see in Figure 4.5. This area displays information about a preloaded Python programme and contains a single button to run the full automatic verification pipeline. The output on this tab includes a consistency check between the results of the SMV and Z3 engines.

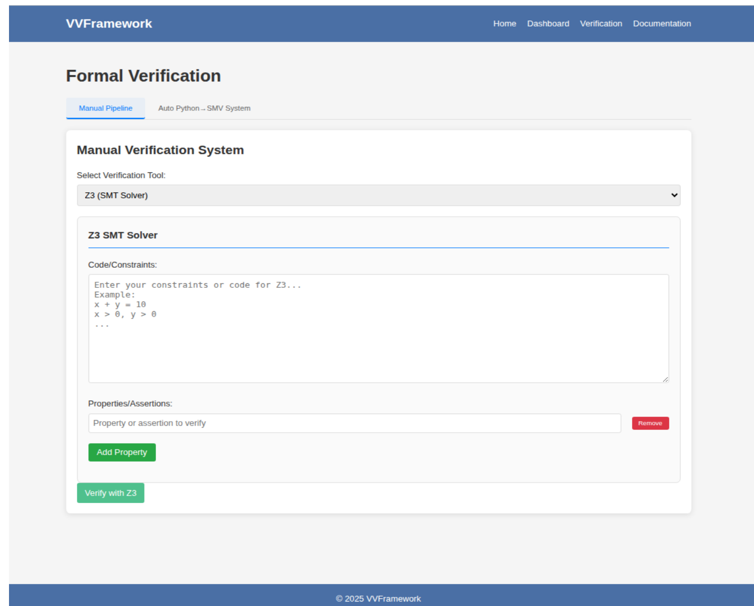


Figure 4.4: VVFramework Manual Verifications Tab Z3

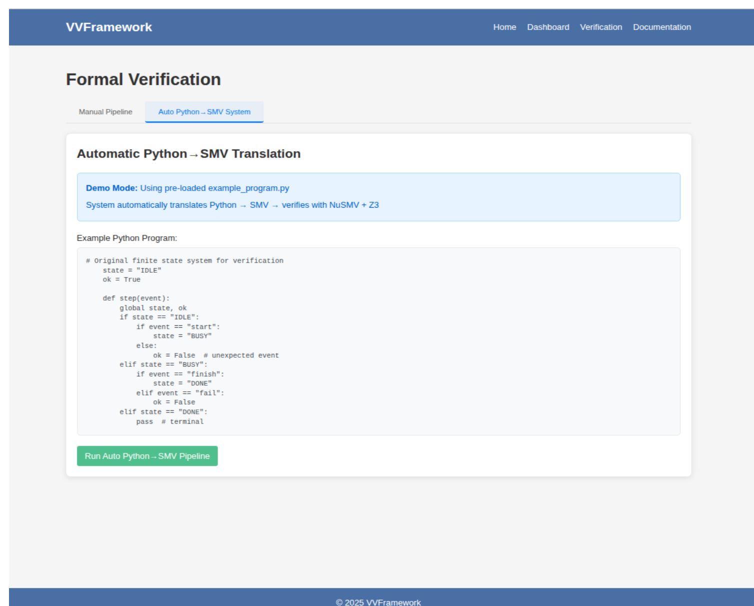


Figure 4.5: VVFramework Automatic Verifications Tab View

Below the tabs, a standardised results display area shows the verification results. Individual results for manual verifications and a summary for the automated pipeline are displayed here. The interface also includes clear error messages for troubleshooting and an option to download the results in JSON format. This interface combines the manual and automated systems into a single experience.

4.2.3 Underlying System Components

The Documentation Interface (documentation.html) is currently a placeholder that is being prepared for future development.

The entire front-end is based on a Base Template System (base.html), which ensures the consistency of all pages. This template contains a standardised header for navigation, a footer with framework information and a base for interactive elements built with Vue.js and Axios. The CSS is modular and designed for a responsive user experience.

4.3 Backend Service Implementation

The backend services of the VVFramework form the core of the system's verification and validation functions. They are responsible for orchestrating user requests, translating Python programmes into formal models and interacting with verification engines. Each service is encapsulated as a Docker container and provides a REST API to ensure modularity and interoperability within the microservices architecture. This section describes the implementation of the main backend components, including the Python Service Agent, which acts as an orchestrator, the Translation Agent, that generates SMV models, and the NuSMV and Z3- based verification agents. Together, these services enable VVFramework's hybrid workflow, which supports both manual and automated verification pipelines.

4.3.1 Python Service Agent (Orchestrator)

The Python Service Agent was implemented in Flask and acts as the central node of the framework. It receives user input and determines the workflow, whether this is manual or automated. The agent then delegates translation requests to the translation agent and verification tasks to the NuSMV or Z3 agents. A key function is to normalise all responses from the various services into consistent JSON objects for the interface.

4.3.2 Translation Agent (Python $\rightarrow$ SMV)

The Translation Agent is responsible for converting Python programmes, Listing 4.1, into SMV models, Listing 4.2. Implements a translator that is specially designed for finite state systems and converts a program with a predefined set of states into its formal equivalent. The translator handles the representation of variables (integers, Boolean values, and enumerations) and Boolean expressions. It is important to note that this is not a general-purpose parser as its functionality is focussed on the specific case study of state machines.

```python
# Example Python program for verification
state = "IDLE"
ok = True

def step(event):
    global state, ok

    if state == "IDLE":
        if event == "start":
            state = "BUSY"
        else:
            ok = False  # Unexpected event at Idle
    elif state == "BUSY":
        if event == "finish":
            state = "DONE"
        elif event == "fail":
            ok = False
    elif state == "DONE":
        pass  # terminal state
```

Listing 4.1: Input Python Code

```
MODULE main
VAR
  state : {IDLE, BUSY, DONE};
  ok    : boolean;
  event : {start, other, finish, fail, any};

ASSIGN
  init(state) := IDLE;
  init(ok)    := TRUE;

  next(state) :=
    case
      (state = IDLE & event = start) : BUSY;
      (state = IDLE & event = other) : IDLE;
      (state = BUSY & event = finish) : DONE;
      (state = BUSY & event = fail) : BUSY;
      TRUE : state;
    esac;
```

Listing 4.2: Generated SMV model

This example illustrates how Python loop semantics are translated into SMV state transitions.

4.3.3 NuSMV Agent

The NuSMV agent runs as a Docker container on which NuSMV is preinstalled. It provides an API that includes the submitModel, addSpecification, and run_nusmv endpoints. This agent receives an SMV model and a CTL/LTL property as input, as can be seen in Listing 4.3 and returns a verification result, shown in Listing 4.4 along with a counterexample trace if the property fails.

```
{
  "model": "MODULE main\nVAR state: {IDLE, BUSY, DONE};",
  "property": "AG ok"
}
```

Listing 4.3: API call

```
{
  "result": "false",
  "status": "verification_failed",
  "counterexample": "State 1.1: state = IDLE, ok = TRUE, event = other\
   nState 1.2: state = IDLE, ok = FALSE"
}
```

Listing 4.4: Verification output

4.3.4 Z3 Agent

The Z3 Agent is also encapsulated in a Docker container and is used to check invariants and solve constraint satisfaction problems. Provides the z3-proof and manual/z3 endpoints. Accepts logical constraints in the SMT-LIB2 format, below in Listing 4.5 we can see one example of that and also its responses in Listing 4.6.

```
    (declare-const x Int)
    (assert (> x 0))
    (assert (forall ((s String) (e String))
      (implies (and (= s "IDLE") (= e "other"))
        (= ok false))))
    (check-sat)
```

Listing 4.5: SMT-LIB2 input

```
{
  "result": "sat",
  "model": {"x": 1},
  "verification_status": "property_violated"
}
```

Listing 4.6: JSON Result

This confirms that the invariant is valid within the given restrictions.

4.4 Communication Workflows and Data Exchange Model

The services of the VVFramework communicate mainly via HTTP REST APIs. The main application, Flask, acts as a central gateway that forwards requests from the web interface to the corresponding microservices and, at the same time, provides a standardised interface for the front-end. In addition to direct API calls, artefacts such as .smv files are passed between the services via shared Docker volumes. This dual-channel approach enables a key architectural principle: decoupling, where services can work independently of each other without having to know the specific internal mechanisms of other components.

4.4.1 Verification workflows

The framework supports three different verification workflows, each with its own communication flow.

Manual verification workflows: These two workflows give the user direct control over the formal tools.

- Manual NuSMV flow: A user submits an SMV model and properties via the frontend. The request is forwarded from the main API to the python-flask service. This service writes the model to a temporary file on the shared volume and executes NuSMV via a sub-process. The filtered verification results are then returned to the frontend.
- Manual Z3 Flow: Similarly, a user deploys Z3 code and properties. The python-flask service receives the request, imports and executes the Z3 solver with the user's input and returns the proof results to the frontend via the API gateway.

Automated pipeline flow: This flow represents the core of the automation of the VVFramework. The process is triggered by a single user action that initiates a complex, orchestrated sequence of internal processes. The python-flask service first translates the user's Python code into an SMV model and then makes internal calls to the NuSMV and Z3 agents to perform parallel verification. Finally, it compares the results of both engines to ensure consistency before returning a combined summary to the frontend. This unified pipeline demonstrates the system's ability to automate complex verification tasks.

4.5 Microservices Architecture and Deployment

The VVFramework implements a microservices-based architecture where each verification component is encapsulated in separate Docker containers. This architecture, as shown in Figure 4.6, enables:

- Tool isolation: Each verifier (NuSMV, Z3) runs in its own environment.
- Scalability: Services can be scaled independently of each other.
- Maintainability: Updates without impacting other components.
- Interoperability: Communication via standard REST APIs.

4.5.1 Containers Deployed

The VVFramework microservices architecture is put into operation by running the defined Docker containers. When deploying the system, the containers are visualised as shown in Figure 4.7 running. This layout confirms that each service agent (from the web interface to the verification engines) runs independently, which is essential for the modularity of the framework.

Container	**Image**	**Port**	**Function**
python-flask	framework-python-flask	:5000	Orchestrator, Python translation $\rightarrow$ SMV
z3	framework-z3	:5001	Verification SMT, proof obligations
nusmv	badouralix/nusmv	-	Model checking, verification CTL/LTL

Table 4.1: Description of containers and their functions

To better understand the function of each component, Table 4.1 contains a brief description of each container, explaining its specific role in the verification process.

