applied sciences

MDPI

Article

A Survey on Formal Verification and Validation Techniques for Internet of Things

Moez Krichen [1,2]

[1] Faculty of Computer Science and Information Technology, Al-Baha University, Al-Baha 65528, Saudi Arabia; moez.krichen@redcad.org
[2] ReDCAD Laboratory, University of Sfax, Sfax 3038, Tunisia

Abstract: The Internet of Things (IoT) has brought about a new era of connected devices and systems, with applications ranging from healthcare to transportation. However, the reliability and security of these systems are critical concerns that must be addressed to ensure their safe and effective operation. This paper presents a survey of formal verification and validation (FV&V) techniques for IoT systems, with a focus on the challenges and open issues in this field. We provide an overview of formal methods and testing techniques for the IoT and discuss the state explosion problem and techniques to address it. We also examined the use of AI in software testing and describe examples of tools that use AI in this context. Finally, we discuss the challenges and open issues in FV&V for the IoT and present possible future directions for research. This survey paper aimed to provide a comprehensive understanding of the current state of FV&V techniques for IoT systems and to highlight areas for further research and development.

Keywords: Internet of Things; formal verification; validation; testing techniques

Citation: Krichen, M. A Survey on Formal Verification and Validation Techniques for Internet of Things. *Appl. Sci.* **2023**, *13*, 8122. https://doi.org/10.3390/app13148122

Academic Editor: Dimitris Mourtzis

Received: 9 May 2023
Revised: 7 July 2023
Accepted: 10 July 2023
Published: 12 July 2023

1. Introduction

The Internet of Things (IoT) has become an indispensable part of modern life, connecting billions of devices to the Internet and enabling seamless connectivity, automation, and real-time monitoring [1–4]. The IoT has transformed various industries, including healthcare, transportation, smart homes, and industrial automation, resulting in significant improvements in efficiency, productivity, and convenience [5–8]. However, the weaknesses of IoT systems, such as limited resources, heterogeneous devices, and a lack of standardization, make them prone to dangerous faults and attacks [9–12].

These vulnerabilities have resulted in significant losses and damages in the past. For example, in 2016, the Mirai botnet attack exploited the security weaknesses of IoT devices to launch a massive distributed denial-of-service (DDoS) attack, which disrupted the Internet across the globe. The attack caused an estimated USD 323,000 loss per hour for the affected companies, totaling over USD 100 million in damages [13]. Another example is the Stuxnet worm, which targeted industrial control systems and caused physical damage to nuclear centrifuges in Iran. The attack is estimated to have set back Iran's nuclear program by two years and caused over USD 1 billion in damages [14].

To ensure the reliability, security, and safety of IoT systems, it is crucial to apply FV&V techniques. Formal methods use mathematical techniques to model and analyze systems rigorously [15–17]. As illustrated in Figure 1, they allow system designers to specify the behavior and properties of the system using precise mathematical notations, such as logic formulas and state machines. Formal methods can then use automated tools to analyze the system's behavior and properties, such as checking for consistency, completeness, and correctness [18]. They can also identify potential errors, vulnerabilities, and attacks by exploring the system's behavior under different scenarios [19]. Formal methods can help detect design errors early in the development process and ensure that the system meets the specified requirements and standards.

Testing (TST_{NG}) is an essential step in the verification and validation process, and various techniques have been proposed to test IoT systems effectively [20–22]. For example, model-based testing (MBT) techniques generate test cases automatically from a formal model of the system, which can help achieve better test coverage and reduce the TST_{NG} effort [23–25]. Fuzzy TST_{NG} techniques generate random input data to stress-test the system and identify vulnerabilities and edge cases [26,27]. Moreover, hardware-in-the-loop TST_{NG} techniques can test the interaction between the software and the hardware components of IoT systems, which can help detect integration issues and compatibility problems [28–30].

The need for formal method verification arises from the criticality of ensuring the reliability and security of IoT systems, which are often deployed in safety-critical domains such as healthcare, transportation, and industrial control. FV&V techniques provide a systematic approach to verifying the correctness of these systems by mathematically modeling their behavior and rigorously verifying that they meet their specifications. While FV&V techniques can be time-consuming, they can also reduce the overall cost of development by identifying and eliminating potential issues early in the design process. Moreover, the use of automated tools and techniques can help reduce the manual effort required for FV&V, making it more applicable to meet time-to-market constraints.

The application of updates and patches to IoT systems can potentially introduce new vulnerabilities or alter the behavior of the system. To address this, FV&V techniques can be used to verify the correctness of updates and patches before they are applied to the system. This can help ensure that the system remains reliable and secure even after updates and patches are applied. Additionally, FV&V techniques can be used to verify the correctness of the updates and patches themselves, reducing the risk of introducing new vulnerabilities or errors.

Developing a generic methodology to validate all types of IoT devices using FV&V techniques is an area of active research. While FV&V techniques are applicable to a wide range of IoT systems, the specific techniques and tools used may vary depending on the characteristics of the system being verified. Researchers have developed methodologies for specific types of IoT devices, such as sensors, actuators, and network protocols. However, developing a generic methodology that can be applied to all types of IoT devices remains a challenge.

The motivation for using FV&V techniques in IoT systems lies in the criticality of ensuring their reliability and security. While other techniques such as testing and simulation can also be used to verify the correctness of IoT systems, they may not be sufficient to guarantee their reliability and security in all cases. FV&V techniques provide a formal and rigorous approach to verifying the correctness of IoT systems and can identify potential issues that may not be detected through other means. Additionally, FV&V techniques can be used to verify the correctness of the system throughout its development lifecycle, from design to deployment to updates and patches.

The goal of this paper was to provide a comprehensive survey of formal verification (FV), validation, and TST_{NG} techniques for IoT systems. We reviewed the state-of-the-art methods and tools for modeling, specification, verification, and TST_{NG} of IoT systems. We also discuss their strengths and limitations and identified the open challenges and future research directions in this area. By providing a holistic view of the FV, validation, and TST_{NG} landscape for the IoT, this paper aimed to help researchers and practitioners in developing more-secure, -reliable, and -trustworthy IoT systems.

The main contributions of this paper are summarized below:

- Overview of FV&V techniques for IoT systems;
- Discussion of challenges and open issues in FV&V for IoT systems;
- Examination of formal methods and TST_{NG} techniques for IoT systems;
- Exploration of the use of AI in software TST_{NG} for IoT systems;
- Identification of areas for future research and development in FV&V for IoT systems.

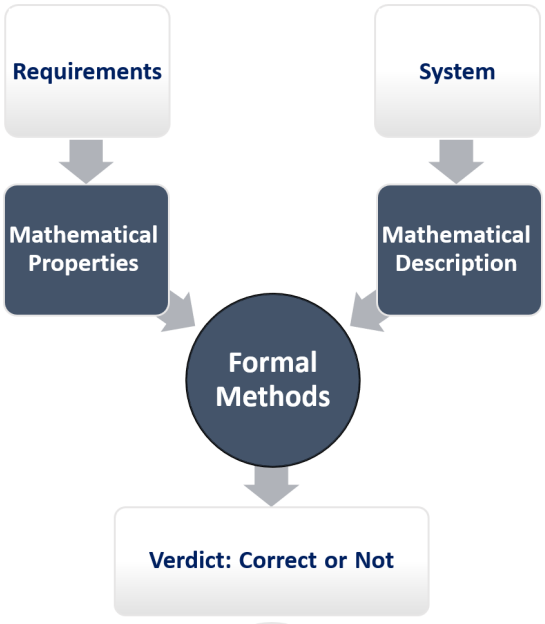

Figure 1. A simplified diagram illustrating how formals methods function.

The rest of the paper is structured as follows. Section 2 is a review of the related work. In Section 3, we discuss the preliminaries related to the IoT, including the definition of the IoT, its characteristics, and the challenges it presents. Section 4 provides an overview of formal methods for the IoT. In Section 5, we explore TST_{NG} techniques for the IoT, including the importance of TST_{NG}, different types of TST_{NG}, and the challenges of TST_{NG} in the IoT. Section 6 examines the use of AI in software TST_{NG} for IoT systems, including its advantages and examples of tools. Section 7 presents a case study. In Section 8, we discuss the challenges and open issues in FV&V for IoT systems and present possible future directions for research. Finally, we conclude the paper in Section 9, summarizing the main contributions and their significance and highlighting areas for future work.

2. Related Work

In this section, we review several formal verification approaches proposed for IoT protocols in recent years, including the IoT Conflict Checker, trust-based service management, BiAgents*, and formal models of popular messaging protocols. We also discuss formal approaches to defining semantics for modeling IoT applications and ensuring physical layer security. Finally, we examine VerificationTalk, a mechanism for verifying device and network configurations to prevent incorrect deployment of IoT applications.

The article [31] reviewed formal verification approaches for many IoT protocols. Formal verification is essential for early vulnerability detection, according to the report. The authors detailed the properties and approaches. An in-depth literature study identified four application fields: (1) functional checks, (2) security property checks, (3) ideas for better schemes including a priori security property checks, and (4) protocol implementation checks. The paper covered security properties and protocol tools. The authors also described typical model checks and discuss IoT difficulties and limitations.

The fundamental contribution of [32] is the invention of a formal method approach, the IoT Conflict Checker (IoTC2), for ensuring the safety of controller and actuator behavior in the Internet of Things (IoT) in the face of conflicts. The work specified and implemented safety policies for controllers, actions, and triggering events in Prolog to demonstrate logical completeness and soundness. The Matlab Simulink Environment with its built-in Model Verification blocks was used to implement the detection policies. The authors used Simulink to develop a smart home environment and demonstrated how conflicts affect actions and the corresponding features. The method's scalability, efficiency, and accuracy

were evaluated in a simulated environment, demonstrating its potential utility in real-world IoT applications.

The work [33] represented and verified trust-based service management components in the Internet of Things (IoT) using a formal method based on higher-order logic (HOL). The study examined IoT service composition difficulties and trust-based alternatives, which outperform non-trust-based ones. The authors offered a trust and reputation system to identify appropriate service providers (SPs) for service composition plans and utilized HOL to validate the varied behaviors in the trust system and trust value computation procedures. Malicious nodes can skew trust value computation, leading in incorrect SP selection during service composition.

The paper [34] proposed a Bigraphical Agents (BiAgents*) model to formalize the structure and behavior of Internet of Things (IoT) systems. While there has been much research on IoT networking and devices, formalizing and evaluating IoT systems is still in its infancy, according to the article. The BiAgents* specification is encoded in the Maude language to allow IoT systems to behave autonomously. An intelligent collision-avoidance system illustrated and evaluated the proposed approach. It formalizes and standardizes the intricate interactions between smart devices in the IoT ecosystem, making the article important.

The primary result of the work [35] was the introduction of a timed message-passing process algebra-based formal model of the MQ Telemetry Transport (MQTT) Version 3.1 protocol. Modeling decisions were made, and inconsistencies in the original protocol specification were highlighted in this study. After performing a static analysis on the formal protocol model, the authors found that the protocol provided, at most, once and, at least, once delivery semantics to subscribers, as defined, for the first two QoS modes of operation. The authors, however, concluded that the third and highest QoS semantics were inaccurate in some respects and, at best, unclear in others. Finally, the authors proposed improving the protocol's QoS to that point. This work contributes significantly since it created a formal model of a popular messaging protocol and pinpointed ways in which its specification could be enhanced.

The paper [36] proposed an MDE-based formal approach to define ThingML's formal semantics for modeling Internet of Things (IoT) applications. The variability of IoT components and communication protocols makes designing and developing them difficult. ThingML, a promising UML profile for these difficulties, lacks strict semantics, making it inappropriate for formal verification and analysis of system architectures [37]. Rewriting logic and Maude are used to build ThingML's formal semantics, implementing all concepts and behaviors. A program developed by the authors automatically converts ThingML specifications into Maude, enabling advanced analytical methods such as simulation and model testing. This work proposed a semantics mapping between ThingML ideas and Maude constructs, defined and implemented an operational semantics for the ThingML action language in Maude, and provided a case study.