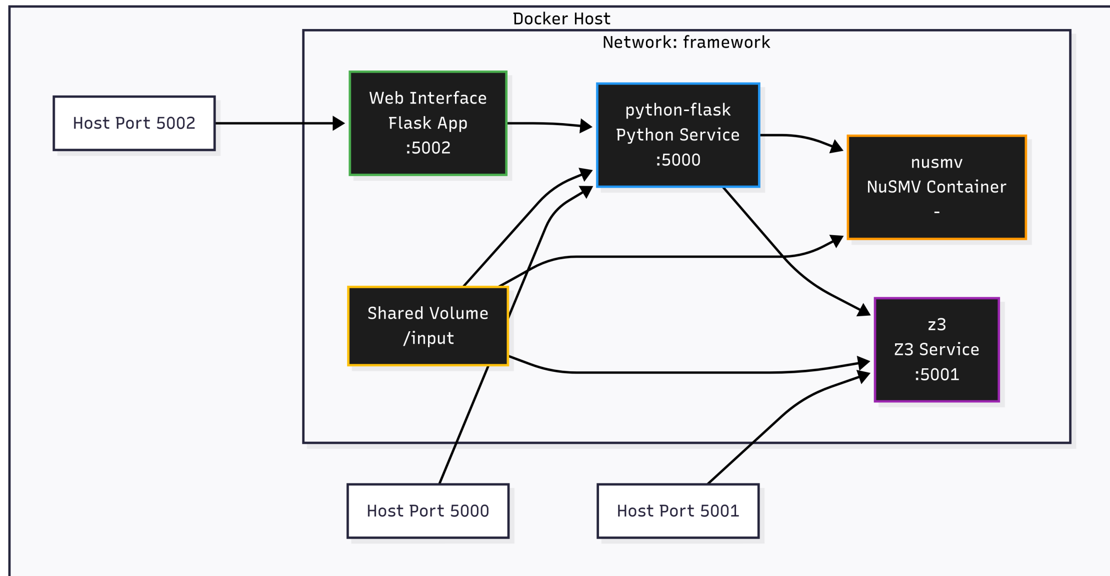


Figure 4.6: Microservices-based Architecture

```
>_ docker ps --format "table {{.Names}}\t{{.Image}}\t{{.Status}}\t
   {{.Ports}}"
NAMES           IMAGE                      STATUS        PORTS
z3              framework-z3               Up 47 hours   0.0.0.0:5001-
>5000/tcp, [::]:5001->5000/tcp
python-flask    framework-python-flask     Up 4 days     0.0.0.0:5000-
>5000/tcp, [::]:5000->5000/tcp
nusmv           badouralix/nusmv           Up 5 days
```

Figure 4.7: Running Containers Status

4.5.2 Docker Compose Configuration

As seen in the previous Subsection 4.5.1, we have that each service is encapsulated in its own Docker containers. The deployment is orchestrated through the Docker Compose file, as shown in Listing 4.7, whose format is YAML. This file is the orchestration mechanism that defines and executes the multicontainer application. It describes the three main services: Python-Flask, NuSMV, and Z3. To facilitate communication between all services, the file uses shared volumes. This allows different containers to access and share files, such as the SMV input models, which are essential for the verification workflow.

```
1 services:
2   python-flask:        # Principal service
3     container_name: python-flask
4     build:
5       context: ./services/python-service
6       dockerfile: Dockerfile
```

```
7     volumes:
8       - .\textit{/input}:\textit{/input} # Shared volume
9       - /var/run/docker.sock:/var/run/docker.sock # Docker socket
10    ports:
11      - "5000:5000"
12    environment:
13      FLASK_APP: app.py
14    working_dir: /app
15    command: python3 -m flask run --host=0.0.0.0
16
17  nusmv:                 # NuSMV service
18    container_name: nusmv
19    image: badouralix/nusmv
20    volumes:
21      - .\textit{/input}:\textit{/input}  # Shared volume
22    command: sh -c "nusmv -source \textit{/input}/commands.smv"
23    stdin_open: true
24    tty: true
25    depends_on:
26      - python-flask
27
28  z3:                    # Z3 service
29    container_name: z3
30    build:
31      context: ./services/z3-service
32      dockerfile: Dockerfile
33    ports:
34      - "5001:5000"
35    volumes:
36      - .\textit{/input}:/app/z3\textit{/input}
37    restart: always
```

Listing 4.7: Docker Compose File for VVFramework

The choice of Docker Compose was fundamental to the architecture of the VVFramework, as it is not only a convenient tool, but also an integral mechanism that manages the orchestration and connection of the verification agents. Its use guarantees the reproducibility of experiments with a single command while ensuring the modularity and isolation of each tool. Moreover, its extensible design corresponds to the principle of a hybrid framework, which facilitates the future integration of new agents and directly contributes to lowering the entry barrier for formal verification.

4.5.3 Dockerfile Configuration

As shown in the previous subsection 4.5.2 in the configuration of the Docker Compose file, dockerfile is used as a tool for deploying the various microservices. Dockerfiles are important for the architecture of the VVFramework, as they form the basis for the reproducibility and isolation of the individual services. While Docker Compose manages the overall system, each **Dockerfile** defines the exact build process for an individual microservice, such as the Z3, NuSMV Agent, or the Python Orchestrator. This approach was chosen to ensure a consistent and portable environment.

Python service (Flask): The first Dockerfile shown in the Listing 4.8 sets up an environment for the main Python service. Initially, a base image of Python version 3.12-slim is used, which is ideal due to its small size. After defining the working directory, pip is updated, and the requirements.txt file is copied to install the project dependencies. Once all dependencies are in place, all remaining application files are copied into the container. Finally, the container is configured to run the Flask application on startup, making it the central orchestrator of your microservices.

```
# Using Python slim image is a good choice for smaller footprint
FROM python:3.12-slim

# Set the working directory inside the container
WORKDIR /app

# Update pip to the latest version
RUN pip install --upgrade pip

# Copy the file of dependencies from project root
COPY requirements.txt requirements.txt

# Install project dependencies
RUN pip3 install -r requirements.txt

# Copy files from the current host directory to the container
    working directory
COPY . .

# Execute the application with the required entrypoints
CMD [ "python3", "-m", "flask", "run", "--host=0.0.0.0" ]
```

Listing 4.8: Python Orchestrator Dockerfile

NuSMV service: The Dockerfile for the NuSMV service starts with an Ubuntu base image, see Listing 4.9. This file is responsible for preparing the environment for compiling and installing NuSMV from the source code. For this purpose, it first updates the system and installs the necessary tools such as gcc, make, and bison. The NuSMV file is then downloaded, unpacked, and compiled in order to install the tool in the container. Finally, the working directory is defined, and the standard command is configured to start NuSMV in interactive mode and enable model checking.

```
 # Use an Ubuntu base image
FROM ubuntu:22.04

# Update the system and install the necessary dependencies
RUN apt-get update && \
     # apt-get install -y wget gcc make bison flex
     apt-get install -y wget gcc g++ make bison flex cmake libxml2-
    utils && \
     apt-get clean && \
     rm -rf /var/lib/apt/lists/*

# Download NuSMV (you can change the URL according to the required
    version)
RUN wget http://nusmv.fbk.eu/distrib/NuSMV-2.6.0.tar.gz && \
```

```
13     tar -xzvf NuSMV-2.6.0.tar.gz && \
14     rm NuSMV-2.6.0.tar.gz && \
15     cd NuSMV-2.6.0 && \
16     # ./configure && \
17     mkdir build && \
18     cd build && \
19     cmake .. && \
20     make && \
21     make install
22 # Set the PATH environment variable to make NuSMV accessible
23 ENV PATH="${PATH}:/NuSMV-2.6.0/bin"
24 # Establishing the working directory
25 WORKDIR /app/nusmv
26 # Default command when starting the container
27 CMD ["nusmv", "-int"]
```

Listing 4.9: NuSMV Agent Dockerfile

Z3 service: The third Dockerfile shown in the Listing 4.10 creates the environment for the Z3 service. Like the Python service, it uses a python:3.12-slim image. The file installs the required dependencies at the system level before the Python libraries specified in requirements.txt are copied and installed. The application files are then copied into the container. The Dockerfile shares port 5000 to enable communication with the service and finally sets the start command to run the uvicorn application on this port and configure it as an API server ready to receive requests for logical problem solving.

```
1  # Using Python slim image is a good choice for smaller footprint
2 FROM python:3.12-slim
3
4 # Set the working directory inside the container
5 WORKDIR /app/z3
6
7 # Install system dependencies required for Z3
8 RUN apt-get update && apt-get install -y \
9     curl \
10    && rm -rf /var/lib/apt/lists/*
11
12 # Copy requirements from project root
13 COPY requirements.txt .
14
15 # Install Python dependencies from central requirements.txt
16 RUN pip3 install --no-cache-dir -r requirements.txt
17
18 # Copy the necessary files to the container
19 COPY solver.py .
20 COPY main.py .
21 COPY contracts_example.py .
22 # Expose port for the API Flask and FastAPI application
23 EXPOSE 5000
24 # Set the command to run the service
25 # CMD ["python3", "app.py"]
26 CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "5000"]
```

Listing 4.10: Z3 Agent Dockerfile

While the Python-Flask and Z3 services are created from their respective local Dockerfiles, the NuSMV service uses a different approach, as can be seen in Listing 4.7. Instead of compiling NuSMV from source code, this service relies on a pre-built, existing image (badouralix/nusmv) that is downloaded directly from a container repository. This does not mean that the associated Dockerfile is not used, it was simply decided to make both options available. This decision avoids the lengthy compilation process and ensures a pre-configured verification environment, which significantly speeds up the deployment of the framework.

The use of Dockerfiles in the framework serves several important purposes. Firstly, they enable **controlled environment definition** by specifying the exact base images and dependencies required for each service. For example, the NuSMV agent is built with its specific version of the solver, separate from the Python libraries required by the orchestrator. This leads to the second advantage: **isolation of dependencies**. By containing each tool in its own environment, Dockerfiles prevent conflicts and ensure that changes to one service do not affect another. And finally, they are crucial for **reproducibility of experiments**, as a Dockerfile can be version controlled and used to recreate the same environment at any time, allowing others to reproduce the results of the work.

4.5.4 Python Dependencies

As shown in the previous subsection 4.5.3 in the Dockerfile configuration is used as a requirement file for deploying all Python dependencies.