An Event-B proof-based formal model of Internet of Things (IoT) physical layer security and threats from requirements analysis to the goal level is the main contribution of the study [38]. The article underlined the need for security in IoT devices, especially those that generate, gather, or process sensitive data, and that preventing vulnerabilities is better than identifying them. The authors offered a three-step formal approach: building the IoT physical layer, checking for security weaknesses, and detecting physical layer assaults such as jamming and MAC spoofing. To demonstrate generalizability, an electrocardiogram (ECG) IoT system and a fire alarm system were used as case studies. The Rodin model-checking tool's proof responsibilities and ProB animator validated the authors' approach. This work's formal approach for IoT physical layer security and attack detection can assist in preventing security breaches and securing sensitive data.

The proposed framework in [39] uses the Event-B formal technique to develop a safe Internet of Things (IoT) architecture and handle IoT security and privacy issues. The report noted that, while various novel IoT designs have been presented, they still

face security and privacy issues, and formal verification can assist identify flaws early on. The authors used Event-B properties such as formal verification, functional checks, and model checkers to design formal spoofing attacks for the IoT environment and obtain IoT architecture accuracy by running simulations, proof obligation, and invariant checking. Formal verification, functional checks, and model checkers were used to validate IoT-EAA architecture models, which automatically fulfilled 82.35% of the proof responsibilities using Event-B provers. Finally, the study recommended a well-defined IoT security infrastructure to decrease IoT security issues. This study is important because it gives a formal way for developing a safe IoT architecture and resolving IoT security and privacy issues.

The article [40] reviewed many Internet of Things (IoT) security challenges and proposed formal verification as a promising approach to detect flaws and guarantee security. Due to battery and processing limitations, IoT devices make online cyberattack detection difficult. The authors suggested pre-deploying the device with stricter security tests to reduce the attack surface. The study covered IoT security challenges such functional soundness, code errors, side-channel analysis, and hardware Trojans. State-of-the-art procedures use formal verification tools to detect vulnerabilities before device deployment. This paper is important since it reviews IoT device security vulnerabilities and presents formal verification as a solution.

The VerificationTalk mechanism, which verifies device and network configurations to prevent incorrect deployment of Internet of Things (IoT) applications, is the key contribution of [41]. The study showed that, in a two-domain context, application developers may implement the inappropriate network functions or connect IoT devices that should never be linked, resulting in network function activities that are incorrect. BigraphTalk verifies IoT device configuration, and AFLtalk evaluates network functions. The authors suggested online anomaly detection utilizing a runtime monitor and offline using the American Fuzzy Lop (AFL). The runtime monitor can intercept potentially hazardous messages targeting IoT devices, and VerificationTalk offers feedback for debugging issues. By finding network application security vulnerabilities, VerificationTalk helps design secure IoT apps. The authors showed that, by properly developing the IoTtalk execution engine, AFLtalk's testing capacity is three-times that of typical AFL methods. This work presents a technique to avoid inappropriate deployment of IoT applications by validating configurations in both the device and network domains, enabling secure and dependable IoT application development.

These formal verification approaches have the potential to address several challenges faced by IoT applications and assist in the development of secure and dependable IoT systems.

3. Preliminaries Related to IoT

The Internet of Things (IoT) has evolved significantly over the past few decades, from simple machine-to-machine (M2M) communication to a vast network of interconnected devices and systems [42–47]. The evolution of the IoT has been driven by advancements in communication technologies, such as wireless networks, sensors, and cloud computing, as well as the increasing demand for automation and real-time monitoring in various industries [48–52].

One of the main characteristics of the IoT is heterogeneity, where devices from different manufacturers and with varying capabilities and resources must work together seamlessly. This heterogeneity also applies to the data generated by these devices, which can have different formats, structures, and semantics. As a result, IoT systems must use standardized protocols and formats to ensure interoperability and compatibility across devices and networks. Another characteristic of the IoT is scalability, where IoT systems must be able to handle large numbers of devices and data without compromising performance and reliability. This scalability can be achieved through distributed architectures, which can scale horizontally or vertically, depending on the needs of the application. However, scaling IoT systems can also increase their complexity and introduce new challenges related to data management, security, and privacy. IoT systems must also be resilient to

failures and attacks, due to the critical nature of many applications, such as healthcare and industrial automation. This resilience can be achieved through redundancy, fault-tolerance, and disaster-recovery mechanisms, which can ensure the availability and integrity of the data and services provided by IoT systems. However, ensuring resilience can also increase the costs and complexity of IoT systems, requiring specialized skills and tools. Finally, IoT systems must be able to operate in dynamic and unpredictable environments, such as outdoor environments or moving vehicles. This requires IoT systems to be able to adapt to changing conditions, such as changes in the network topology, interference, or mobility. Moreover, IoT systems must be able to handle data in real-time, which requires efficient data-processing and -analysis algorithms.

The architecture of the IoT typically consists of four layers: the perception layer, the network layer, the middleware layer, and the application layer. The perception layer includes the sensors and actuators that collect and manipulate data from the physical world. These sensors can be of different types, such as temperature, pressure, light, and motion sensors, and can be connected to the network using various communication protocols, such as Zigbee, WiFi, and Bluetooth. The network layer includes the communication protocols and networks that enable devices to connect and communicate with each other. This layer can include different types of networks, such as cellular networks, satellite networks, and ad hoc networks, depending on the application requirements. The middleware layer provides the necessary software and services to manage the devices and data, such as data storage, processing, and security. This layer can include various components, such as data brokers, message queues, and data analytics tools, which can facilitate the integration and analysis of data from different sources. Finally, the application layer includes the software and services that utilize the data and devices to provide value-added services, such as smart homes, healthcare monitoring, and industrial automation. This layer can include various types of applications, such as web applications, mobile applications, and embedded applications, depending on the target platform and user requirements.

Despite the numerous benefits of the IoT, there are also several limitations that need to be addressed. One of the main limitations is the lack of standardization, which can hinder the interoperability and compatibility of different IoT systems. The absence of a common language and notation for describing IoT systems can also hinder the adoption and integration of formal methods into the development process. Therefore, there is a need for standardization efforts to establish a common framework for modeling, specifying, verifying, and TST_{NG} IoT systems using formal methods. Another limitation is the security and privacy risks associated with the IoT, due to the large attack surface and the vulnerabilities of some devices and networks. IoT systems can be subject to various types of attacks, such as denial-of-service attacks, data breaches, and malware attacks, which can compromise the confidentiality, integrity, and availability of the data and services provided by IoT systems. Moreover, IoT systems can collect sensitive data, such as personal health information, financial information, and location data, which can be misused if not properly protected.

IoT systems also face challenges related to power consumption, as many devices are battery-powered and need to operate for long periods without recharging. This requires IoT systems to use efficient power management techniques, such as duty cycling, sleep modes, and energy harvesting, that can extend the battery life of the devices and reduce their environmental impact. Finally, the complexity of IoT systems can make it challenging to develop, test, and maintain them, requiring specialized skills and tools. IoT systems can involve multiple layers, devices, protocols, and standards, which can increase the TST_{NG} effort and make it difficult to achieve comprehensive test coverage. Moreover, IoT systems can be subject to changes in requirements, technologies, and regulations, which can require frequent updates and maintenance. Table 1 summarizes the layers, functions, examples, and protocols of the IoT.

Table 1. Layers , functions, examples, and protocols of the IoT.

Layer	Function	Examples	Protocols
Perception	Data collection	Sensors, actuators	Zigbee, WiFi, Bluetooth
Network	Communication	Cellular, satellite, ad hoc	TCP/IP, MQTT, CoAP
Middleware	Data management	Data brokers, message queues	AMQP, MQTT, DDS
Application	Service provision	Smart homes, healthcare monitoring	REST, SOAP, CoAP

4. Formal Methods

In this section, we provide a concise overview of the most-prevalent forms of formal techniques currently available to the research community [15,16] (Figure 2):

- **Abstract interpretation [53]:** Abstract interpretation is a formal method used to analyze and verify the behavior of computer programs. It is a technique that involves approximating the behavior of a program by abstracting away some of its details. The goal of abstract interpretation is to prove that a program satisfies certain properties, such as safety, liveness, or termination. Abstract interpretation works by defining a set of abstract values that represent the possible states of a program. These abstract values are defined in such a way that they over-approximate the set of possible concrete values. This allows abstract interpretation to reason about the behavior of a program without actually executing it. One of the key benefits of abstract interpretation is that it can be used to analyze programs that are too complex to be analyzed using other methods. This is because abstract interpretation can reason about the behavior of a program at a higher level of abstraction, which makes it possible to handle a much larger state space.
- **Semantic static analysis [54]:** Semantic static analysis is a formal method used to analyze the behavior of computer programs by examining their source code. The goal of semantic static analysis is to detect errors and potential problems in a program before it is executed. Semantic static analysis works by analyzing the syntax and structure of a program to infer its meaning. This is done by constructing a mathematical model of the program's behavior, which can then be used to reason about its properties. One of the advantages of semantic static analysis is that it can be applied early in the development process, which can save time and resources. By detecting errors before a program is executed, semantic static analysis can help to ensure that the final product is correct and reliable. However, semantic static analysis can be challenging because it relies on the ability to reason about complex mathematical models of program behavior. This requires specialized knowledge and expertise in formal methods and mathematical analysis. Additionally, the accuracy of semantic static analysis depends on the quality of the model used, which can be difficult to construct for complex programs [55,56].
- **Model checking [57]:** Model checking is a formal method used to verify the correctness of a system by exhaustively exploring its possible behaviors. It works by constructing a model of the system and specifying the desired properties that the system should satisfy. The model checker then systematically explores all possible states of the system to determine whether these properties hold for all possible behaviors. Model checking is particularly useful for verifying complex systems, such as concurrent and distributed systems, where traditional TST_{NG} methods may not be sufficient. It can also be used to verify hardware designs and protocols. One of the main advantages of model checking is that it can provide complete coverage of all possible behaviors, making it a powerful tool for ensuring the correctness of critical systems.

- **Proof Assistants [58]:** Proof assistants are software tools that help users construct and verify mathematical proofs. They provide a formal language for expressing mathematical statements and a set of rules for manipulating these statements to construct proofs. Proof assistants are useful for formalizing mathematical theories and verifying their correctness. They can also be used to verify the correctness of software and hardware designs. One of the main advantages of proof assistants is that they provide a high level of assurance that the proof is correct, since the proof is constructed using formal rules and the software checks the proof for correctness.
- **Deductive verification [59]:** Deductive verification is a formal method used to verify the correctness of software by constructing a formal proof that the software satisfies its specifications. It works by starting with the specifications of the software and then systematically constructing a proof that the implementation of the software satisfies these specifications. Deductive verification is particularly useful for ensuring the correctness of safety-critical systems, such as those used in aviation and medical devices. One of the main advantages of deductive verification is that it can provide a high level of assurance that the software is correct, since the proof is constructed using formal rules and the proof can be checked by a computer.
- **Design by refinement [60,61]:** Design by refinement is a formal method used to develop correct software by iteratively refining an abstract specification of the software until a detailed implementation is obtained. It works by starting with a high-level specification of the software and then refining this specification step-by-step until a detailed implementation is obtained. Design by refinement can help to ensure that the software meets its specifications and is free of errors. It can also help to ensure that the software is maintainable and can be easily modified as requirements change. One of the main advantages of design by refinement is that it provides a systematic approach to software development, which can help ensure that the final product is correct and meets its specifications.
- **Model-based testing (MBT) [62,63]**: MBT is a formal method used to test software by generating test cases from a model of the software. It works by constructing a model of the software and then using this model to automatically generate test cases that exercise different parts of the software. MBT can help to ensure that the software meets its specifications and is free of errors. It can also help to reduce the time and effort needed to test the software. One of the main advantages of MBT is that it provides a systematic approach to TST_{NG} that can help to ensure that the final product is correct and meets its specifications.

Table 2 summarizes these different types of formal methods, along with their advantages. Furthermore, formal methods can be classified into three categories: complete, partial, and asymptotically complete. In this classification, complete methods are guaranteed to provide a definitive answer to a given problem, while partial methods may not provide a definitive answer, but can provide useful information. Asymptotically complete methods are not guaranteed to find a solution, but as the problem size grows, the probability of finding a solution approaches 1. Here are some further details regarding these three classes:

- **Complete methods [64]**: Complete methods are formal methods that are guaranteed to provide a definitive answer to a given problem. This means that, if a problem has a solution, a complete method will find it. For example, model checking is a complete method because it can systematically explore all possible behaviors of a system to determine whether a given property holds or not. Different types of complete FV exist, namely: SMT-based methods [65] and MILP-based methods.
- **Partial methods [66]:** Partial methods are formal methods that may not provide a definitive answer to a given problem. This means that a partial method may not be able to determine whether a problem has a solution or not. For example, abstract interpretation is a partial method because it can provide an over-approximation of

the behavior of a program, but it may not be able to determine whether the program satisfies a given property or not.

- **Asymptotically complete methods:** Asymptotically complete methods are formal methods that are not guaranteed to provide a definitive answer to a problem, but as the size of the problem grows, the probability of finding a solution approaches 1. This means that, for very large problems, an asymptotically complete method will almost always find a solution. For example, heuristic search is an asymptotically complete method because, as the size of the search space grows, the probability of finding a solution approaches 1, even though there is no guarantee that a solution will be found for any given problem instance.

Table 2. Different types of formal methods.

Formal Method	Description	Main Benefits
Abstract Interpretation	Analyzes and verifies the behavior of computer programs by abstracting away some details and approximating the program's behavior	Can analyze complex programs and reason about their behavior at a higher level of abstraction
Semantic Static Analysis	Analyzes the behavior of computer programs by examining their source code and inferring their meaning	Can detect errors early in the development process and ensure the final product is correct and reliable
Model Checking	Verifies the correctness of a system by exhaustively exploring its possible behaviors	Provides complete coverage of all possible behaviors and can ensure the correctness of critical systems
Proof Assistants	Software tools that help users construct and verify mathematical proofs	Provides a high level of assurance that the proof is correct and can verify the correctness of software and hardware designs
Deductive Verification	Verifies the correctness of software by constructing a formal proof that the software satisfies its specifications	Provides a high level of assurance that the software is correct and can ensure the correctness of safety-critical systems
Design by Refinement	Develops correct software by iteratively refining an abstract specification of the software until a detailed implementation is obtained	Provides a systematic approach to software development that can ensure the final product is correct and meets its specifications
MBT	Tests software by generating test cases from a model of the software	Can ensure the software meets its specifications and is free of errors and can reduce the time and effort needed to test the software

Table 3 summarizes these three classes of formal methods, along with their advantages and limitations. Complete methods provide a definitive answer, but may be computationally expensive and may not scale well to large problems. Partial methods can handle complex problems, but may not be able to detect all errors or find all solutions. Asymptotically complete methods can handle very large problems and can often find solutions quickly, but may not always find a solution and may not be able to guarantee correctness. Understanding the strengths and limitations of these different types of formal methods can help developers choose the most-appropriate method for their specific problem.

Table 3. Different classes of formal methods.

Class	Definition	Example	Advantages	Limitations
Complete Methods	Formal methods that are guaranteed to provide a definitive answer to a given problem	Model checking, theorem proving	Provide a definitive answer and can find all solutions	May be computationally expensive and may not scale well to large problems
Partial Methods	Formal methods that may not provide a definitive answer to a given problem	Abstract interpretation, type checking	Can handle complex problems and can provide useful information even if a definitive answer cannot be found	May not be able to detect all errors or find all solutions
Asymptotically Complete Methods	Formal methods that are not guaranteed to provide a definitive answer to a problem, but as the size of the problem grows, the probability of finding a solution approaches 1	Heuristic search, stochastic methods	Can handle very large problems and can often find solutions quickly	May not always find a solution and may not be able to guarantee correctness

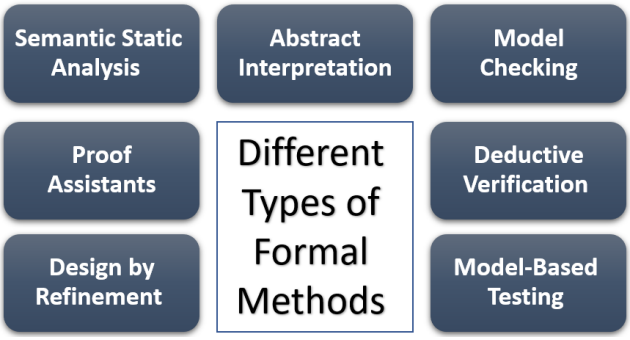

Figure 2. Different types of formal techniques.

The most-important advantages of utilizing formal approaches are as follows:

- **Abstraction:** Formal approaches allow abstraction, which means that they can provide a higher-level view of the software system. This can help to manage complexity by hiding irrelevant details and focusing on the essential characteristics of the system. Abstraction also makes it easier to reason about the behavior of the system and to identify potential errors and defects.
- **Rigorous analysis:** Formal methods provide a rigorous and systematic approach to analyzing software systems. This means that they use well-defined mathematical models and techniques to analyze the software, which can help to ensure that the analysis is accurate and complete. Rigorous analysis can identify defects and errors that may be missed by other methods, such as TST_{NG} or informal reviews.
- **Early defect discovery:** Formal methods can be applied early in the software-development process, which can help to identify defects and errors before they become more difficult and expensive to fix. Early defect discovery can also help to improve the overall quality of the software and reduce the risk of defects, which could lead to system failures or safety hazards.
- **Correctness guarantees:** Formal methods can provide correctness guarantees, which means that they can prove that the software meets its specifications and behaves correctly. This can provide a high level of assurance that the software is correct and

reliable. Correctness guarantees are particularly important for safety-critical systems, where errors or defects could have serious consequences.

- **Reliability:** Formal methods can improve the reliability of software systems by reducing the risk of errors and defects. This can help to ensure that the software behaves as expected and that it is robust and resilient to unexpected inputs or conditions. Reliability is particularly important for systems that need to operate continuously or that cannot be easily repaired or replaced.
- **Efficient test scenarios:** Formal methods can help to identify the most-efficient test scenarios for a software system. This can reduce the time and effort needed to test the software, while still ensuring that the software meets its specifications and behaves correctly. Efficient test scenarios can also help to improve the overall quality of the software and reduce the risk of defects that could lead to system failures or safety hazards [62,63].
- **Maintainability:** Formal methods can improve the maintainability of software systems by providing a clear and precise specification of the system's behavior. This can make it easier to modify or refactor the software without introducing errors or defects. Formal methods can also help to ensure that modifications do not violate the system's specifications or requirements.
- **Reusability:** Formal methods can improve the reusability of software components by providing a clear and precise specification of their behavior. This means that software components can be reused in different contexts without introducing errors or defects. Formal methods can also help to ensure that reused components behave correctly in all contexts.
- **Standardization:** Formal methods can provide a standardized approach to software development and verification. This means that software systems can be developed and verified using a common set of techniques and tools, which can improve interoperability and reduce the risk of errors or compatibility issues.
- **Confidence:** Formal methods can provide developers and stakeholders with confidence in the correctness and reliability of the software system. This can increase trust in the software system and reduce the risk of negative consequences, such as system failures or safety hazards.

Table 4 summarizes the advantages of utilizing formal methods in software engineering, including abstraction, rigorous analysis, early defect discovery, correctness guarantees, reliability, and reusability.

Table 4. Advantages of formal methods in software engineering.

Advantage	Description
Abstraction	Formal methods provide a way to represent complex software systems in a simplified and abstract manner, which can help to reduce the complexity of the system and make it easier to reason about.
Rigorous Analysis	Formal methods provide a way to analyze software systems rigorously and to prove that they meet their specifications. This can help to ensure that the system behaves correctly and that it meets the needs of its stakeholders.
Early Defect Discovery	Formal methods can help detect errors and defects early in the development process, which can make them easier and less expensive to fix.
Correctness Guarantees	Formal methods can provide guarantees that a software system is correct and meets its specifications. This can help to increase the confidence in the system and reduce the risk of errors or defects.

Table 4. *Cont.*

Advantage	Description
Reliability	Formal methods can help to ensure that a software system is reliable and performs as expected under different conditions. This can help to increase the trust in the system and reduce the risk of failures or errors.
Efficient Test Scenarios	Formal methods can help to identify test scenarios that cover all possible system behaviors, which can reduce the number of tests needed and the time required for TST_{NG}. This can result in more-efficient TST_{NG} and faster time-to-market for software systems.
Maintainability	Formal methods can improve the maintainability of software systems by providing a clear and precise specification of the system's behavior. This can make it easier to modify or refactor the software without introducing errors or defects. Formal methods can also help to ensure that modifications do not violate the system's specifications or requirements.
Reusability	Formal methods can improve the reusability of software components by providing a clear and precise specification of their behavior. This means that software components can be reused in different contexts without introducing errors or defects. Formal methods can also help to ensure that reused components behave correctly in all contexts.
Standardization	Formal methods can provide a standardized approach to software development and verification. This means that software systems can be developed and verified using a common set of techniques and tools, which can improve interoperability and reduce the risk of errors or compatibility issues.
Confidence	Formal methods can provide developers and stakeholders with confidence in the correctness and reliability of the software system. This can increase trust in the software system and reduce the risk of negative consequences, such as system failures or safety hazards.

5. Testing Techniques

Testing (TST_{NG}) is an essential part of the software development process, as it helps to ensure that software systems are reliable, performant, and meet the needs of their users. There are many different approaches to TST_{NG} software, each with its own specific objectives and procedures (Figure 3):

- **Unit** TST_{NG} **[67]:** Unit TST_{NG} is a method of TST_{NG} that focuses on individual units or components of a software system. The goal of unit TST_{NG} is to guarantee that each component of the system works as expected and meets its standards. Unit TST_{NG} is normally accomplished by creating and executing test cases for each unit. Unit TST_{NG} has the main advantage of detecting faults and defects early in the development process, making them easier and less expensive to rectify. The biggest disadvantage of unit TST_{NG} is that it may fail to uncover flaws or faults that occur when units are merged.
- **Integration** TST_{NG} **[68]:** Integration TST_{NG} is a TST_{NG} method that focuses on the interactions of several units or components of a software system. The goal of integration TST_{NG} is to guarantee that the system as a whole works as planned and that the units work properly when joined. Integration TST_{NG} is often accomplished by TST_{NG} with various combinations of units and ensuring that they function as expected. The primary benefit of integration TST_{NG} is that it can uncover flaws and defects that occur

when units are merged, making them easier and less expensive to correct. The biggest disadvantage of integration TST_{NG} is that it may miss faults or problems that occur when the system is stressed or loaded.

- **Acceptance** TST_{NG} **[69]:** Acceptance TST_{NG} is a method of determining whether a software system meets its requirements and specifications. The goal of acceptance TST_{NG} is to guarantee that the system is acceptable to its stakeholders and meets their requirements. Acceptance TST_{NG} is often accomplished by putting the system through its paces in a real-world setting and ensuring that it fits the requirements and specifications. Acceptance TST_{NG} has the primary benefit of ensuring that the system meets the needs of its stakeholders. The fundamental shortcoming of acceptance TST_{NG} is that it may fail to uncover mistakes or problems that occur when the system is stressed or loaded.
- **Functional** TST_{NG} **[70]:** Functional TST_{NG} is a TST_{NG} approach that focuses on TST_{NG} of the functionality of a software system. The objective of functional TST_{NG} is to ensure that the system functions correctly and that it meets its requirements and specifications. Functional TST_{NG} is typically achieved by TST_{NG} of the system against a set of predefined test cases that cover all aspects of its functionality. The main advantage of functional TST_{NG} is that it can ensure that the system functions correctly and that it meets its requirements and specifications.
- **Usability** TST_{NG} **[71]:** Usability TST_{NG} is a TST_{NG} approach that focuses on TST_{NG} for how easy it is to use a software system. The objective of usability TST_{NG} is to ensure that the system is usable and that it meets the needs of its users. Usability TST_{NG} is typically achieved by TST_{NG} of the system with a group of representative users and observing how they interact with the system. The main advantage of usability TST_{NG} is that it can ensure that the system is easy to use and that it meets the needs of its users.
- **Stress** TST_{NG} **[72]:** Stress TST_{NG} is a TST_{NG} approach that focuses on TST_{NG} for how well a software system performs under stress or load. The objective of stress TST_{NG} is to ensure that the system can handle high volumes of traffic or requests without crashing or failing. Stress TST_{NG} is typically achieved by TST_{NG} of the system with a high volume of traffic or requests and observing how it performs. The main advantage of stress TST_{NG} is that it can ensure that the system is reliable and can handle high volumes of traffic or requests.
- **Performance** TST_{NG} **[73]:** Performance TST_{NG} is a method of determining how well a software system operates under regular operating conditions. The goal of performance TST_{NG} is to guarantee that the system is responsive and meets the needs of its users. Typically, performance TST_{NG} is accomplished by subjecting the system to a representative load and evaluating how it performs. The primary benefit of performance TST_{NG} is that it ensures that the system is responsive and works properly for its users. The fundamental disadvantage of performance TST_{NG} is that it may miss mistakes or problems that occur when the system is stressed or loaded.
- **Regression** TST_{NG} **[74]:** Regression TST_{NG} is a TST_{NG} method that focuses on determining whether changes to a software system have introduced new mistakes or faults. The goal of regression TST_{NG} is to guarantee that the system continues to work appropriately after modifications have been made to it. Regression TST_{NG} is often accomplished by retesting the system against a set of specified test cases following modifications to it. The primary benefit of regression TST_{NG} is that it ensures that the system continues to function properly after modifications have been made to it. The fundamental shortcoming of regression TST_{NG} is that it may fail to uncover mistakes or problems that occur when the system is under stress or pressure.