Requirements File: This file as can be seen in Listing 4.11, centralises all Python dependencies that are shared by the framework's services. Specifies libraries such as Flask and FastAPI, which are used to create backend API services. It also includes tools such as uvicorn for running web servers, the Docker library for interacting with containers, and z3-solver for the direct integration of this engine. By defining all dependencies in one place, the file ensures that all Python-based services have the necessary tools to run conflict-free.

```
1    Flask==3.0.3
2    docker==7.1.0
3    fastapi==0.116.1
4    uvicorn==0.35.0
5    z3-solver==4.15.3.0
6    requests==2.31.0
```

Listing 4.11: Requirements File

4.5.5 Conclusions

Together, the Python dependencies file, the Docker Compose file, and the Dockerfiles are essential for dependency management and service orchestration in the VVFramework. Together, they ensure that the runtime environment is consistent, reproducible, and easy to implement, making it a modular and portable hybrid verification framework.

4.6 Shared Volume Architecture for Decoupling

The */input* directory serves as a shared data carrier that enables cross-service communication via the file system within the VVFramework. This approach is central to the system design as it allows different containers to access and share data without direct API calls, promoting

a decoupled architecture. The directory contains generated models and command files for both, automated and manual workflows.

4.6.1 Auto-generated Files

In the automated workflow the Python service translates a user's code with **py2smv.py**. The resulting SMV model and verification commands are then written to the **auto_model.smv** and **auto_commands.smv** files in the shared */input* directory. The NuSMV service can then read these files directly from the mounted data carrier in order to perform the verification.

4.6.2 Manual Files:

Similarly, for the manual workflow, the Python service writes the model and properties provided by the user to the **manual_model.smv** and **manual_commands.smv** files in the same shared location. This allows the NuSMV service to process the files and perform a manual verification run. This method enables a clean separation of the areas in which the translation and web services are responsible for file creation, and the verification service is responsible for file processing.

4.6.3 Benefits

This shared volume approach offers advantages. It ensures that the services are decoupled as they communicate via a file system interface and not via direct API calls. It also facilitates troubleshooting as the generated files persist across the service calls and can be checked at any time. Using Docker Compose to mount the */input* directory in all relevant containers ensures that the entire process is reproducible and easy to manage. This design provides clear service boundaries while supporting complex and cohesive verification workflows.

Chapter 5

Validation of the VVFramework

This chapter focuses on the validation of the VVFramework through a series of formal verification case studies that demonstrate its functionality and effectiveness using a two-approach methodology. The validation focuses on testing the usefulness of the tool both as a versatile platform for manual verification and as an innovative solution for automatic code translation. This hybrid approach allows an honest assessment of the framework's architecture and its potential to democratise formal methods, and also demonstrates the usability of the framework in a practical environment by showing its ability to handle different systems and properties.

The validation of the VVFramework was carried out using two complementary strategies. First, the manual verification function of the framework was used to analyse three representative systems to demonstrate its ability to handle different properties and detect errors in different contexts. By manually modelling each system, we can tightly control the state space and properties, allowing direct and transparent evaluation of the framework for processing and presentation of results. The validation used both NuSMV for model checking and Z3 for SMT resolution, allowing the integration of different verification paradigms in a single environment. Secondly, the automated translation pipeline was validated using a single case study, which served as a crucial proof-of-concept and demonstrated the feasibility of automation in the verification chain. All processes were carried out via the framework's standardised web interface, which confirmed its user-friendliness and architectural consistency. All interactions were handled through a unified web interface, which has proven effective because it provides a seamless user experience regardless of whether the user performs a manual or automated workflow.

5.1 Evaluation Metrics

To assess the effectiveness of VVFramework, the following evaluation metrics were considered:

- Correctness: Whether the translated SMV model preserves the intended semantics of the original Python program.
- Verification Success Rate: Proportion of properties (safety, liveness, invariants) that can be successfully verified using NuSMV and Z3.
- Counterexample Generation: Ability of the framework to automatically generate meaningful counterexamples when properties are violated.

These metrics enable a more comprehensive evaluation, moving beyond functional correctness.

5.2 Case studies for manual verification

To demonstrate the versatility of the framework as a tool for formal verification, three systems were modelled and analysed manually in handling different system types and property classes.

5.2.1 Case study: Traffic Light Control

The first case study concerns a simple Traffic Light Control system that is represented in NuSMV model, as can be seen in Listing 5.1. The system switches between the states RED, GREEN, and YELLOW with certain time specifications.

```
-- Traffic Light Control
MODULE main
VAR
  state : {RED, GREEN, YELLOW};
  timer : 0..10;

ASSIGN
  init(state) := RED;
  init(timer) := 0;

  next(state) := case
    state = RED & timer >= 3    : GREEN;
    state = GREEN & timer >= 5 : YELLOW;
    state = YELLOW & timer >= 2: RED;
    TRUE : state;
  esac;

  next(timer) := case
    state = RED & timer >= 3    : 0;
    state = GREEN & timer >= 5 : 0;
    state = YELLOW & timer >= 2: 0;
    timer < 10 : timer + 1;
    TRUE : timer;
  esac;
```

Listing 5.1: Traffic Light in NuSMV model

The verification focuses on a combination of properties Listing 5.2: a safety property that ensures that the system never enters an undefined state, an activity property that ensures that the traffic light eventually returns to the RED state, and timing properties that confirm that each state maintains its minimum duration.

```
-- Safety property: System never enters an indefinite state
CTLSPEC AG (state = RED | state = GREEN | state = YELLOW)
-- Liveness property: From any state, it eventually returns to Red
CTLSPEC AG (AF state = RED)
-- Timing property: Each state has minimal duration
CTLSPEC AG (state = RED -> timer <= 3)
CTLSPEC AG (state = GREEN -> timer <= 5)
CTLSPEC AG (state = YELLOW -> timer <= 2)
```

Listing 5.2: CTLSPEC properties evaluated in Traffic Light

The safety, and timing properties were defined in the CTL language and successfully verified by NuSMV using the manual interface of the VVFramework, as shown in Figure 5.1, confirming that all five properties hold TRUE, as can be seen in Figure 5.2. To see full page, go to Appendix A. This case study confirms the framework's ability to demonstrate the correctness of a simple, deterministic system.

VVFramework Home Dashboard Verification Documentation

Formal Verification

Manual Pipeline Auto Python→SMV System

Manual Verification System

Select Verification Tool:

NuSMV (Model Checking)

NuSMV Model Checking

SMV Model:

```
MODULE main
VAR
  state : {RED, GREEN, YELLOW};
  timer : 0..10;

ASSIGN
  init(state) := RED;
  init(timer) := 0;

  next(state) := case
    state = RED & timer >= 3   : GREEN;
    state = GREEN & timer >= 5 : YELLOW;
    state = YELLOW & timer >= 2: RED;
    TRUE : state;
  esac;

  next(timer) := case
    state = RED & timer >= 3   : 0;
    state = GREEN & timer >= 5 : 0;
    state = YELLOW & timer >= 2: 0;
    timer < 10 : timer + 1;
    TRUE : timer;
  esac;
```

CTL Specifications:

```
CTLSPEC AG (state = RED | state = GREEN | state = YELLOW)
CTLSPEC AG (AF state = RED)
CTLSPEC AG (state = RED -> timer <= 3)
CTLSPEC AG (state = GREEN -> timer <= 5)
CTLSPEC CTLSPEC AG (state = YELLOW -> timer <= 2)
```

Remove Remove Remove Remove Remove

Add CTL Spec

Verify with NuSMV

Figure 5.1: Traffic Light Control with CTL Specifications in Manual SMV Verification

On the other hand, is checking the Traffic Light using manual verification flow for the Z3 agent with the SMT solver. This process requires the user to enter the logic of the system using Python syntax, which includes the available Z3 functions, and then define a list of specific properties for the system to evaluate, as can be seen in Figure 5.3.

The verification is divided into two different parts within the interface: the Z3 code field and the properties field.

Manual SMV Verification Results

Status: Verified

Details:


```
{
  "tool": "nusmv",
  "execution_time": 1758808139.957,
  "clean_output": "-- specification AG ((state = RED | state = GREEN) | state = YELLOW)  is true\n-- specification AG (AF state = RED)  is
true\n-- specification AG (state = RED -> timer <= 3)  is true\n-- specification AG (state = GREEN -> timer <= 5)  is true\n-- specification AG
(state = YELLOW -> timer <= 2)  is true",
  "model_path": "/input/manual_model.smv",
  "specifications_count": 5
}
```


New Verification | Download Results

Figure 5.2: Traffic Light Control in Manual SMV Verification Result

Formal Verification

Manual Pipeline | Auto Python→SMV System

Manual Verification System

Select Verification Tool:

Z3 (SMT Solver)

Z3 SMT Solver

Code/Constraints:

```
# Simple traffic light system with 3 states

# State variables
state = Int('state')  # 0=RED, 1=GREEN, 2=YELLOW
timer = Int('timer')

# Next state variables
next_state = Int('next_state')
next_timer = Int('next_timer')

# Domain restrictions
solver.add(state >= 0, state <= 2)
solver.add(timer >= 0, timer <= 5)
solver.add(next_state >= 0, next_state <= 2)
solver.add(next_timer >= 0, next_timer <= 5)

# Traffic light transition logic
# RED (0) -> GREEN (1) when timer=0
solver.add(Implies(And(state == 0, timer == 0), And(next_state == 1, next_timer == 5)))

# GREEN (1) -> YELLOW (2) when timer=0
solver.add(Implies(And(state == 1, timer == 0), And(next_state == 2, next_timer == 2)))

# YELLOW (2) -> RED (0) when timer=0
solver.add(Implies(And(state == 2, timer == 0), And(next_state == 0, next_timer == 3)))

# The timer decrements when > 0
solver.add(Implies(timer > 0, next_timer == timer - 1))
solver.add(Implies(timer > 0, next_state == state))
```

Properties/Assertions:

```
Implies(state == 0, timer <= 3)
```

Remove

Add Property

Verify with Z3

Figure 5.3: Traffic Light Control in Manual Z3 Verification

Z3 code (System Definition): The user specifies the core logic and the boundary conditions of the system. In the example of the simple traffic light system, this code defines the state variables (`state`, `timer`), specifies the domain constraints (e.g. $0 \leq$ `state` ≤ 2) and determines the transition logic for the system states (e.g. `RED` $\rightarrow$ `GREEN` only if `timer` $= 0$). The code block is executed by the agent to define the entire problem context for the Z3 solver.

Properties (Verification Targets): The user specifies a list of boolean expressions. The system then takes each property and performs an eval() operation within the context in which the state variables are already defined by the Z3 code. This determines whether the property is satisfiable (sat) or unsatisfiable (unsat) with respect to the defined system constraints.