Table 5 summarizes the objectives, main procedures, advantages, and limitations of each approach to TST_{NG} software. By understanding the strengths and weaknesses of each approach, developers can choose the most-appropriate approach for their specific needs and ensure that their software systems are reliable and perform as expected.

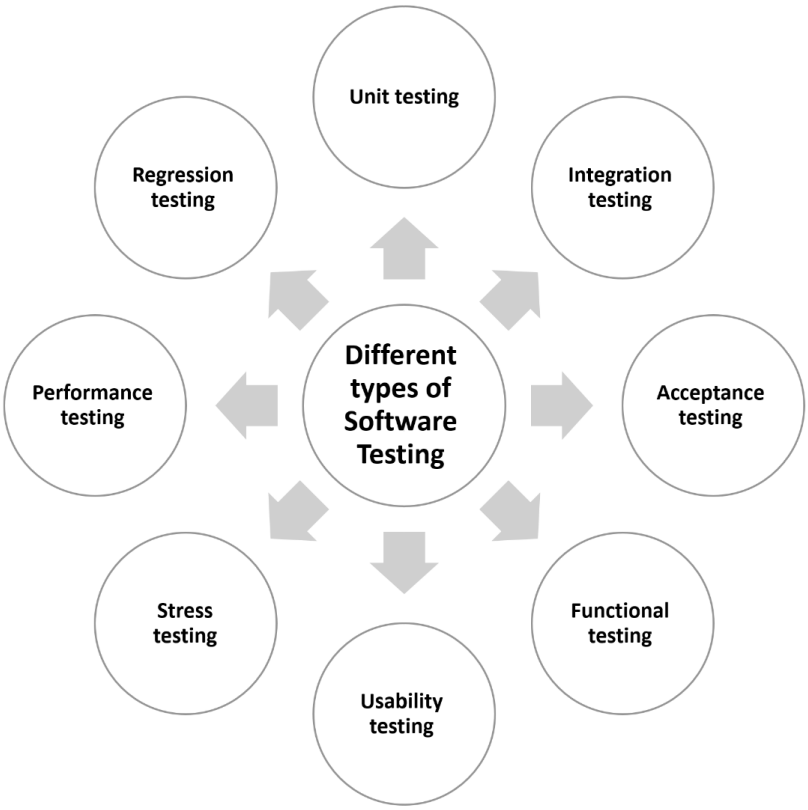

Figure 3. Different kinds of software testing.

Table 5. Approaches to testing software.

Testing Approach	Objective	Main Procedures	Limitations
Unit TST_{NG}	Ensures that each unit of the system is working as expected and meets its specifications.	Writing test cases for each unit and executing those test cases.	May not detect errors or defects that arise when units are combined.
Integration TST_{NG}	Ensures that the system as a whole is working as expected and that the units are functioning correctly when combined.	TST_{NG} of different combinations of units and verifying that they work together as expected.	May not detect errors or defects that arise when the system is under stress or load.
Acceptance TST_{NG}	Ensures that the system meets its requirements and specifications and is acceptable to stakeholders.	TST_{NG} of the system in a real-world environment and verifying that it meets the requirements and specifications.	May not detect errors or defects that arise when the system is under stress or load.
Functional TST_{NG}	Ensures that the system functions correctly and meets its requirements and specifications.	TST_{NG} of the system against a set of predefined test cases that cover all aspects of its functionality.	May not detect errors or defects that arise when the system is under stress or load.
Usability TST_{NG}	Ensures that the system is easy to use and meets the needs of its users.	TST_{NG} of the system with a group of representative users and observing how they interact with the system.	May not detect errors or defects that arise when the system is under stress or load.

Table 5. *Cont.*

Testing Approach	Objective	Main Procedures	Limitations
Stress TST_{NG}	Ensures that the system can handle high volumes of traffic or requests without crashing or failing.	TST_{NG} of the system with a high volume of traffic or requests and observing how it performs.	May not detect errors or defects that arise under normal operating conditions.
Performance TST_{NG}	Ensures that the system is responsive and performs well for its users.	TST_{NG} of the system with a representative load and observing how it performs.	May not detect errors or defects that arise when the system is under stress or load.
Regression TST_{NG}	Ensures that the system continues to function correctly after changes have been made to it.	Retesting the system against a set of predefined test cases after changes have been made to it.	May not detect errors or defects that arise when the system is under stress or load.

6. Use of AI in Software Testing

The use of artificial intelligence (AI) [75–78] in software TST_{NG} has been gaining popularity in recent years [79,80]. AI can help automate various tasks involved in software TST_{NG}, such as test case generation, test execution, and result analysis. One of the main advantages of using AI in software TST_{NG} is the ability to improve test coverage and quality, as AI can analyze large amounts of data and identify patterns and anomalies that may not be apparent to human testers [79]. This can lead to better detection of defects and vulnerabilities, reducing the risk of software failures and downtime.

Another advantage of using AI in software TST_{NG} is the ability to reduce the time and effort required for TST_{NG} [80]. AI can automate repetitive and time-consuming tasks, such as regression TST_{NG}, allowing testers to focus on more-complex and creative tasks, such as exploratory TST_{NG}. This can lead to faster release cycles and reduced time-to-market, which is crucial in today's fast-paced software development industry.

Moreover, AI can also help improve the efficiency and effectiveness of TST_{NG} teams, as it can provide insights and recommendations based on data analysis and machine learning algorithms [79]. This can help testers prioritize their TST_{NG} efforts and focus on the areas that are most critical and likely to have defects. Additionally, AI can also help reduce the cost of TST_{NG}, as it can identify defects and vulnerabilities early in the development cycle, reducing the need for costly rework and maintenance.

6.1. Advantages

The use of AI in software TST_{NG} offers numerous advantages that can help improve the efficiency, effectiveness, and quality of software development. In this section, we will discuss some of the major advantages that can be gained from using AI strategies for software TST_{NG} (Figure 4):

- **Automatic writing of test cases:** Automatic writing of test cases is one of the most-significant advantages of using AI in software TST_{NG}. AI can analyze code and identify potential areas of weakness, allowing it to generate test cases that can thoroughly test the software. This can save significant amounts of time and effort that would otherwise be spent writing test cases manually. Moreover, AI-generated test cases can often cover more scenarios and edge cases than human-written test cases, leading to more thorough TST_{NG} and better software quality.
- **Fast time-to-market:** Ising AI in software TST_{NG} can help reduce the time-to-market for software products. By automating repetitive and time-consuming tasks, such as regression TST_{NG}, AI can help speed up the TST_{NG} process. This can help software companies release products more quickly, gaining an edge in the competitive marketplace. Additionally, a faster time-to-market can lead to increased revenue and

improved customer satisfaction, as customers are more likely to choose products that are released quickly and regularly updated with new features and functionality.

- **Earliest response/feedback:** Another advantage of using AI in software TST_{NG} is the ability to provide early feedback on software quality. AI can detect defects and vulnerabilities early in the development cycle, allowing developers to address them before they become major issues. This can help improve software quality and reliability, leading to better customer satisfaction. Additionally, early detection of issues can help reduce the cost and effort required for fixing them later in the development process.
- **Prognostic analysis:** Prognostic analysis is another advantage of using AI in software TST_{NG}. AI can analyze historical and real-time data to predict the future behavior of a software system. This can help identify potential issues before they occur, allowing developers to take preventative measures to avoid downtime or system failures. Additionally, prognostic analysis can help optimize the performance and efficiency of software systems, leading to better user experiences and improved customer satisfaction.
- **Integrated platform:** Using AI in software TST_{NG} can help integrate various TST_{NG} tools and platforms. AI can help unify different TST_{NG} methods, such as unit TST_{NG}, integration TST_{NG}, and system TST_{NG}. This can help software companies save time and reduce costs by using a single, integrated TST_{NG} platform. Additionally, an integrated TST_{NG} platform can provide a holistic view of software quality, allowing developers to identify and address issues more effectively.
- **Reduction of UI-based TST_{NG}:** AI can help reduce the need for UI-based TST_{NG}, which is often time-consuming and expensive. By automating backend TST_{NG}, AI can help identify issues without the need for extensive UI TST_{NG}, reducing the overall TST_{NG} effort required. Additionally, reducing the need for UI-based TST_{NG} can help improve the efficiency of TST_{NG} teams, allowing them to focus on more-complex and critical TST_{NG} tasks.
- **Better code coverage:** AI can help improve code coverage by identifying areas that are not adequately covered by existing test cases. This can help ensure that all parts of the software are thoroughly tested, reducing the risk of issues and vulnerabilities. Additionally, better code coverage can lead to better software quality and reliability, improving the overall user experience and customer satisfaction.
- **Improved reliability:** By automating TST_{NG} tasks, AI can help improve the reliability of software products. Automated TST_{NG} can detect defects and vulnerabilities that may be missed by manual TST_{NG}, leading to more-reliable and stable software products. Additionally, improved reliability can help reduce the cost and effort required for maintenance and support, improving the overall efficiency and effectiveness of software development teams.
- **Improved quality:** Using AI in software TST_{NG} can help improve the overall quality of software products. By detecting defects and vulnerabilities early in the development cycle, AI can help ensure that software products are of high quality and meet customer expectations. Additionally, improved quality can lead to better customer satisfaction, increased revenue, and a competitive edge in the marketplace.
- **Automated visual validation TST_{NG}:** AI can also be used for automated visual validation TST_{NG}, which involves comparing the visual output of a software system with expected results. This can help identify visual defects and inconsistencies, improving the overall quality and user experience of the software. Additionally, automated visual validation TST_{NG} can help reduce the effort required for manual visual TST_{NG}, allowing TST_{NG} teams to focus on more-complex and critical TST_{NG} tasks.

Overall, the use of AI in software TST_{NG} offers numerous advantages, including faster time-to-market, improved reliability and quality, and reduced TST_{NG} effort and costs. As AI technology continues to evolve, it is likely that more benefits will become apparent, making it an increasingly valuable tool for software development and TST_{NG}.