The primary objective of the Z3 test is to check for fulfilment or invariance. To demonstrate the ability of the Z3 Agent to solve logical constraints, two important property tests were performed. The first scenario confirmed the satisfiability of the property:

`Implies(state==0, timer` ≤ 3), Figure 5.4 shows the result that confirms the **satisfiability** (sat) of the implication property.

While the second proved the **unsatisfiability** (unsat) of the assertion:

`And(state==0, timer==10)` regarding an impossible timer value. as can be seen in Figure 5.5. The verification results shown the ability of the system to analyse both valid and impossible state assertions within the formally defined model.

5.2.2 Case study: Login System

The second case study concerns a Login System, as can be seen in Listing 5.3 with a limited number of authentication attempts before the user is locked out. This system was selected to test the framework's ability to detect security vulnerabilities.

```
1  -- Login System
2  MODULE main
3  VAR
4    authenticated : boolean;
5    attempts : 0..3;
6    password_correct : boolean;
7    locked : boolean;
8
9  ASSIGN
10   init(authenticated) := FALSE;
11   init(attempts) := 0;
12   init(password_correct) := {TRUE, FALSE}; -- Not deterministic
13   init(locked) := FALSE;
14
15   next(authenticated) := case
16     locked : FALSE;
17     password_correct & attempts < 3 : TRUE;
18     TRUE : authenticated;
19   esac;
20
21   next(attempts) := case
22     locked : attempts;
23     !password_correct & attempts < 3 : attempts + 1;
```

```
24    password_correct : 0; -- Reset on success
25    TRUE : attempts;
26  esac;
27
28  next(locked) := case
29    attempts >= 3 : TRUE;
30    password_correct : FALSE; -- Reset on success
31    TRUE : locked;
32  esac;
33
34  next(password_correct) := {TRUE, FALSE}; -- Model External Entry
```

Listing 5.3: Login System in NuSMV model

```
# Domain restrictions
solver.add(state >= 0, state <= 2)
solver.add(timer >= 0, timer <= 5)
solver.add(next_state >= 0, next_state <= 2)
solver.add(next_timer >= 0, next_timer <= 5)

# Traffic light transition logic
# RED (0) -> GREEN (1) when timer=0
solver.add(Implies(And(state == 0, timer == 0), And(next_state == 1, next_timer == 5)))

# GREEN (1) -> YELLOW (2) when timer=0
solver.add(Implies(And(state == 1, timer == 0), And(next_state == 2, next_timer == 2)))

# YELLOW (2) -> RED (0) when timer=0
solver.add(Implies(And(state == 2, timer == 0), And(next_state == 0, next_timer == 3)))

# The timer decrements when > 0
solver.add(Implies(timer > 0, next_timer == timer - 1))
solver.add(Implies(timer > 0, next_state == state))
```

Properties/Assertions:

Implies(state == 0, timer <= 3) Remove

Add Property

Verify with Z3

Manual Z3 Verification Results

Status: Verified

Details:


```
{
  "tool": "z3",
  "execution_time": 1758827152.701,
  "code_processed": "# Simple traffic light system with 3 states\n\n# State variables\nstate = Int('state')  # 0=RED, 1=GREEN, 2=YELLOW\ntimer = Int('timer')\n\n# Next state variables\nnext_state = Int('next_state')\nnext_timer = Int('next_timer')\n\n# Domain restrictions\nsolver.add(state >= 0, state <= 2)\nsolver.add(timer >= 0, timer <= 5)\nsolver.add(next_state >= 0, next_state <= 2)\nsolver.add(next_timer >= 0, next_timer <= 5)\n\n# Traffic light transition logic\n# RED (0) -> GREEN (1) when timer=0\nsolver.add(Implies(And(state == 0, timer == 0), And(next_state == 1, next_timer == 5)))\n\n# GREEN (1) -> YELLOW (2) when timer=0\nsolver.add(Implies(And(state == 1, timer == 0), And(next_state == 2, next_timer == 2)))\n\n# YELLOW (2) -> RED (0) when timer=0\nsolver.add(Implies(And(state == 2, timer == 0), And(next_state == 0, next_timer == 3)))\n\n# The timer decrements when > 0\nsolver.add(Implies(timer > 0, next_timer == timer - 1))\nsolver.add(Implies(timer > 0, next_state == state))",
  "properties_checked": 1,
  "properties_list": [
    "Implies(state == 0, timer <= 3)"
  ],
  "satisfiable": "sat",
  "model": {
    "next_state": "1",
    "next_timer": "0",
    "state": "1",
    "timer": "1"
  }
}
```


New Verification Download Results

Figure 5.4: Traffic Light Control in Manual Z3 SAT Result

```
# Next state variables
next_state = Int('next_state')
next_timer = Int('next_timer')

# Domain restrictions
solver.add(state >= 0, state <= 2)
solver.add(timer >= 0, timer <= 5)
solver.add(next_state >= 0, next_state <= 2)
solver.add(next_timer >= 0, next_timer <= 5)

# Traffic light transition logic
# RED (0) -> GREEN (1) when timer=0
solver.add(Implies(And(state == 0, timer == 0), And(next_state == 1, next_timer == 5)))

# GREEN (1) -> YELLOW (2) when timer=0
solver.add(Implies(And(state == 1, timer == 0), And(next_state == 2, next_timer == 2)))

# YELLOW (2) -> RED (0) when timer=0
solver.add(Implies(And(state == 2, timer == 0), And(next_state == 0, next_timer == 3)))

# The timer decrements when > 0
solver.add(Implies(timer > 0, next_timer == timer - 1))
solver.add(Implies(timer > 0, next_state == state))
```

Properties/Assertions:

And(state == 0, timer == 10) Remove

Add Property

Verify with Z3

Manual Z3 Verification Results

Status: Not Verified

Details:


```
{
  "tool": "z3",
  "execution_time": 1758827254.383,
  "code_processed": "# Simple traffic light system with 3 states\n\n# State variables\nstate = Int('state')  # 0=RED, 1=GREEN, 2=YELLOW\ntimer = Int('timer')\n\n# Next state variables\nnext_state = Int('next_state')\nnext_timer = Int('next_timer')\n\n# Domain restrictions\nsolver.add(state >= 0, state <= 2)\nsolver.add(timer >= 0, timer <= 5)\nsolver.add(next_state >= 0, next_state <= 2)\nsolver.add(next_timer >= 0, next_timer <= 5)\n\n# Traffic light transition logic\n# RED (0) -> GREEN (1) when timer=0\nsolver.add(Implies(And(state == 0, timer == 0), And(next_state == 1, next_timer == 5)))\n\n# GREEN (1) -> YELLOW (2) when timer=0\nsolver.add(Implies(And(state == 1, timer == 0), And(next_state == 2, next_timer == 2)))\n\n# YELLOW (2) -> RED (0) when timer=0\nsolver.add(Implies(And(state == 2, timer == 0), And(next_state == 0, next_timer == 3)))\n\n# The timer decrements when > 0\nsolver.add(Implies(timer > 0, next_timer == timer - 1))\nsolver.add(Implies(timer > 0, next_state == state))",
  "properties_checked": 1,
  "properties_list": [
    "And(state == 0, timer == 10)"
  ],
  "satisfiable": "unsat",
  "message": "No satisfying assignment found (UNSAT)"
}
```


New Verification Download Results

Figure 5.5: Traffic Light Control in Manual Z3 UNSAT Result

The check focuses on four properties, as can be seen in Listing 5.4: a security property that ensures that authentication only takes place with the correct password, an abort property that shows that the system is locked out after failed attempts, a recovery system after correct authentication and a liveness property if authentication is possible.

```
-- Security property: Non -authentication without correct password
CTLSPEC AG (authenticated -> password_correct)
-- Termination property: After 3 failed attempts, system is blocked
CTLSPEC AG (attempts = 3 -> AX locked)
-- Recovery property: System can be unlocked after correct password
CTLSPEC AG (locked & password_correct -> AF !locked)
-- Liveness property: It is always possible to eventually authenticate
CTLSPEC AG (EF authenticated)
```

Listing 5.4: CTLSPEC properties evaluated in Login System

The logical system model was verified using the manual interface of the VVFramework, as shown in Figure 5.6. While some properties were successfully validated, the critical security property was found to be invalid. This result was particularly significant as it demonstrated the core capability of the framework to not only validate correct behaviour, but also to effectively detect and diagnose faults within a system.

The NuSMV engine automatically generated a counterexample, shown in Figure 5.7, which provides an execution trace detailing the sequence of events that led to the violation. This log confirmed that the system was able to authenticate a user with an incorrect password under certain conditions, confirming the utility of the framework as a debugging and validation tool.

5.2.3 Case study: Mutual Exclusion Protocol

The third case study is a Mutual Exclusion Protocol, a classic problem in concurrent systems where two processes compete for a common critical section, shown in Listing 5.5.

```
-- Mutual Exclusion Protocol
MODULE main
VAR
  p1_state : {IDLE, WAITING, CRITICAL};
  p2_state : {IDLE, WAITING, CRITICAL};
  lock : boolean;
  turn : {P1, P2}; -- Fairness mechanism

ASSIGN
  init(p1_state) := IDLE;
  init(p2_state) := IDLE;
  init(lock) := FALSE;
  init(turn) := P1;

  -- Process 1 behavior
  next(p1_state) := case
    p1_state = IDLE & !lock : WAITING;
    p1_state = WAITING & !lock & turn = P1 : CRITICAL;
    p1_state = CRITICAL : IDLE;
    TRUE : p1_state;
  esac;

  -- Process 2 behavior
  next(p2_state) := case
    p2_state = IDLE & !lock : WAITING;
    p2_state = WAITING & !lock & turn = P2 : CRITICAL;
    p2_state = CRITICAL : IDLE;
    TRUE : p2_state;
  esac;

  -- Lock management
  next(lock) := case
    p1_state = WAITING & turn = P1 : TRUE;
    p2_state = WAITING & turn = P2 : TRUE;
    p1_state = CRITICAL : FALSE;
    p2_state = CRITICAL : FALSE;
    TRUE : lock;
  esac;

  -- Fair turn scheduling
  next(turn) := case
```

```
42     p1_state = CRITICAL : P2;
43     p2_state = CRITICAL : P1;
44     TRUE : turn;
45   esac;
```

Listing 5.5: Mutual Exclusion Protocol in NuSMV model

Manual Pipeline   Auto Python→SMV System

Manual Verification System

Select Verification Tool:

NuSMV (Model Checking)