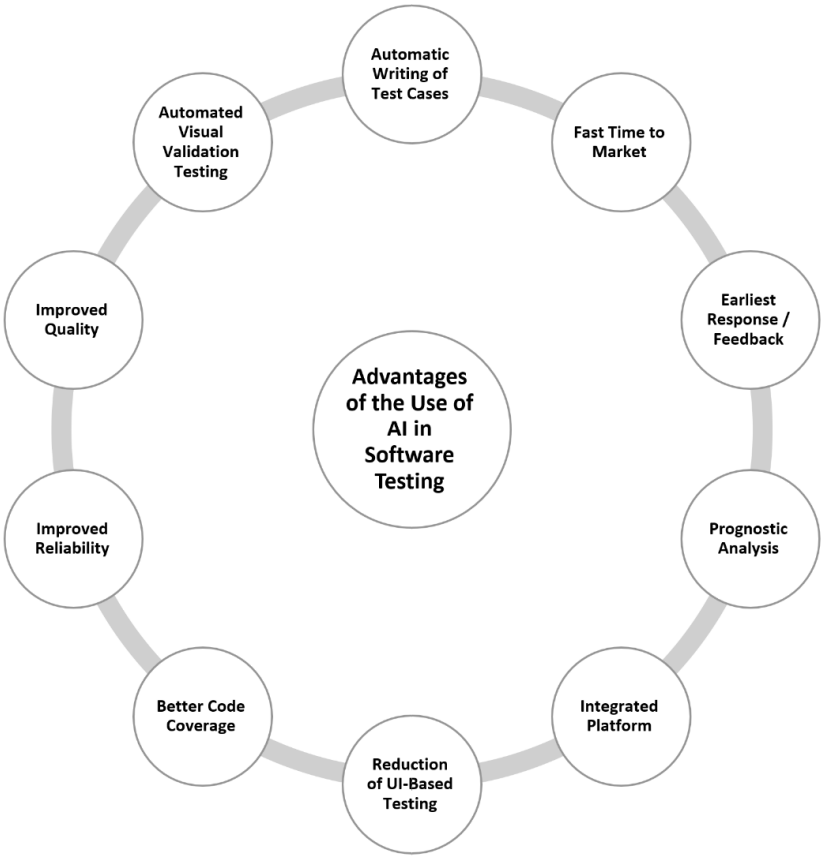

Figure 4. Advantages of the use of AI in software TST_{NG}.

6.2. Examples of Tools

There are many AI-based TST_{NG} tools available in the market that can help improve the efficiency and effectiveness of software TST_{NG}. These tools use AI to automate various TST_{NG} tasks, such as test case generation, test execution, and defect detection. Some examples include the following:

- **Applitools** is a visual TST_{NG} tool that employs AI to automatically detect visual defects and inconsistencies in web and mobile applications. By utilizing computer vision algorithms, Applitools can compare screenshots of an application across various devices, browsers, and resolutions to identify differences that may indicate a defect. The tool can integrate with popular TST_{NG} frameworks, such as Selenium and Appium, to seamlessly incorporate visual TST_{NG} into existing processes. Applitools also provides a dashboard that highlights visual issues and streamlines defect tracking and management. Testers can leverage Applitools to enhance their visual TST_{NG} coverage and accuracy, leading to better software products and increased customer satisfaction.
- **Appvance IQ** is an AI-based TST_{NG} tool that utilizes machine learning algorithms to automatically generate and execute test cases across multiple platforms and environments. The tool can analyze user behavior to generate test cases that cover the most-critical and -common use cases. Appvance IQ can also detect defects and vulnerabilities and provide recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities. Testers can optimize their test coverage and accuracy while saving time and effort on test case creation and maintenance by utilizing Appvance IQ.
- **Functionize** is an AI-based TST_{NG} tool that allows testers to autonomously generate and execute test cases and detect and prioritize defects. Using advanced machine learning algorithms, Functionize can analyze user behavior to generate test cases

that cover critical and common use cases and automatically prioritize defects based on severity. Functionize provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities.

- **Mabl** is an AI-based TST_{NG} tool that enables testers to automatically identify and prioritize issues and generate and maintain test cases. The tool uses advanced machine learning algorithms to analyze user behavior and generate test cases that cover critical and common use cases. Mabl can also detect issues and vulnerabilities and prioritize them based on severity, reducing the effort required for manual defect triage. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities.
- **ReTest** is an artificial-intelligence-based TST_{NG} solution that allows testers to assess software requirements and produce test cases that cover all potential combinations of input parameters. The program analyzes requirements and generates test cases that cover all conceivable combinations of input parameters, ensuring complete test coverage. ReTest can also automatically find problems and vulnerabilities and provide insights and recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities. Testers can increase their TST_{NG} productivity and effectiveness while ensuring complete test coverage and reducing the risk of faults and vulnerabilities.
- **Sauce Labs** is an AI-based TST_{NG} tool that automates TST_{NG} for web and mobile applications. The tool uses advanced machine learning algorithms to automatically generate and execute test cases across multiple platforms and environments, ensuring complete test coverage. Sauce Labs can also detect defects and vulnerabilities and provide recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities. By utilizing Sauce Labs, testers can improve their TST_{NG} efficiency and effectiveness while ensuring complete test coverage across multiple platforms and environments.
- **Test.AI** is an AI-powered TST_{NG} platform that enables testers to create and execute test cases while detecting and prioritizing errors. The tool analyzes user activity and generates test cases that cover crucial and common use scenarios using powerful machine learning methods. Test.AI can also detect and prioritize flaws and vulnerabilities based on severity, minimizing the time and effort necessary for manual defect triage. The tool provides a dashboard that simplifies defect tracking and management and offers extensive results and analytics on TST_{NG} efforts. Testers can increase their TST_{NG} efficiency and effectiveness while reducing the time and effort required for test case generation and maintenance by using Test.AI.
- **Testim** is an AI-driven TST_{NG} tool that enables testers to create and execute test cases with ease. The tool uses advanced machine learning algorithms to analyze user behavior and generate test cases that cover critical and common use cases. Testim can also detect defects and vulnerabilities and provide recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities.
- **Tricentis Tosca** is an AI-based TST_{NG} tool that enables testers to generate, maintain, and execute test cases across multiple platforms and environments. The tool uses advanced machine learning algorithms to analyze user behavior and generate test cases that cover critical and common use cases. Tricentis Tosca can also detect defects and vulnerabilities and provide recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities across multiple platforms and environments.
- **Usetrace** is an AI-powered TST_{NG} tool that enables testers to automatically generate and execute test cases. The tool uses machine learning algorithms to analyze user

behavior and generate test cases that cover critical and common use cases. Usetrace can also detect defects and vulnerabilities and provide recommendations for improving software quality. The tool provides a dashboard that simplifies defect tracking and management and offers detailed reports and analytics on TST_{NG} activities.

These tools demonstrate the diverse ways in which AI can be applied to software TST_{NG}, including visual TST_{NG}, test case generation, defect detection, and automated TST_{NG} across multiple platforms and environments. By using AI-based TST_{NG} tools, developers and testers can improve the efficiency, effectiveness, and quality of software TST_{NG}, ultimately leading to better software products and greater customer satisfaction. Table 6 provides a summary of various AI-based TST_{NG} tools with their functionalities and advantages.

Table 6. Summary of AI-based testing tools.

Tool Name	Functionality	Advantages
Applitools	AI-powered visual TST_{NG}	Automatically detects visual defects and inconsistencies. Compares screenshots across multiple devices, browsers, and resolutions. Integrates with popular TST_{NG} frameworks such as Selenium and Appium. Provides a dashboard for easy defect tracking and management.
Appvance IQ	AI-based test case generation and execution	Automatically generates and executes test cases across multiple platforms and environments. Analyzes user behavior to generate test cases covering the most-critical and -common use cases. Detects defects and vulnerabilities. Provides insights and recommendations for improving software quality. Offers a dashboard for easy defect tracking and management.
Functionize	AI-based test case generation, execution, and defect detection	Autonomously generates and executes test cases. Analyzes user behavior to generate test cases covering the most-critical and common use cases. Automatically detects and prioritizes defects based on severity. Provides a dashboard for easy defect tracking and management. Reduces time and effort required for test case creation and maintenance.
Mabl	AI-based issue identification and prioritization	Automatically identifies and prioritizes issues. Generates and maintains test cases. Automatically detects issues and vulnerabilities. Provides a dashboard for easy defect tracking and management. Reduces the time and effort required for test case creation and maintenance.
ReTest	AI-based test case generation and defect detection	Analyzes software requirements to generate test cases covering all possible combinations of input parameters. Automatically detects defects and vulnerabilities. Provides insights and recommendations for improving software quality. Offers a dashboard for easy defect tracking and management. Ensures complete test coverage.
Sauce Labs	AI-based automated TST_{NG} for web and mobile applications	Provides automated TST_{NG} across multiple platforms and environments. Detects defects and vulnerabilities. Provides insights and recommendations for improving software quality. Offers a dashboard for easy defect tracking and management. Ensures complete test coverage.

Table 6. *Cont.*

Tool Name	Functionality	Advantages
Test.AI	AI-based test case generation and defect detection	Generates and executes test cases. Analyzes user behavior to generate test cases covering the most-critical and -common use cases. Automatically detects and prioritizes defects based on severity. Provides a dashboard for easy defect tracking and management. Reduces time and effort required for test case creation and maintenance.
Testim	AI-based test case generation and maintenance	Generates and maintains test cases. Automatically analyzes user behavior to generate test cases covering the most-critical and -common use cases. Provides a dashboard for easy defect tracking and management. Reduces time and effort required for test case creation and maintenance.
Tricentis Tosca	AI-based end-to-end TST_{NG} for web and mobile applications	Provides automated TST_{NG} across multiple platforms and environments. Detects defects and vulnerabilities. Provides insights and recommendations for improving software quality. Offers a dashboard for easy defect tracking and management. Ensures complete test coverage.

7. Case Study: Temperature-Measuring System

This section presents a short case study to demonstrate the previously introduced methodologies. As shown in Figure 5, our case study has four sensors (*SSR*s) and one collector (*CLLTR*). *SSR*s communicate ambient temperature (*TMPRT*) to the *CLLTR*. The *CLLTR* records the *SSR* values in a database for future use.

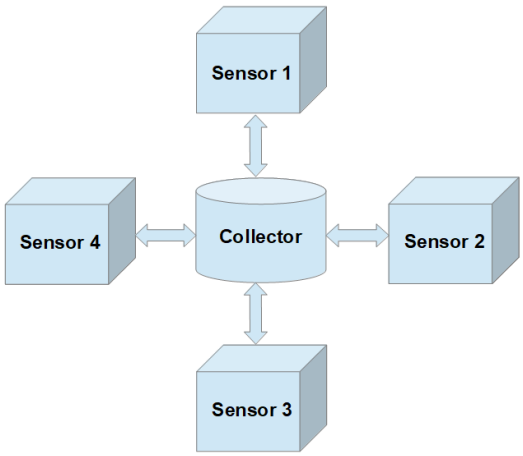

Figure 5. Temperature-measuring system (TMS).

We offer a streamlined model for the suggested case study, composed of eight finite state machines, in Figure 6. On the one hand, each *SSR*'s activity is described as a distinct finite state machine with three nodes and three transitions: *TMPRT* measurements and *TMPRT* transmission to the *CLLTR*, obtaining the *CLLTR*'s acknowledgment. On the other hand, the *CLLTR*'s behavior is provided as a result of four finite state machines. Additionally, each of these finite state machines has three nodes and three transitions: Receiving the *TMPRT* from the associated *SSR*, storing the *TMPRT*, and sending the *SSR* an acknowledgment are the first three steps.

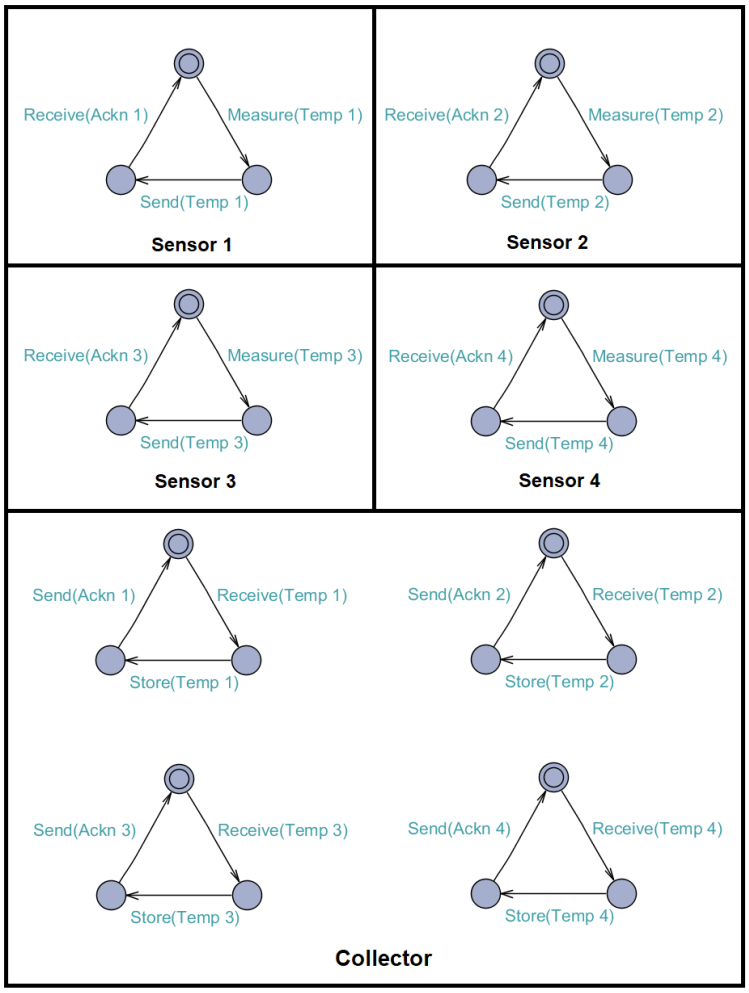

Figure 6. A simplified model for the TMS.