NuSMV Model Checking

SMV Model:

```
MODULE main
VAR
  authenticated : boolean;
  attempts : 0..3;
  password_correct : boolean;
  locked : boolean;

ASSIGN
  init(authenticated) := FALSE;
  init(attempts) := 0;
  init(password_correct) := {TRUE, FALSE}; -- Not deterministic
  init(locked) := FALSE;

  next(authenticated) := case
    locked : FALSE;
    password_correct & attempts < 3 : TRUE;
    TRUE : authenticated;
  esac;

  next(attempts) := case
    locked : attempts;
    !password_correct & attempts < 3 : attempts + 1;
    password_correct : 0; -- Reset on success
    TRUE : attempts;
  esac;

  next(locked) := case
    attempts >= 3 : TRUE;
    password_correct : FALSE; -- Reset on success
    TRUE : locked;
  esac;
```

CTL Specifications:

CTLSPEC AG (authenticated -> password_correct)   Remove

CTLSPEC AG (attempts = 3 -> AX locked)   Remove

CTLSPEC AG (locked & password_correct -> AF !locked)   Remove

CTLSPEC AG (EF authenticated)   Remove

Add CTL Spec

Verify with NuSMV

Figure 5.6: Login System with CTL Specifications in Manual SMV Verification

Manual SMV Verification Results

Status: Not Verified

Details:


```
{
  "tool": "nusmv",
  "execution_time": 1758809126.024,
  "clean_output": "-- specification AG (attempts = 3 -> AX locked)  is true\n-- specification AG ((locked & password_correct) -> AF !locked)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 1.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 1.2 <-\n    attempts = 1\n  -> State: 1.3 <-\n    attempts = 2\n  -> State: 1.4 <-\n    attempts = 3\n  -- Loop starts here\n  -> State: 1.5 <-\n    password_correct = TRUE\n    locked = TRUE\n  -> State: 1.6 <-\n-- specification AG (authenticated -> password_correct)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 2.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 2.2 <-\n    attempts = 1\n    password_correct = TRUE\n  -> State: 2.3 <-\n    authenticated = TRUE\n    attempts = 0\n    password_correct = FALSE\n-- specification AG (EF authenticated)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 3.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 3.2 <-\n    attempts = 1\n  -> State: 3.3 <-\n    attempts = 2\n  -> State: 3.4 <-\n    attempts = 3",
  "model_path": "/input/manual_model.smv",
  "specifications_count": 4
}
```


Counterexample:


```
{
  "details": "-- specification AG (attempts = 3 -> AX locked)  is true\n-- specification AG ((locked & password_correct) -> AF !locked)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 1.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 1.2 <-\n    attempts = 1\n  -> State: 1.3 <-\n    attempts = 2\n  -> State: 1.4 <-\n    attempts = 3\n  -- Loop starts here\n  -> State: 1.5 <-\n    password_correct = TRUE\n    locked = TRUE\n  -> State: 1.6 <-\n-- specification AG (authenticated -> password_correct)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 2.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 2.2 <-\n    attempts = 1\n    password_correct = TRUE\n  -> State: 2.3 <-\n    authenticated = TRUE\n    attempts = 0\n    password_correct = FALSE\n-- specification AG (EF authenticated)  is false\n-- as demonstrated by the following execution sequence\nTrace Description: CTL Counterexample \nTrace Type: Counterexample \n  -> State: 3.1 <-\n    authenticated = FALSE\n    attempts = 0\n    password_correct = FALSE\n    locked = FALSE\n  -> State: 3.2 <-\n    attempts = 1\n  -> State: 3.3 <-\n    attempts = 2\n  -> State: 3.4 <-\n    attempts = 3",
  "message": "Property verification failed",
  "type": "property_violation"
}
```


New Verification | Download Results

Figure 5.7: Login System with CTL Specifications in Manual SMV Verification Result

This case was chosen to test the framework's ability to handle more complex concurrent behaviours and properties. The verification focussed on a combination of safety (mutual exclusion), liveness (no blocking), fairness, deadlock freedom, and turn fairness, shown in Listing 5.6.

```
1  -- Safety property: Mutual exclusion - Not two processes in critic
       simultaneously
2  CTLSPEC AG !(p1_state = CRITICAL & p2_state = CRITICAL)
3  -- Liveness property: If a process wants to enter, it can eventually
4  CTLSPEC AG (p1_state = WAITING -> AF p1_state = CRITICAL)
5  CTLSPEC AG (p2_state = WAITING -> AF p2_state = CRITICAL)
6  -- Fairness property: Both processes can eventually access
7  CTLSPEC AG (EF p1_state = CRITICAL)
8  CTLSPEC AG (EF p2_state = CRITICAL)
9  -- Deadlock freedom: There is always possible progress
10 CTLSPEC AG (EF (p1_state = CRITICAL | p2_state = CRITICAL))
11 -- Turn fairness: The turn alternate correctly
12 CTLSPEC AG (p1_state = CRITICAL -> AX turn = P2)
13 CTLSPEC AG (p2_state = CRITICAL -> AX turn = P1)
```

Listing 5.6: CTLSPEC properties evaluated in Mutual Exclusion Protocol

The VVFramework receive the Mutual Exclusion Protocol model, see Figure 5.8 successfully processed the NuSMV model and confirmed that the protocol correctly applies mutual exclusion, see Figure 5.9 while ensuring that both processes can ultimately access the critical section. This validation demonstrates the robustness of the framework in analysing non-trivial concurrent systems.

Select Verification Tool:

NuSMV (Model Checking)

NuSMV Model Checking

SMV Model:

```
  init(p2_state) := IDLE;
  init(lock) := FALSE;
  init(turn) := P1;

  -- Process 1 behavior
  next(p1_state) := case
    p1_state = IDLE & !lock : WAITING;
    p1_state = WAITING & !lock & turn = P1 : CRITICAL;
    p1_state = CRITICAL : IDLE;
    TRUE : p1_state;
  esac;

  -- Process 2 behavior
  next(p2_state) := case
    p2_state = IDLE & !lock : WAITING;
    p2_state = WAITING & !lock & turn = P2 : CRITICAL;
    p2_state = CRITICAL : IDLE;
    TRUE : p2_state;
  esac;

  -- Lock management
  next(lock) := case
    p1_state = WAITING & turn = P1 : TRUE;
    p2_state = WAITING & turn = P2 : TRUE;
    p1_state = CRITICAL : FALSE;
    p2_state = CRITICAL : FALSE;
    TRUE : lock;
  esac;

  -- Fair turn scheduling
  next(turn) := case
    p1_state = CRITICAL : P2;
```

CTL Specifications:

```
CTLSPEC AG !(p1_state = CRITICAL & p2_state = CRITICAL)
CTLSPEC AG (p1_state = WAITING -> AF p1_state = CRITICAL)
CTLSPEC AG (p2_state = WAITING -> AF p2_state = CRITICAL)
CTLSPEC AG (EF p1_state = CRITICAL)
CTLSPEC AG (EF p2_state = CRITICAL)
CTLSPEC AG (EF (p1_state = CRITICAL | p2_state = CRITICAL))
CTLSPEC AG (p1_state = CRITICAL -> AX turn = P2)
CTLSPEC AG (p2_state = CRITICAL -> AX turn = P1)
```

Remove Remove Remove Remove Remove Remove Remove Remove

Add CTL Spec

Figure 5.8: Mutual Exclusion Protocol in Manual SMV Verification

5.3 Validation of the automation pipeline

The second part of the validation focused on the main contribution of the framework: the automation of the modelling process.

5.3.1 Case study: A Finite State System

To validate the end-to-end functionality of the framework, a simple finite state system was used. The system models a process with three different states: IDLE, BUSY and DONE. This simple model was chosen because its behaviour is simple enough to be fully understood, yet complex enough to demonstrate important verification principles.

The validation was carried out in two phases, utilising the core architectural components of the VV framework.

CTL Specifications:

CTLSPEC AG !(p1_state = CRITICAL & p2_state = CRITICAL) Remove

CTLSPEC AG (p1_state = WAITING -> AF p1_state = CRITICAL) Remove

CTLSPEC AG (p2_state = WAITING -> AF p2_state = CRITICAL) Remove

CTLSPEC AG (EF p1_state = CRITICAL) Remove

CTLSPEC AG (EF p2_state = CRITICAL) Remove

CTLSPEC AG (EF (p1_state = CRITICAL | p2_state = CRITICAL)) Remove

CTLSPEC AG (p1_state = CRITICAL -> AX turn = P2) Remove

CTLSPEC AG (p2_state = CRITICAL -> AX turn = P1) Remove

Add CTL Spec

Verify with NuSMV

Manual NUSMV Verification Results

Status: Verified

Details:


```
{
  "tool": "nusmv",
  "execution_time": 1758825349.733,
  "clean_output": "-- specification AG !(p1_state = CRITICAL & p2_state = CRITICAL)  is true\n-- specification AG (p1_state = WAITING -> AF p1_state = CRITICAL)  is true\n-- specification AG (p2_state = WAITING -> AF p2_state = CRITICAL)  is true\n-- specification AG (EF p1_state = CRITICAL)  is true\n-- specification AG (EF p2_state = CRITICAL)  is true\n-- specification AG (EF (p1_state = CRITICAL | p2_state = CRITICAL)) is true\n-- specification AG (p1_state = CRITICAL -> AX turn = P2)  is true\n-- specification AG (p2_state = CRITICAL -> AX turn = P1)  is true",
  "model_path": "/input/manual_model.smv",
  "specifications_count": 8
}
```


New Verification Download Results

Figure 5.9: Mutual Exclusion Protocol in Manual SMV Verification Result

Automated verification pipeline:

The behaviour of the system was first recorded in a Python program, example_program.py, shown in Appendix B. This program served as input for the automated verification pipeline. The Python-to-SMV translator (py2smv.py), shown in Appendix C processed this code and generated a corresponding SMV model, auto_model.smv, as can be seen in Appendix D. This automatic translation step is an important contribution of the framework as it simplifies the initial modelling process for developers.

Once the model has been created, the orchestration layer of the VVFramework passes it to the NuSMV verification engine. A critical property, CTLSPEC AG ok, was specified to verify that the system remains in a safe state at all times. The result of the check was INVALID, along with a detailed counterexample that showed a path where the condition ok=FALSE could be reached after an error event from the BUSY state. This result confirmed that the automated pipeline is effective in both translating high-level code and detecting subtle errors.

Manual verification and cross-validation:

The automatic check produced an INVALID result and provided a counterexample that pointed to a possible error in the model. To check the consistency of the process, an equivalent model was entered manually in Z3 (contracts_example.py), see Appendix E, via the framework interface, shown in Figure 5.10. The Z3 results matched those of NuSMV, also showed the property violation and provided an equivalent counterexample, confirming

that the automated translation process is accurate and the framework provides reliable results, regardless of the verification engine used.