Next, we describe how a number of optimization techniques improved the FV and formal-based TST_{NG} for this case study. We begin with the FV aspects:

- Abstraction: At this level, it is quite simple to see that the various components of the system under consideration are described at a very high level. Additionally, many details are abstracted away, and the interactions between the various SSRs are not taken into account either.
- Modularization and compositionality: The suggested system is represented as a network of eight finite state machines. Every finite state machine has three states and three transitions. By multiplying the products of these different finite state machines, we obtain a large finite state machine with around $3^8 = 6501$ states. If we expand the number of SSRs to eight, the product's states might reach $3^{16} = 43,046,721$, which is a big quantity. This demonstrates the significance of studying modularization and compositionality in order to reduce the size of the models under consideration.
- Symmetry detection: It is easy to observe how the various SSRs and $CLLTR$ pieces play symmetric roles. As a result, the FV of the entire studied system may be simplified to the verification of the product of only two finite state machines: one for each of the four SSRs and one for each of the four collecting elements (as shown in Figure 7).

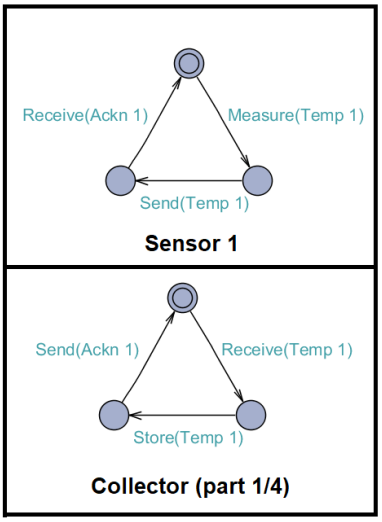

Figure 7. A simplification of the model considering symmetry aspects.

- Data independence detection: We can suppose that the system's various SSRs will measure additional elements such as pressure and humidity. However, if there is no association between $TMPRT$ and these additional variables, there is no need to include them in the system model. This definitely provides for a reduction in the complexity and size of the considered model.
- Eliminating functional dependencies: Assume that the $CLLTR$ saves the average of the readings received from the various SSRs. In this situation, it is evident that there is a direct relationship between the stored variable and the data measured by the SSRs. Thus, in order to simplify the complexity of the verification process, we must account for the connection between these various variables by removing the new variable corresponding to the average value because it can be determined from other variables.
- Exploiting reversible rules: Consider the finite state machine that describes the behavior of SSR 1. This finite state machine can be simplified further by merging the two nodes connected by the transition $Measure(Temp\ 1)$, as this operation can be viewed as internal and has no effect on the FV of the entire system. Similarly, we can compress the pairs of $CLLTR$ finite state machine nodes connected by $Store(Temp\ i)$ transitions. After verification, the collapsed nodes can be separated as they were originally.

Second, we discuss MBT:

- Refinement techniques: This technique considers an untimed specification ($Spec$) and a set of refinement rules that allow each high-level untimed activity to be transformed into a sequence of low-level timed actions. Test cases ($TSTCS$s) are retrieved from the untimed $Spec$ and refined into timed $TSTCS$s using the refinement procedures that have been established. For example, in Figure 8, the operation $Measure(Temp\ 1)$ is refined into a sequence of four timed actions: $TMPRT$ is measured by recording three successive values and then taking the average of these three values. This dramatically simplifies the test-creation technique and greatly decreases the calculation time and space requirements.

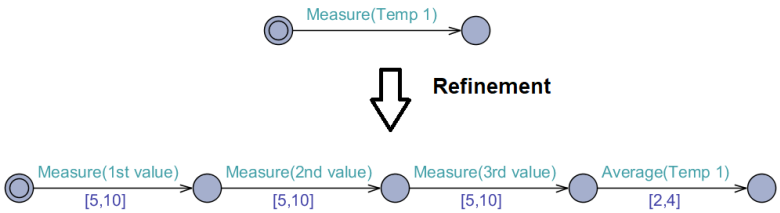

Figure 8. A refinement of a high-level action.

- Reducing the size of digital-clock tests: A digital-clock test can be thought of as a particular tree with a special *Tick* action, which mimics time progression. The purpose of this phase is to reduce the size of the test tree by compacting *Tick* action sequences. This technique is presented in Figure 9, where a sequence of ten *Tick* actions is replaced with just one transition labeled with 10 *Ticks*. The size of the exams is greatly decreased in this way.

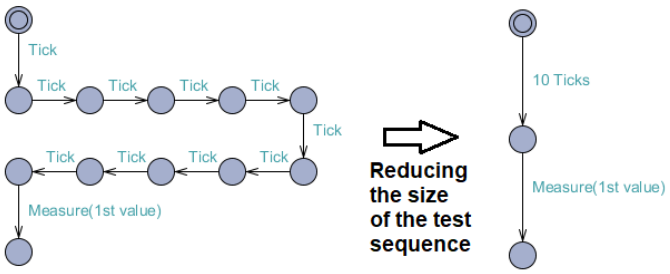

Figure 9. Reduction of the size of digital-clock tests.

- Timed automaton (TA) tester generation: When the system *Spec* is presented as a non-deterministic timed automaton, there are two alternatives for test generation. The first option is to generate *TSTCS* on the fly. That is, test generation and test execution are carried out concurrently. This first option is challenging in general since it necessitates the use of high-performance calculators. The second option is to pre-determine the considered timed automaton before running the test. In Figure 10, for example, we suggest a non-deterministic version of a section of the model of the investigated system. This automaton is non-deterministic since the same action leads to different successor nodes from the beginning node. This non-determinism corresponds to the fact that the *SSR* may estimate *TMPRT* by computing the average of three collected data or only two values depending on some internal choices (for example, available resources). The result of the determinization of the non-deterministic automaton is shown in the same image. As previously stated, this method is not always practicable in a precise manner. As a result, we may need to make some assumptions.

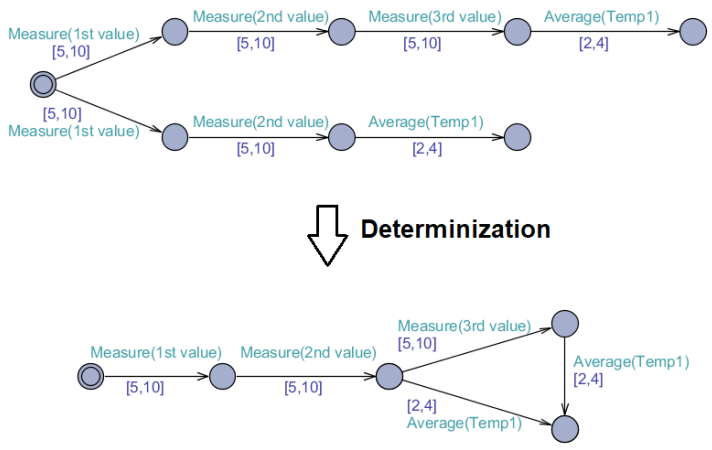

Figure 10. TA tester generation using determinization techniques.

- *TSTCS* updating: As our system evolves, we must update the model accordingly. In this instance, we must also update the previously generated *TSTCS*, as recreating them from the beginning would be prohibitively expensive. In Figure 11, we propose a potential system evolution. The *CLLTR* initially sent an acknowledgment to the corresponding *SSR* after storing the *TMPRT* in the database. In the new version, the acknowledgment is transmitted once the storage is complete. In order to minimize the cost and duration of the test-regeneration phase, the previously generated tests

must be updated appropriately. As previously described, we must compare the two system models (the old and the new) and classify the available $TSTCS$ to achieve this objective.

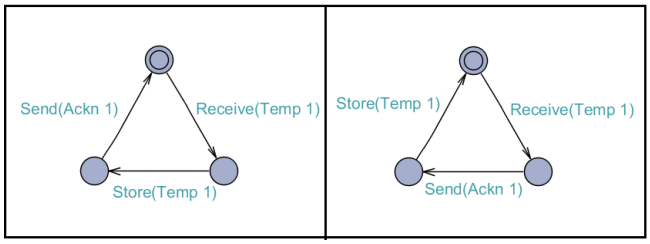

Figure 11. An example of a possible update of the model of the considered system.

- Resource-aware tester component placement: In general, the architecture for TST_{NG} may be centralized or decentralized. In the second scenario, we must devise a method for distributing the various test components across the system's computational elements. If the $CLLTR$ has sufficient resources, for instance, it can host some of the test components devoted to verifying some of the SSRs. Similarly, if one of the SSRs has sufficient resources, it can host the test component responsible for the TST_{NG} of another SSR. Optimization techniques must be employed for this purpose.
- Coverage techniques: These techniques enable the intelligent reduction of the number of created tests by defining specific selection criteria. In our case, we may consider a criterion that enables us to cover the various nodes of the various finite state machines of the model under consideration. Similarly, we may consider a second criterion that encompasses the set of transitions, the set of transition pairs that occur consecutively, etc. This can be accomplished by constructing the observable graph as previously described.

8. Challenges and Open Issues

The Internet of Things (IoT) presents a unique set of challenges for FV&V techniques. One of the main challenges is the complexity and heterogeneity of IoT systems. IoT systems can involve numerous devices and networks, each with different hardware, software, and communication protocols. This makes it difficult to develop a unified FV framework that can be applied to all devices. Moreover, the lack of standardization of IoT devices and networks makes it difficult to develop formal models that accurately capture the behavior of these systems. For example, different devices may have different communication protocols or may use different data formats, making it challenging to develop a unified formal model that can be applied to all devices. Another challenge is the dynamic nature of IoT systems. Devices may join or leave the system at any time, and the behavior of the system may change depending on the context and environment. This makes it difficult to develop a static formal model that can accurately capture the behavior of the system. To address these challenges, researchers have developed device-specific formal models and verification techniques that can be tailored to the specific characteristics of each device. Moreover, they have developed standardized interfaces and protocols that enable interoperability between devices and networks, improving the accuracy and reliability of formal models.

Another challenge in FV&V techniques for the IoT is the state explosion problem. IoT systems can involve a large number of devices and states, making it difficult to analyze them exhaustively. This problem can be addressed using abstraction, modularization, and symmetry-detection techniques. Abstraction involves simplifying the system model by removing extraneous features, while modularization involves breaking down the verification of complex systems into smaller subproblems. Symmetry detection involves minimizing the state space by identifying symmetries that occur during system execution and creating a mapping from states to equivalence class representatives. These techniques have been used in previous research to address the state explosion problem in the FV&V

of IoT systems. However, there is still a need to develop more-efficient and -effective techniques that can handle the dynamic and heterogeneous nature of IoT systems.

Another challenge in the FV&V of IoT systems is the need to ensure that they meet performance requirements while also maintaining security and reliability. Many IoT systems are used in safety-critical applications, such as healthcare and transportation, where reliability and security are of paramount importance. Moreover, IoT systems often involve real-time constraints, which can make it difficult to ensure that they meet the performance requirements. This can be addressed using TAs and other formal models that capture the temporal behavior of IoT systems. However, there is still a need to develop more-sophisticated models and techniques that can handle the complex interactions and dependencies that exist in IoT systems. Moreover, there is a need to ensure that FV&V techniques are integrated into the software development process for IoT systems, rather than being treated as an afterthought. This requires a cultural shift towards a more-formalized approach to software development, as well as the development of tools and frameworks that make it easier to apply FV&V techniques.