Manual Pipeline | Auto Python → SMV System

Automatic Python → SMV Translation

Demo Mode: Using pre-loaded example_program.py

System automatically translates Python → SMV → verifies with NuSMV + Z3

Example Python Program:

```
# Original finite state system for verification
    state = "IDLE"
    ok = True

    def step(event):
        global state, ok
        if state == "IDLE":
            if event == "start":
                state = "BUSY"
            else:
                ok = False  # unexpected event
        elif state == "BUSY":
            if event == "finish":
                state = "DONE"
            elif event == "fail":
                ok = False
        elif state == "DONE":
            pass  # terminal
```

Run Auto Python → SMV Pipeline

Auto Python → SMV Pipeline Results

Pipeline Summary:

Consistency: CONSISTENT

SMV Verdict: INVALID

Z3 Counterexample Found: Yes

Translation:


```
{
  "message": "Python program auto-translated to SMV",
  "python_source": "/input/example_program.py",
  "smv_path": "/input/auto_model.smv"
}
```


NuSMV Verification:

Result: INVALID

Model: auto-generated

▶ **Clean Output**

▶ **Raw Output** (with banner)

Z3 Verification:


```
{
  "counterexample": {
    "event": "\"fail\"",
    "next_ok": "False",
    "state": "\"BUSY\""
  },
  "satisfiable": true
}
```


New Pipeline Run | Download Results

Figure 5.10: Mutual Exclusion Protocol in Manual SMV Verification Result

5.4 Discussion of Results

The VVFramework manual validation process can be summarised by the results of three representative case studies that demonstrate the effectiveness of the tool for manual formal verification.

The first case, the Traffic Light Controller, was used to verify the robustness of the system. Five properties- safety, liveness and timing- were analysed and all successfully validated, confirming the framework's ability to test the correctness of a system.

In the second case, the Login System, the verification focussed on an important security property. The framework proved its worth by detecting an unexpected breach and generating a clear execution trace that served as a detailed counterexample. This result emphasises the tool's ability to not only confirm correctness, but also to identify and diagnose errors.

The third case study, the Mutual Exclusion Protocol, demonstrated the framework's ability to handle concurrent systems. Five key properties- safety, liveness, fairness, turn fairness, and deadlock freedom- were verified and all proved to be valid. This confirms the robustness of the system in handling the complexity associated with concurrency.

Although the VVFramework has achieved its validation goals, it is important to point out its limitations. For instance, in the finally case of studies, the Python to SMV translator is still a proof-of-concept that only supports a limited number of programming constructs. Likewise, the properties for Z3 still have to be coded manually.

The integration of SMV and Z3 shows that the framework can support both automatic and manual verification processes. While a complete translation and verification pipeline (Python → SMV → NuSMV) was implemented in SMV, a manual coding and verification approach (Constraints → Z3) was chosen in Z3.

Taken together, these results show that the framework is able to consistently combine model-checking and theorem-proving approaches and provide a flexible environment for the validation of critical system properties.

5.5 Interpretation of the Validation Results

The validation phase not only confirmed the core functionality of the framework, but also provided important insights into its current scope, performance and internal consistency.

1. Limitations of the VVFramework revealed

The case studies revealed that the VVFramework is currently an effective proof of concept rather than a generalised tool. The main limitation was **the limited scope of the Translation Agent**. As the translator (py2smv.py) was hard-coded for the finite state system (IDLE → BUSY → DONE), the full automated pipeline could only be validated with this one specific scenario.

In addition, the reliance on manual coding for Z3 properties(even in the cross-validation test) showed that **the framework does not yet provide full automation for SMT-based verification**. These limitations illustrate that the main contribution is in the architecture and consistency checking and not in a fully developed, universal Pythoncompiler for formal models.

2. Impact of automation on user effort

The single automated case study has shown that automation **significantly reduces the effort required to initiate formal verification**.

In this particular scenario, the user's task was performed by:

1. Manually writing or converting the system logic into an SMV model.

2. Manually defining the property to be checked (CTLSPEC AG ok).
3. Manual execution of NuSMV and Z3.
4. Manual comparison of the resulting outputs.

Too easy:

1. Deploy the Python base programme (example_program.py).
2. Click on a single "Execute" button.

This reduction represents an important step towards lowering the **entry barrier** for developers who are not familiar with temporal logic or formal modelling languages, thus confirming the main objective of the automated pipeline.

3. Consistency between Z3 and NuSMV

In the single automated case study, the results of the two different verification engines were **completely consistent**:

NuSMV: Returned **INVALID** for the CTLSPEC AG ok property, providing a clear trail.

Z3: Confirmed the violation (i.e. the contract was not fulfilable) and provided a corresponding counterexample.

The fact that both engines independently came to the same conclusion is an important finding, as it confirms the **correctness of the Translation Agent** and the **reliability of the Orchestrator's internal communication**. Crucially, the lack of inconsistencies confirms the integrity of the data flow and modelling semantics within the VVFramework.

Chapter 6

Conclusions and Future work

This thesis addressed the challenge of integrating formal verification techniques into microservices architectures. For this purpose, a hybrid framework called VVFramework Verification and Validation Framework was proposed and implemented. The framework was designed as a microservices-based web system that can combine the manual specification of temporal properties with automatic verification pipelines that translate Python programmes into formal models and send them to verification engines such as NuSMV and Z3.

This thesis makes important contributions to formal software verification and validation. These contributions are in the conceptual design and implementation of VVFramework, a hybrid and modular framework designed to democratise access to formal verification techniques.

One of the main contributions is proposing a microservices architecture for hybrid verification and validation. This approach breaks away from traditional monoliths by defining a series of independent but interconnected system agents. At the heart of the system is the Python Service Agent, which acts as an orchestrator and coordinates the tasks of the other components. This includes the Web Interface Agent, which provides an intuitive interface, specialised agents for verification engines such as the NuSMV and Z3 agents, and a basic translation agent for interoperability.

A translator has been developed to convert a subset of Python programmes into SMV models. This tool is crucial as it allows developers to manually specify properties or automatically generate standard invariants, facilitating the creation of formal models without in-depth knowledge of SMV syntax.

On the other hand, this thesis demonstrates the feasibility of integrating multiple verification engines into a single ecosystem. By encapsulating NuSMV and Z3 as independent services, seamless orchestration was achieved through REST APIs and Docker Compose. This integration not only facilitates the management of the individual tools but also emphasises the complementarity between model checking and theorem proving, allowing a wider range of verification problems to be addressed.

Representative case studies were conducted to validate the framework, including a traffic light control system, a login system, and a mutual exclusion protocol. The experimental results confirmed the ability of VVFramework to successfully verify safety and liveness properties, generate counterexamples when needed, and demonstrate scalability within finite state bounds. This practical validation emphasises the usefulness of the proposed framework.

Finally, this work provides a proof of concept that illustrates how automation, interaction via a web interface, and deployment via microservices can lower the barrier to entry for

formal verification. VVFramework is not just a tool, but a concrete demonstration of how formal verification can become more accessible and applicable in a wider range of software development contexts.

This research began by addressing a critical gap: the lack of accessible frameworks that effectively combine microservices architectures with formal verification. VVFramework successfully addressed this challenge by demonstrating several key principles. We have shown that core principles of microservices - such as loose coupling, scalability and modularity- can be effectively applied to create a flexible and robust verification environment. Furthermore, the framework has proven that an automatic translation pipeline can successfully bridge the gap between general programming languages and formal models, making formal methods more accessible to developers. Finally, the VVFramework has proven that hybrid approaches that combine both manual and automatic verification can significantly improve usability without compromising mathematical rigour, lowering the barrier to entry for formal methods.

Although the VVFramework has achieved its primary goals, it is important to recognise its current limitations, which highlight opportunities for future work. The language support of the Python to SMV translator is currently limited to basic constructs and does not yet handle complex features such as concurrency, recursion or unconstrained data types. In terms of scalability, verification performance degrades for systems with large state spaces, which is a known mirror image of the state explosion problem inherent in model checking. Furthermore, the framework lacks domain-specific integration with cyber-physical simulators such as ROS, which limits its direct applicability to safety-critical domains such as autonomous driving. Finally, the benchmarks used for validation are limited to small case studies and not to industrial-scale systems.

6.1 Comparison with existing approaches

To place the VVFramework in the broader landscape of verification tools and frameworks, it is important to compare it with representative approaches in this area. Table 6.1 summarises the main differences between traditional monolithic verification tools (e.g., NuSMV, Uppaal), research prototypes that integrate simulation and verification (e.g. ROS + NuSMV pipelines), and the proposed VVFramework.

In contrast to classical tools that rely on manual modelling and are provided as standalone binaries, VVFramework uses a microservices-based architecture that offers modularity, reproducibility and scalability. In addition, the framework integrates both **manual verification** (property specification via the web interface) and **automatic verification** (Python-to-SMV translation pipeline), and thus bridges the gap between developer-friendly programming languages and formal models.

As shown in Table 6.1, VVFramework introduces a number of innovations that are not present in other frameworks. Firstly, the *microservices architecture* allows each verification component to be deployed independently in a container, making the system scalable and portable across different environments. Secondly, the *automatic translation pipeline* from Python to SMV reduces the manual modelling effort that is usually a major obstacle when using traditional formal verification tools. Finally, the *hybrid nature* of VVFramework (manual and automatic workflows) provides flexibility for both experts who want to define their own properties and non-experts who can automatically use standard invariants and safety checks.

Table 6.1: Comparison of VVFramework with related approaches

Framework / Tool	Architecture Style	Supported Logics	Automation Level	Scalability / Deployment	Integration with Programming Languages
NuSMV (baseline)	Monolithic	CTL, LTL	Manual property specification	Limited (single host binary)	None (requires manual model creation)
Uppaal	Monolithic	TCTL (timed logics)	Manual model building	Limited (GUI tool, standalone)	None
ROS + NuSMV Pipelines (research prototypes)	Mixed (simulation + verifier)	CTL	Semi-automatic (ROS → SMV)	Moderate (distributed via ROS nodes)	Partial (ROS messages only)
VVFramework (this work)	Microservices (Docker + REST APIs)	CTL, LTL, SMT constraints	Hybrid (manual and automatic pipelines)	High (containerized, scalable, reproducible)	Python-to-SMV translation supported

6.2 Future work and strategic expansion

The validation of the VVframework provides a solid foundation for its core functionality and architecture. To evolve the framework from a proof-of-concept to a robust, production-ready tool, several key avenues for future development are proposed, focussing on improving scalability, extending verification capabilities, and applying the system to complex problems relevant to industry.