Thus, FV&V techniques offer a promising approach to ensuring the reliability and security of IoT systems. However, there are several challenges and open issues that need to be addressed to fully realize their potential. These challenges include the complexity and heterogeneity of IoT systems, the dynamic nature of IoT systems, the state explosion problem, and the need to ensure that IoT systems meet the performance requirements while also maintaining security and reliability. Addressing these challenges will require the development of more-efficient and -effective techniques, as well as a cultural shift towards a more formalized approach to software development.

9. Conclusions and Future Work

In conclusion, FV&V techniques have the potential to address the challenges of reliability and security in IoT systems. However, the dynamic and heterogeneous nature of IoT systems presents several challenges for applying these techniques. Researchers have developed various techniques, such as abstraction, modularization, symmetry detection, and TAs, to address these challenges. However, there is still a need to develop more-efficient and -effective techniques that can handle the complex interactions and dependencies that exist in IoT systems. Moreover, there is a need for a cultural shift towards a more-formalized approach to software development, where FV&V techniques are integrated into the development process.

One possible direction for future research is the development of more-sophisticated formal models and verification techniques that can handle the dynamic and heterogeneous nature of IoT systems. For example, researchers could develop models that capture the interactions and dependencies between devices and networks, as well as the context and environment in which the system operates. They could also develop verification techniques that can handle the large number of states and events that occur in IoT systems, while also maintaining performance and scalability. Another possible direction is the development of tools and frameworks that make it easier to apply FV&V techniques in the development process. These tools could automate the process of model generation and verification, reducing the manual effort required and improving the accuracy and reliability of the models.

Moreover, there is a need to develop more-comprehensive standards for IoT devices and networks that can improve the accuracy and reliability of formal models. Standardization can also enable interoperability between devices and networks, improving the scalability and flexibility of IoT systems. Finally, there is a need to investigate the use of machine learning and artificial intelligence techniques in conjunction with FV&V techniques [81–84]. These techniques can help identify patterns and anomalies in IoT systems, improving their reliability and security.

In addition to the future directions discussed above, there are two more directions that warrant further investigation in the context of FV&V for IoT systems. Firstly, the incorpora-

tion of novel AI techniques in dynamic and heterogeneous IoT systems is an area of great interest. Specific AI algorithms, such as federated learning and 0NKP, have shown promise in improving the security of IoT systems. Researchers could explore the integration of these techniques into formal models and verification techniques to detect vulnerabilities and ensure the safety and security of IoT systems.

Secondly, there is a need to provide more details related to post-quantum security [85]. With the potential impact of quantum computing on IoT security, it is crucial to investigate post-quantum security mechanisms and protocols. Researchers could explore the use of formal methods in conjunction with these techniques to ensure the safety and security of IoT systems in the face of emerging threats.

With these future directions in mind, the continued research and development of FV&V techniques for IoT systems will require collaboration between researchers, developers, and industry stakeholders. By addressing the challenges and open issues discussed in this paper, FV&V techniques can help ensure the reliable and secure operation of IoT systems, enabling their full potential to be realized.

In summary, FV&V techniques offer a promising approach to ensuring the reliability and security of IoT systems. Addressing the challenges and open issues discussed in this paper will require a concerted effort from researchers, developers, and industry stakeholders. With continued research and development, FV&V techniques can help realize the full potential of IoT systems in a safe and secure manner.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Laghari, A.A.; Wu, K.; Laghari, R.A.; Ali, M.; Khan, A.A. A review and state of art of Internet of Things (IoT). *Arch. Comput. Methods Eng.* **2021**, *29*, 1395–1413. [CrossRef]
2. Abdalzaher, M.S.; Fouda, M.M.; Elsayed, H.A.; Salim, M.M. Toward Secured IoT-Based Smart Systems Using Machine Learning. *IEEE Access* **2023**, *11*, 20827–20841. [CrossRef]
3. Hassan, F.; Hussain, S.F.; Qaisar, S.M. Fusion of multivariate EEG signals for schizophrenia detection using CNN and machine learning techniques. *Inf. Fusion* **2023**, *92*, 466–478. [CrossRef]
4. Imtiaz, S.I.; Khan, L.A.; Almadhor, A.S.; Abbas, S.; Alsubai, S.; Gregus, M.; Jalil, Z. Efficient Approach for Anomaly Detection in Internet of Things Traffic Using Deep Learning. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, 8266347. [CrossRef]
5. Alamer, M.; Almaiah, M.A. Cybersecurity in Smart City: A systematic mapping study. In Proceedings of the 2021 International Conference on Information Technology (ICIT), Amman, Jordan, 14–15 July 2021; pp. 719–724.
6. Allouch, A.; Cheikhrouhou, O.; Koubâa, A.; Toumi, K.; Khalgui, M.; Nguyen Gia, T. Utm-chain: Blockchain-based secure unmanned traffic management for Internet of drones. *Sensors* **2021**, *21*, 3049. [CrossRef] [PubMed]
7. Abdalzaher, M.S.; Salim, M.M.; Elsayed, H.A.; Fouda, M.M. Machine learning benchmarking for secured IoT smart systems. In Proceedings of the 2022 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS), Bali, Indonesia, 24–26 November 2022; pp. 50–56.
8. Malik, A.; Khan, M.Z.; Qaisar, S.M.; Faisal, M.; Mehmood, G. An Efficient Approach for the Detection and Prevention of Gray-Hole Attacks in VANETs. *IEEE Access* **2023**, *11*, 46691–46706. [CrossRef]
9. Abdalzaher, M.S.; Elsayed, H.A.; Fouda, M.M. Employing remote sensing, data communication networks, ai, and optimization methodologies in seismology. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2022**, *15*, 9417–9438. [CrossRef]
10. Lee, I. Internet of Things (IoT) cybersecurity: Literature review and IoT cyber risk management. *Future Internet* **2020**, *12*, 157. [CrossRef]
11. Koubaa, A.; Allouche, A.; Khalgui, M.; Cheikhrouhou, O. Blockchain-Based Solution for Internet of Drones Security and Privacy. U.S. Patent 11,488,488, 31 March 2022.
12. Javed, A.R.; Shahzad, F.; ur Rehman, S.; Zikria, Y.B.; Razzak, I.; Jalil, Z.; Xu, G. Future smart cities: Requirements, emerging technologies, applications, challenges, and future aspects. *Cities* **2022**, *129*, 103794. [CrossRef]

13. Antonakakis, M.; April, T.; Bailey, M.; Bernhard, M.; Bursztein, E.; Cochran, J.; Durumeric, Z.; Halderman, J.A.; Invernizzi, L.; Kallitsis, M.; et al. Understanding the Mirai Botnet. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; USENIX Association: Vancouver, BC, Canada, 2017; pp. 1093–1110.
14. Bakić, B.; Milić, M.; Antović, I.; Savić, D.; Stojanović, T. 10 years since Stuxnet: What have we learned from this mysterious computer software worm? In Proceedings of the 2021 25th International Conference on Information Technology (IT), Zabljak, Montenegro, 16–20 February 2021; pp. 1–4.
15. Wang, F.; Cao, Z.; Tan, L.; Zong, H. Survey on learning-based formal methods: Taxonomy, applications and possible future directions. *IEEE Access* **2020**, *8*, 108561–108578. [CrossRef]
16. Gleirscher, M.; Marmsoler, D. Formal methods in dependable systems engineering: A survey of professionals from Europe and North America. *Empir. Softw. Eng.* **2020**, *25*, 4473–4546. [CrossRef]
17. Gleirscher, M.; Foster, S.; Woodcock, J. New opportunities for integrated formal methods. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–36. [CrossRef]
18. Hofer-Schmitz, K.; Stojanović, B. Towards formal methods of IoT application layer protocols. In Proceedings of the 2019 12th CMI Conference on Cybersecurity and Privacy (CMI), Copenhagen, Denmark, 28–29 November 2019; pp. 1–6.
19. Souri, A.; Norouzi, M. A state-of-the-art survey on formal verification of the Internet of things applications. *J. Serv. Sci. Res.* **2019**, *11*, 47–67. [CrossRef]
20. Siboni, S.; Sachidananda, V.; Meidan, Y.; Bohadana, M.; Mathov, Y.; Bhairav, S.; Shabtai, A.; Elovici, Y. Security testbed for Internet-of-Things devices. *IEEE Trans. Reliab.* **2019**, *68*, 23–44. [CrossRef]
21. Jeannotte, B.; Tekeoglu, A. Artorias: IoT security testing framework. In Proceedings of the 2019 26th International Conference on Telecommunications (ICT), Hanoi, Vietnam, 8–10 April 2019; pp. 233–237.
22. Matheu-García, S.N.; Hernández-Ramos, J.L.; Skarmeta, A.F.; Baldini, G. Risk-based automated assessment and testing for the cybersecurity certification and labelling of IoT devices. *Comput. Stand. Interfaces* **2019**, *62*, 64–83. [CrossRef]
23. Garousi, V.; Keleş, A.B.; Balaman, Y.; Güler, Z.Ö.; Arcuri, A. Model-based testing in practice: An experience report from the web applications domain. *J. Syst. Softw.* **2021**, *180*, 111032. [CrossRef]
24. Ahmad, T.; Iqbal, J.; Ashraf, A.; Truscan, D.; Porres, I. Model-based testing using UML activity diagrams: A systematic mapping study. *Comput. Sci. Rev.* **2019**, *33*, 98–112. [CrossRef]
25. Krichen, M.; Mechti, S.; Alroobaea, R.; Said, E.; Singh, P.; Khalaf, O.I.; Masud, M. A formal testing model for operating room control system using Internet of things. *Comput. Mater. Contin.* **2021**, *66*, 2997–3011. [CrossRef]
26. Miller, B.P.; Zhang, M.; Heymann, E.R. The relevance of classic fuzz testing: Have we solved this one? *IEEE Trans. Softw. Eng.* **2020**, *48*, 2028–2039. [CrossRef]
27. Fu, Y.; Ren, M.; Ma, F.; Shi, H.; Yang, X.; Jiang, Y.; Li, H.; Shi, X. Evmfuzzer: Detect evm vulnerabilities via fuzz testing. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 1110–1114.
28. Mihalič, F.; Truntič, M.; Hren, A. Hardware-in-the-loop simulations: A historical overview of engineering challenges. *Electronics* **2022**, *11*, 2462. [CrossRef]
29. Kiesbye, J.; Messmann, D.; Preisinger, M.; Reina, G.; Nagy, D.; Schummer, F.; Mostad, M.; Kale, T.; Langer, M. Hardware-in-the-loop and software-in-the-loop testing of the move-ii cubesat. *Aerospace* **2019**, *6*, 130. [CrossRef]
30. Xie, B.; Wang, S.; Wu, X.; Wen, C.; Zhang, S.; Zhao, X. Design and hardware-in-the-loop test of a coupled drive system for electric tractor. *Biosyst. Eng.* **2022**, *216*, 165–185. [CrossRef]
31. Hofer-Schmitz, K.; Stojanović, B. Towards formal verification of IoT protocols: A Review. *Comput. Netw.* **2020**, *174*, 107233. [CrossRef]
32. Al Farooq, A.; Al-Shaer, E.; Moyer, T.; Kant, K. Iotc 2: A formal method approach for detecting conflicts in large scale iot systems. In Proceedings of the 2019 IFIP/IEEE symposium on integrated network and service management (IM), Arlington, VA, USA, 8–12 April 2019; pp. 442–447.
33. Ahmed, A.I.A.; Hamid, S.H.A.; Gani, A.; Abdelaziz, A.; Abaker, M. Formal Analysis of Trust and Reputation for Service Composition in IoT. *Sensors* **2023**, *23*, 3192. [CrossRef]
34. Souad, M.; Faiza, B.; Nabil, H. Formal modeling iot systems on the basis of biagents* and maude. In Proceedings of the 2020 International Conference on Advanced Aspects of Software Engineering (ICAASE), Constantine, Algeria, 28–30 November 2020; pp. 1–7.
35. Aziz, B. A formal model and analysis of an IoT protocol. *Ad Hoc Netw.* **2016**, *36*, 49–57. [CrossRef]
36. Fortas, A.; Kerkouche, E.; Chaoui, A. Formal verification of IoT applications using rewriting logic: An MDE-based approach. *Sci. Comput. Program.* **2022**, *222*, 102859. [CrossRef]
37. Hagar, J.; Wendland, M.F. Defining Software Test Architectures with the UML Testing Profile. In Proceedings of the 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Dublin, Ireland, 16–20 April 2023; pp. 271–280.
38. Toman, Z.H.; Hamel, L.; Toman, S.H.; Graiet, M.; Valadares, D.C.G. Formal verification for security and attacks in IoT physical layer. *J. Reliab. Intell. Environ.* **2023**, 1–19. [CrossRef]
39. Elsayed, E.K.; Diab, L.; Ibrahim, A.A. Formal Verification of an Efficient Architecture to Enhance the Security in IoT. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 134–139. [CrossRef]