1. Scaling the Architecture with Kubernetes

While Docker Compose has proven effective for managing the current microservices architecture and ensuring reproducibility, integrating Kubernetes (K8s) is the next logical step to meet production-level requirements. K8s would make this possible:

- Elastic Scalability: automatic horizontal scaling of verification agents (NuSMV, Z3) as needed, allowing the framework to handle multiple concurrent verification requests.
- High Availability: Ensuring continuous operation by automatically restarting failed containers and distributing workloads across a cluster.
- Advanced Resource Management: Optimise the allocation of computing resources for intensive verification tasks.

This integration would transform the VVFramework into a distributed, robust service that can support both research and industrial deployment.

2. Extending the verification capabilities

The current framework focuses on model checking and solving SMT problems. To expand the range of properties and system types that the VVFramework can handle, two strategic integrations are required:

- Support for UPPAAL: The integration of a timed automata verifier such as UPPAAL would allow the framework to formally analyse temporal properties of real-time systems. This is crucial for the verification of protocols and controllers where specific timing constraints (e.g. delays, deadlines) are essential for correctness.
- Integration of Coq: Supporting a proof assistant such as Coq would allow the framework to handle higher-order logic and provide greater confidence in the correctness of complex algorithms or security properties that are beyond the scope of traditional model checking.

3. Application to CPS Examples

The current validation relied on a limited finite state machine. The crucial future work lies in the application of the VVFramework to complex, real-world Cyber-Physical Systems (CPS), such as:

- Automotive Protocols: application of the framework to verify communication protocols (e.g. CAN bus) or functional safety functions in vehicles.
- Industrial IoT (IIoT) Systems: Use of the automated pipeline to verify state-based logic in Programmable Logic Controllers (PLCs) or distributed sensor networks.

These applications would fully explore the limits of the Translation Agent's current capabilities, drive the necessary evolution towards a more universal Python-to-SMV compiler, and demonstrate the framework's broader impact on industrial quality assurance and reliability.

6.3 Concluding remarks

In this thesis, the VVFramework was presented as a first step towards hybrid microservice-based verification frameworks. Although still a proof-of-concept, the work shows that the integration of multiple verification tools into a modular and user-friendly architecture is feasible and beneficial.

By lowering the barriers to formal verification and validating its effectiveness in small case studies, VVFramework helps to bridge the gap between theoretical verification methods and practical software development workflows. Its design and implementation provide a foundation for future frameworks that target safety-critical applications and ultimately advance the goal of developing more reliable and trustworthy systems.

Bibliography

Ali, Syed Afraz and Muhammad Waleed Zafar (2021). "API GATEWAY ARCHITECTURE EXPLAINED". In: *INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY* 5.1, pp. 506–546.

Alshuqayran, Nuha, Nour Ali, and Roger Evans (Nov. 2016). "A Systematic Mapping Study in Microservice Architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51. doi: 10.1109/SOCA.2016.15.

Bakshi, Kapil (Mar. 2017). "Microservices-based software architecture and approaches". In: *2017 IEEE Aerospace Conference*, pp. 1–8. doi: 10.1109/AERO.2017.7943959.

Boettiger, Carl (Jan. 2015). "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1, pp. 71–79. issn: 0163-5980. doi: 10.1145/2723872.2723882. url: https://doi.org/10.1145/2723872.2723882.

Bolton, Matthew L., Ellen J. Bass, and Radu I. Siminiceanu (May 2013). "Using Formal Verification to Evaluate Human-Automation Interaction: A Review". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 43.3, pp. 488–503. issn: 2168-2232. doi: 10.1109/TSMCA.2012.2210406.

Bozga, Marius et al. (1998). "Kronos: A model-checking tool for real-time systems". In: *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer, pp. 546–550.

Calçado, P (2014). *Building products at soundcloud—Part III: Microservices in scala and finagle. SoundCloud Limited, 2014*.

Cassar, Ian et al. (2017). "A survey of runtime monitoring instrumentation techniques". In: *arXiv preprint arXiv:1708.07229*.

Chen, Zhe et al. (Sept. 2015). "Formal Semantics of Runtime Monitoring, Verification, Enforcement and Control". In: *2015 International Symposium on Theoretical Aspects of Software Engineering*, pp. 63–70. doi: 10.1109/TASE.2015.11.

Choi, Yunja and Mats Heimdahl (Jan. 2002). "Model checking RSML-e requirements". In: pp. 109–118. doi: 10.1109/HASE.2002.1173111.

Cimatti, Alessandro et al. (2000). "NuSMV: a new symbolic model checker". In: *International journal on software tools for technology transfer* 2, pp. 410–425.

Darvas, Dániel et al. (2014). "Formal Verification of Complex Properties on PLC Programs". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 284–299. isbn: 978-3-662-43613-4.

Fakhfakh, Fairouz, Slim Kallel, and Saoussen Cheikhrouhou (Apr. 2021). "Formal Verification of Cloud and Fog Systems:A Review and Research Challenges". In: *JUCS - Journal of Universal Computer Science* 27, pp. 341–363. doi: 10.3897/jucs.66455.

Fan, W. et al. (May 2018). "Method of Maintaining Data Consistency in Microservice Architecture". In: *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 47–50. doi: 10.1109/BDS/HPSC/

IDS18.2018.00023. url: https://doi.ieeecomputersociety.org/10.1109/BDS/HPSC/IDS18.2018.00023.

Ghimire, Devndra (2020). "Comparative study on Python web frameworks: Flask and Django". In.

Grimm, Tomás, Djones Lettnin, and Michael Hübner (2018). "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip". In: *Electronics* 7.6. issn: 2079-9292. doi: 10.3390/electronics7060081. url: https://www.mdpi.com/2079-9292/7/6/81.

Hannousse, Abdelhakim and Salima Yahiouche (2021). "Securing microservices and microservice architectures: A systematic mapping study". In: *Computer Science Review* 41, p. 100415. issn: 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2021.100415. url: https://www.sciencedirect.com/science/article/pii/S1574013721000551.

Jacobsen, Frederik Krogsdal and Jørgen Villadsen (Mar. 2023). "On Exams with the Isabelle Proof Assistant". In: *Electronic Proceedings in Theoretical Computer Science* 375, pp. 63–76. issn: 2075-2180. doi: 10.4204/eptcs.375.6. url: http://dx.doi.org/10.4204/EPTCS.375.6.

Jamshidi, Pooyan et al. (May 2018). "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3, pp. 24–35. issn: 1937-4194. doi: 10.1109/MS.2018.2141039.

Karabey Aksakalli, Işıl et al. (2021). "Deployment and communication patterns in microservice architectures: A systematic literature review". In: *Journal of Systems and Software* 180, p. 111014. issn: 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111014. url: https://www.sciencedirect.com/science/article/pii/S0164121221001114.

Kern, Christoph and Mark R. Greenstreet (Apr. 1999). "Formal verification in hardware design: a survey". In: *ACM Trans. Des. Autom. Electron. Syst.* 4.2, pp. 123–193. issn: 1084-4309. doi: 10.1145/307988.307989. url: https://doi.org/10.1145/307988.307989.

Krichen, Moez (2023). "A Survey on Formal Verification and Validation Techniques for Internet of Things". In: *Applied Sciences* 13.14. issn: 2076-3417. doi: 10.3390/app13148122. url: https://www.mdpi.com/2076-3417/13/14/8122.

Larsen, Kim G, Paul Pettersson, and Wang Yi (1997). "UPPAAL in a nutshell". In: *International journal on software tools for technology transfer* 1, pp. 134–152.

Lewis, James and Martin Fowler (2014). "Microservices: a definition of this new architectural term". In: *MartinFowler. com* 25.14-26, p. 12.

Lotz, Jannik et al. (Mar. 2019). "Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 45–52. doi: 10.1109/ICSA-C.2019.00016.

Moura, Leonardo de and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 337–340. isbn: 978-3-540-78800-3.

Nandi, Giann et al. (2022). "MARS: a toolset for the safe and secure deployment of heterogeneous distributed systems". In: *43rd IEEE Real-Time Systems Symposium, RTSS 2022*.

Nandi, Giann Spilere et al. (Dec. 2020). "Work-In-Progress: a DSL for the safe deployment of Runtime Monitors in Cyber-Physical Systems". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*, pp. 395–398. doi: 10.1109/RTSS49844.2020.00047.

Pike, Lee, Sebastian Niller, and Nis Wegmann (2012). "Runtime Verification for Ultra-Critical Systems". In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 310–324. isbn: 978-3-642-29860-8.

Prasandy, Teguh et al. (Aug. 2020). "Migrating Application from Monolith to Microservices". In: *2020 International Conference on Information Management and Technology (ICIMTech)*, pp. 726–731. doi: 10.1109/ICIMTech50083.2020.9211252.

Al-Rakhami, Mabrook et al. (Aug. 2018). "Cost Efficient Edge Intelligence Framework Using Docker Containers". In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 800–807. doi: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00138.

Richardson, C (2018). "Microservices Patterns. Manning Publications". In.

Romanowicz, Ewa (2010). "Verification of programs with Z3". PhD thesis.

Rossi, Fabiana, Valeria Cardellini, and Francesco Lo Presti (Aug. 2020). "Hierarchical Scaling of Microservices in Kubernetes". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 28–37. doi: 10.1109/ACSOS49614.2020.00023.

Sahay, Rahul and Rahul Sahay (2020). "Service Fabric". In: *Microsoft Azure Architect Technologies Study Companion: Hands-on Preparation and Practice for Exam AZ-300 and AZ-303*, pp. 623–659.

Shadija, Dharmendra, Mo Rezai, and Richard Hill (Sept. 2017). "Towards an understanding of microservices". In: *2017 23rd International Conference on Automation and Computing (ICAC)*, pp. 1–6. doi: 10.23919/IConAC.2017.8082018.

Söylemez, Mehmet, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan (2023). "Microservice reference architecture design: A multi-case study". In: *Software: Practice and Experience*.

Sözer, H. (2015). "Integrated static code analysis and runtime verification". In: *Software: Practice and Experience* 45.10, pp. 1359–1373. doi: https://doi.org/10.1002/spe.2287. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2287. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2287.

Taibi, D, V Lenarduzzi, and Claus Pahl (2018). "Architectural Patterns for Microservices: A Systematic Mapping Study". eng. In: Setúbal: SCITEPRESS. isbn: 9789897582950.