40. Keerthi, K.; Roy, I.; Hazra, A.; Rebeiro, C. Formal verification for security in IoT devices. *Secur. Fault Toler. Internet Things* **2019**, 179–200. [CrossRef]
41. Shieh, M.Z.; Lin, Y.B.; Hsu, Y.J. VerificationTalk: A verification and security mechanism for IoT applications. *Sensors* **2021**, *21*, 7449. [CrossRef] [PubMed]
42. Abdalzaher, M.S.; Samy, L.; Muta, O. Non-zero-sum game-based trust model to enhance wireless sensor networks security for IoT applications. *IET Wirel. Sens. Syst.* **2019**, *9*, 218–226. [CrossRef]
43. Cheikhrouhou, O.; Koubâa, A. Blockloc: Secure localization in the Internet of things using blockchain. In Proceedings of the 2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC), Tangier, Morocco, 24–28 June 2019; pp. 629–634.
44. Nasir, M.; Javed, A.R.; Tariq, M.A.; Asim, M.; Baker, T. Feature engineering and deep learning-based intrusion detection framework for securing edge IoT. *J. Supercomput.* **2022**, *78*, 8852–8866. [CrossRef]
45. Ahmad, W.; Rasool, A.; Javed, A.R.; Baker, T.; Jalil, Z. Cyber security in IoT-based cloud computing: A comprehensive survey. *Electronics* **2021**, *11*, 16. [CrossRef]
46. Mihoub, A.; Lefebvre, G. Social intelligence modeling using wearable devices. In Proceedings of the 22nd International Conference on Intelligent User Interfaces, Limassol, Cyprus, 13–16 March 2017; pp. 331–341.
47. Kelati, A.; Dhaou, I.B.; Tenhunen, H. Biosignal monitoring platform using Wearable IoT. In Proceedings of the 22st Conference of Open Innovations Association FRUCT, Jyvaskyla, Finland, 15–18 May 2018; pp. 332–337.
48. Abdalzaher, M.S.; Soliman, M.S.; El-Hady, S.M.; Benslimane, A.; Elwekeil, M. A deep learning model for earthquake parameters observation in IoT system-based earthquake early warning. *IEEE Internet Things J.* **2021**, *9*, 8412–8424. [CrossRef]
49. Maher, A.; Qaisar, S.M.; Salankar, N.; Jiang, F.; Tadeusiewicz, R.; Pławiak, P.; Abd El-Latif, A.A.; Hammad, M. Hybrid EEG-fNIRS brain-computer interface based on the non-linear features extraction and stacking ensemble learning. *Biocybern. Biomed. Eng.* **2023**, *43*, 463–475. [CrossRef]
50. Krichen, M.; Adoni, W.Y.H.; Mihoub, A.; Alzahrani, M.Y.; Nahhal, T. Security challenges for drone communications: Possible threats, attacks and countermeasures. In Proceedings of the 2022 2nd International Conference of Smart Systems and Emerging Technologies (SMARTTECH), Riyadh, Saudi Arabia, 9–11 May 2022; pp. 184–189.
51. Kondoro, A.; Dhaou, I.B.; Tenhunen, H.; Mvungi, N. Real time performance analysis of secure IoT protocols for microgrid communication. *Future Gener. Comput. Syst.* **2021**, *116*, 1–12. [CrossRef]
52. Gupta, M.; Kumar, R.; Shekhar, S.; Sharma, B.; Patel, R.B.; Jain, S.; Dhaou, I.B.; Iwendi, C. Game theory-based authentication framework to secure Internet of vehicles with blockchain. *Sensors* **2022**, *22*, 5119. [CrossRef]
53. Cousot, P. Abstract interpretation based formal methods and future challenges. In *Informatics: 10 Years Back. 10 Years Ahead*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 138–156.
54. Gosain, A.; Sharma, G. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*; Springer: Berlin/Heidelberg, Germany, 2015.
55. Saadatmand, M.; Enoiu, E.P.; Schlingloff, H.; Felderer, M.; Afzal, W. Smartdelta: Automated quality assurance and optimization in incremental industrial software systems development. In Proceedings of the 2022 25th Euromicro Conference on Digital System Design (DSD), Maspalomas, Spain, 31 August–2 September 2022; pp. 754–760.
56. Abbas, M.; Hamayouni, A.; Moghadam, M.H.; Saadatmand, M.; Strandberg, P.E. Making Sense of Failure Logs in an Industrial DevOps Environment. In Proceedings of the International Conference on Information Technology-New Generations, Las Vegas, NV, USA, 24–26 April 2023; pp. 217–226.
57. Müller-Olm, M.; Schmidt, D.; Steffen, B. Model-checking. In Proceedings of the International Static Analysis Symposium, Venice, Italy, 22–24 September 1999; pp. 330–354.
58. Geuvers, H. Proof assistants: History, ideas and future. *Sadhana* **2009**, *34*, 3–25. [CrossRef]
59. Pnueli, A.; Ruah, S.; Zuck, L. Automatic deductive verification with invisible invariants. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Genoa, Italy, 2–6 April 2001; pp. 82–97.
60. Burch, J.R.; Passerone, R.; Sangiovanni-Vincentelli, A.L. Modeling techniques in design-by-refinement methodologies. In *System Specification & Design Languages*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 283–292.
61. Bensalem, S.; Krichen, M.; Majdoub, L.; Robbana, R.; Tripakis, S. A Simplified Approach for Testing Real-Time Systems Based on Action Refinement. In Proceedings of the ISoLA 2007, Workshop on Leveraging Applications of Formal Methods, Verification and Validation, Poitiers, France, 12–14 December 2007; pp. 191–202.
62. Krichen, M. Contributions to Model-Based Testing of Dynamic and Distributed Real-Time Systems. Ph.D. Thesis, École Nationale d'Ingénieurs de Sfax, Sfax, Tunisie, 2018.
63. Krichen, M. A formal framework for conformance testing of distributed real-time systems. In Proceedings of the International Conference on Principles of Distributed Systems, Tozeur, Tunisia, 14–17 December 2010; pp. 139–142.
64. Davis, J.A.; Clark, M.; Cofer, D.; Fifarek, A.; Hinchman, J.; Hoffman, J.; Hulbert, B.; Miller, S.P.; Wagner, L. Study on the barriers to the industrial adoption of formal methods. In Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems, Madrid, Spain, 23–24 September 2013; pp. 63–77.
65. Barrett, C.; Tinelli, C. Satisfiability modulo theories. In *Handbook of Model Checking*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 305–343.

66. Easterbrook, S.; Callahan, J. Formal methods for verification and validation of partial specifications: A case study. *J. Syst. Softw.* **1998**, *40*, 199–210. [CrossRef]
67. Khorikov, V. *Unit Testing Principles, Practices, and Patterns*; Simon and Schuster: New York, NY, USA, 2020.
68. Shashank, S.P.; Chakka, P.; Kumar, D.V. A systematic literature survey of integration testing in component-based software engineering. In Proceedings of the 2010 International Conference on Computer and Communication Technology (ICCCT), Allahabad, India, 17–19 September 2010; pp. 562–568.
69. van Heugten Breurkes, J.; Gilson, F.; Galster, M. Overlap between Automated Unit and Acceptance Testing—A Systematic Literature Review. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022, Gothenburg, Sweden, 13–15 June 2022; pp. 80–89.
70. Tramontana, P.; Amalfitano, D.; Amatucci, N.; Fasolino, A.R. Automated functional testing of mobile applications: A systematic mapping study. *Softw. Qual. J.* **2019**, *27*, 149–201. [CrossRef]
71. Hertzum, M. *Usability Testing: A Practitioner's Guide to Evaluating the User Experience*; Synthesis Lectures on Human-Centered Informatics; Springer: Berlin/Heidelberg, Germany, 2020; Volume 13, pp. 1–105.
72. Maâlej, A.J.; Lahami, M.; Krichen, M.; Jmaïel, M. Distributed and Resource-Aware Load Testing of WS-BPEL Compositions. In Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018), Funchal, Portugal, 21–24 March 2018; pp. 29–38.
73. Ali, A.; Maghawry, H.A.; Badr, N. Performance testing as a service using cloud computing environment: A survey. *J. Softw. Evol. Process.* **2022**, *34*, e2492. [CrossRef]
74. Lahami, M.; Krichen, M. A survey on runtime testing of dynamically adaptable and distributed systems. *Softw. Qual. J.* **2021**, *29*, 555–593. [CrossRef]
75. Holzinger, A.; Saranti, A.; Angerschmid, A.; Retzlaff, C.O.; Gronauer, A.; Pejakovic, V.; Medel-Jimenez, F.; Krexner, T.; Gollob, C.; Stampfer, K. Digital transformation in smart farm and forest operations needs human-centered AI: Challenges and future directions. *Sensors* **2022**, *22*, 3043. [CrossRef] [PubMed]
76. Alyami, H.; Alosaimi, W.; Krichen, M.; Alroobaea, R. Monitoring social distancing using artificial intelligence for fighting COVID-19 virus spread. *Int. J. Open Source Softw. Process. (IJOSSP)* **2021**, *12*, 48–63. [CrossRef]
77. Krichen, M.; Mihoub, A.; Alzahrani, M.Y.; Adoni, W.Y.H.; Nahhal, T. Are Formal Methods Applicable To Machine Learning And Artificial Intelligence? In Proceedings of the 2022 2nd International Conference of Smart Systems and Emerging Technologies (SMARTTECH), Riyadh, Saudi Arabia, 9–11 May 2022; pp. 48–53.
78. Mihoub, A. A deep learning-based framework for human activity recognition in smart homes. *Mob. Inf. Syst.* **2021**, *2021*, 6961343. [CrossRef]
79. Gao, J.; Ramachandran, M. A survey on software testing techniques using artificial intelligence. *J. Big Data* **2018**, *5*, 1–35.
80. Tian, J.; Li, Y.; Zhang, X. A survey on software testing with machine learning. *J. Softw. Evol. Process.* **2019**, *31*, e2176.
81. Hrizi, O.; Gasmi, K.; Ben Ltaifa, I.; Alshammari, H.; Karamti, H.; Krichen, M.; Ben Ammar, L.; Mahmood, M.A. Tuberculosis disease diagnosis based on an optimized machine learning model. *J. Healthc. Eng.* **2022**, *2022*, 8950243. [CrossRef]
82. Aworka, R.; Cedric, L.S.; Adoni, W.Y.H.; Zoueu, J.T.; Mutombo, F.K.; Kimpolo, C.L.M.; Nahhal, T.; Krichen, M. Agricultural decision system based on advanced machine learning models for yield prediction: Case of East African countries. *Smart Agric. Technol.* **2022**, *2*, 100048. [CrossRef]
83. Cedric, L.S.; Adoni, W.Y.H.; Aworka, R.; Zoueu, J.T.; Mutombo, F.K.; Krichen, M.; Kimpolo, C.L.M. Crops yield prediction based on machine learning models: Case of West African countries. *Smart Agric. Technol.* **2022**, *2*, 100049. [CrossRef]
84. Zidi, S.; Mihoub, A.; Qaisar, S.M.; Krichen, M.; Al-Haija, Q.A. Theft detection dataset for benchmarking and machine learning based classification in a smart grid environment. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 13–25. [CrossRef]
85. Teodoraș, D.A.; Popovici, E.C.; Suciu, G.; Sachian, M.A. Quantum technology's role in cyber-security. In Proceedings of the Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI, Constanta, Romania, 25–28 August 2022; Volume 12493, pp. 96–103.