Theorin, Alfred et al. (2017). "An event-driven manufacturing information system architecture for Industry 4.0". In: *International Journal of Production Research* 55.5, pp. 1297–1311. doi: 10.1080/00207543.2016.1201604. eprint: https://doi.org/10.1080/00207543.2016.1201604. url: https://doi.org/10.1080/00207543.2016.1201604.

Todorov, Vassil, Frédéric Boulanger, and Safouan Taha (2018). "Formal verification of automotive embedded software". In: *Proceedings of the 6th Conference on Formal Methods in Software Engineering*. FormaliSE '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 84–87. isbn: 9781450357180. doi: 10.1145/3193992.3194003. url: https://doi.org/10.1145/3193992.3194003.

Tonse, Sudhir (2014). *MicroServices at Netflix-challenges of scale*.

Velepucha, Victor and Pamela Flores (2023). "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges". In: *IEEE Access* 11, pp. 88339–88358. issn: 2169-3536. doi: 10.1109/ACCESS.2023.3305687.

Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, et al. (May 2016). "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 179–182. doi: 10.1109/CCGrid.2016.37.

Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano Merino, et al. (June 2017). "Cost comparison of running web applications in

the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11. doi: 10.1007/s11761-017-0208-y.

Wang, Siao et al. (Oct. 2021). "Microservice Architecture for Embedded Systems". In: *2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*. Vol. 5, pp. 544–549. doi: 10.1109/ITNEC52019.2021.9587154.

Wing, J.M. (Sept. 1990). "A specifier's introduction to formal methods". In: *Computer* 23.9, pp. 8–22. issn: 1558-0814. doi: 10.1109/2.58215.

Zhang, Fang et al. (2021). "Design and implementation of energy management system based on spring boot framework". In: *Information* 12.11, p. 457.

Zhang, Qizhi et al. (2020). "A formal framework for gate-level information leakage using z3". In: *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, pp. 1–6.

A Traffic Light Control with CTL Specifications in Manual SMV

Formal Verification

Manual Pipeline | Auto Python→SMV System

Manual Verification System

Select Verification Tool:

NuSMV (Model Checking)

NuSMV Model Checking

SMV Model:

```
MODULE main
VAR
  state : {RED, GREEN, YELLOW};
  timer : 0..10;

ASSIGN
  init(state) := RED;
  init(timer) := 0;

  next(state) := case
    state = RED & timer >= 3   : GREEN;
    state = GREEN & timer >= 5 : YELLOW;
    state = YELLOW & timer >= 2: RED;
    TRUE : state;
  esac;

  next(timer) := case
    state = RED & timer >= 3   : 0;
    state = GREEN & timer >= 5 : 0;
    state = YELLOW & timer >= 2: 0;
    timer < 10 : timer + 1;
    TRUE : timer;
  esac;
```

CTL Specifications:

```
CTLSPEC AG (state = RED | state = GREEN | state = YELLOW)
CTLSPEC AG (AF state = RED)
CTLSPEC AG (state = RED -> timer <= 3)
CTLSPEC AG (state = GREEN -> timer <= 5)
CTLSPEC AG (state = YELLOW -> timer <= 2)
```

Remove | Remove | Remove | Remove | Remove

Add CTL Spec

Verify with NuSMV

Manual SMV Verification Results

Status: Verified

Details:


```
{
  "tool": "nusmv",
  "execution_time": 1758808139.957,
  "clean_output": "-- specification AG ((state = RED | state = GREEN) | state = YELLOW)  is true\n-- specification AG (AF state = RED)  is true\n-- specification AG (state = RED -> timer <= 3)  is true\n-- specification AG (state = GREEN -> timer <= 5)  is true\n-- specification AG (state = YELLOW -> timer <= 2)  is true",
  "model_path": "/input/manual_model.smv",
  "specifications_count": 5
}
```


New Verification | Download Results

Figure 1: Traffic Light Control with CTL Specifications in Manual SMV Verification Result

B Python Program

```
1  # Example python program for verification
2  state = "IDLE"
3  ok = True
4
5  def step(event):
6      """
7      Finite state machine for verification
8      States: IDLE, BUSY, DONE
9      Events: start, other, finish, fail, any
10     """
11     global state, ok
12
13     if state == "IDLE":
14         if event == "start":
15             state = "BUSY"
16         else:
17             ok = False  # Unexpected event at Idle
18     elif state == "BUSY":
19         if event == "finish":
20             state = "DONE"
21         elif event == "fail":
22             ok = False
23     elif state == "DONE":
24         # terminal; Other events are ignored
25         pass
26
27 # Specification to verify: "You never get to a state with ok = false"
28 # CTL: AG ok
29 # Z3: For every state s and event e, step (s, e) preserves ok = True
```

Listing 1: Python Program Example (example_program.py)

C Python Program

```
from pathlib import Path

def parse_python_program(python_file_path="/input/example_program.py"):
    """
    Parse the example_program.py and extract the state machine
    In a complete implementation, you would use AST to parse any Python
    program
    """
    # For now, we return the Example_program.py system that we already
    know
    return {
        "states": ["IDLE", "BUSY", "DONE"],
        "init": {"state": "IDLE", "ok": True},
        # transitions extracted from Example_program.py
        "trans": {
            ("IDLE", "start"): ("BUSY", "TRUE"),
            ("IDLE", "other"): ("IDLE", "FALSE"),
            ("BUSY", "finish"): ("DONE", "TRUE"),
            ("BUSY", "fail"): ("BUSY", "FALSE"),
            ("DONE", "any"): ("DONE", "TRUE"),
        },
        "events": ["start", "other", "finish", "fail", "any"]
    }

SMV_TEMPLATE = """MODULE main
VAR
  state : {{{states}}};
  ok    : boolean;
  event : {{{events}}};

ASSIGN
  init(state) := {init_state};
  init(ok)    := {init_ok};
  init(event) := {init_event};

  next(event) := {{ {events} }};

  next(state) :=
    case
{cases_state}
      TRUE : state; -- default stutter
    esac;

  next(ok) :=
    case
{cases_ok}
      TRUE : ok; -- default keep ok
    esac;
"""

def gen_cases(system):
    cs_state = []
    cs_ok = []
    for (s, e), (ns, okexpr) in system["trans"].items():
        cond = f"(state = {s} & event = {e})"
        cs_state.append(f"      {cond} : {ns};")
        cs_ok.append(f"      {cond} : {okexpr};")
```

```
57      return "\n".join(cs_state), "\n".join(cs_ok)
58
59  def to_smv(path: str, python_program_path="/input/example_program.py"):
60      """
61      Translate the specified Python program to SMV model
62      """
63      # We stop the Python program to extract the states system
64      system = parse_python_program(python_program_path)
65
66      states = ", ".join(system["states"])
67      events = ", ".join(system["events"])
68      cases_state, cases_ok = gen_cases(system)
69      smv = SMV_TEMPLATE.format(
70          states=states,
71          events=events,
72          init_state=system["init"]["state"],
73          init_ok="TRUE" if system["init"]["ok"] else "FALSE",
74          init_event="start",                # any initial valid value
75          cases_state=cases_state,
76          cases_ok=cases_ok
77      )
78      Path(path).write_text(smv)
79      return path
80
81  if __name__ == "__main__":
82      out = to_smv("model.smv")
83      print(f"SMV model written to {out}")
```

Listing 2: Python Program Example (py2smv.py)

D NuSMV Program

```
MODULE main
VAR
  state : {IDLE, BUSY, DONE};
  ok    : boolean;
  event : {start, other, finish, fail, any};

ASSIGN
  init(state) := IDLE;
  init(ok)    := TRUE;
  init(event) := start;

  next(event) := { start, other, finish, fail, any };

  next(state) :=
    case
      (state = IDLE & event = start) : BUSY;
      (state = IDLE & event = other) : IDLE;
      (state = BUSY & event = finish) : DONE;
      (state = BUSY & event = fail) : BUSY;
      (state = DONE & event = any) : DONE;
      TRUE : state; -- default stutter
    esac;

  next(ok) :=
    case
      (state = IDLE & event = start) : TRUE;
      (state = IDLE & event = other) : FALSE;
      (state = BUSY & event = finish) : TRUE;
      (state = BUSY & event = fail) : FALSE;
      (state = DONE & event = any) : TRUE;
      TRUE : ok; -- default keep ok
    esac;

CTLSPEC AG ok
```

Listing 3: Auto Model NuSMV

E Python Program

```python
from z3 import String, Bool, Solver, sat, StringVal, And, Or, If, Not

def step_contract(state: str, event: str):
    """Function equivalent to the states system for verification with Z3
    """
    # We define symbolic variables
    s = String('state')
    e = String('event')
    ok = Bool('ok')
    next_state = String('next_state')
    next_ok = Bool('next_ok')

    solver = Solver()

    # Valid states
    valid_states = [StringVal("IDLE"), StringVal("BUSY"), StringVal("DONE")]
    valid_events = [StringVal("start"), StringVal("other"), StringVal("finish"), StringVal("fail"), StringVal("any")]

    # Precondition: State and Event must be valid
    solver.add(Or([s == state for state in valid_states]))
    solver.add(Or([e == event for event in valid_events]))

    # Transition logic
    solver.add(
        If(And(s == StringVal("IDLE"), e == StringVal("start")),
           And(next_state == StringVal("BUSY"), next_ok == True),
           If(And(s == StringVal("IDLE"), e != StringVal("start")),
              And(next_state == StringVal("IDLE"), next_ok == False),
              If(And(s == StringVal("BUSY"), e == StringVal("finish")),
                 And(next_state == StringVal("DONE"), next_ok == True),
                 If(And(s == StringVal("BUSY"), e == StringVal("fail")),
                    And(next_state == StringVal("BUSY"), next_ok == False),
                    And(next_state == s, next_ok == True)  # DONE case or default
                 )
              )
           )
        )
    )

    # Postcondition: We want to prove that ok can always be True
    # To find a counterexample, we negate the property
    solver.add(Not(next_ok))

    if solver.check() == sat:
        model = solver.model()
        return {
            "satisfiable": True,
            "counterexample": {
                "state": str(model.eval(s)),
                "event": str(model.eval(e)),
                "next_ok": str(model.eval(next_ok))
            }
        }
    else:
```

```
55            return {"satisfiable": False, "message": "Property holds"}
56
57    def verify_safety_property():
58        """
59        Verifies the safety property: never ok = False
60        """
61        return step_contract("IDLE", "other")  # Case that breaks the property
```

Listing 4: Python Program Example (contracts_example.py)