International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 3, Issue 1, 95-104, 2022
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V3I1P110

Original Article

Formal Verification of Autonomous System Software

Sunil Anasuri
Independent Researcher, USA.

Abstract *- Autonomous systems are transforming numerous industries such as the automotive industry, aerospace industry, robotics and defence industries. Such systems are taking on increasingly safety-critical roles, and thus, reliability and correctness are critical components. Formal verification provides mathematically sound methods of establishing or proving the falseness of software relative to a given formal specification or property. In this paper, a thorough analysis of the formal verification of autonomous system software is offered, including the theoretical background, practical applications, tools and instances of application in a number of fields. Model checking, theorem proving, and Runtime verification are highlighted as the main formal methods. In the course of the critical review of the state-of-the-art, we provide evidence of the achievements and shortcomings of formal verification methods, especially under real-time, adaptive, and AI-controlled autonomous systems. This paper talks of incorporating formal verification in the software development process, its contribution to certification of safety, and its relationship with simulation and testing. The findings reveal the sources of contributions of formal methods in error-early detection, better assurances of safety and a greater 개 increase in robustness of the systems. Conclusively, we state the view that the future challenges will be issues of scalability, the verification of the elements of machine learning, and the development of more user-friendly toolchains to increase industry adoption.*

Keywords *- Formal verification, autonomous systems, model checking, theorem proving, runtime verification.*

1. Introduction

Autonomous systems are intelligent agents aimed at doing work without any or little human control. These systems are more commonly implemented in the challenging, varied and usually unpredictable environment, where the systems have to be able to perceive the external environment, interpret and react in real-time. [1-4] These may be autonomous vehicles on the road or in the city, medical robots operating in the theatre room or patient care areas, or Unmanned Aerial Vehicles (UAVs) in the same airspace due to surveillance or delivery actions. The distinction between these systems is that they are able to produce decisions independently based on real-time information, frequently in a state of uncertainty and time. This independence, though, brings on major obstacles in the verification of correctness, safety, and reliability of the underlying software. Even a single failure or undesired behavior can reach disastrous results in safety-critical areas. Consequently, the software that controls self-driving systems will have to be thoroughly tested to ascertain that the program reacts appropriately to every situation, including situations likely to be rare and unforeseen or so-called edge cases. Although known testing and simulation techniques are valuable, they are in most cases not comprehensive: coverage is very limited and only a fraction of all possible system behaviors is captured by the technique. These points highlight the emergence of the significance of formal verification methods, which provide mathematically sound ways of specifying, modelling, and proving the correctness of autonomous system software. Formal methods provide a high-level resource to establish trust in safe autonomous technology by offering comprehensive analysis and logical assurances of their use.

1.1. Need for Formal Verification

Since the advent of autonomous systems, their integration into or role in increasingly important applications such as transportation/healthcare, aerospace/defence, has never been higher the software they use and depend on therefore must be assured to the intended level of the most rigorous possible. The long-established methods of verification, including simulation and testing, though valuable, do not demonstrate efficiency in ensuring the correctness of these systems because they are rather narrow and fail to address all scenarios of execution. This is where formal verification comes in as a necessity.

- **Limitations of Traditional Testing:** The classical testing methods involve the use of particular inputs and test data in the process, which means that only part of the system behavior can be tested. This is only possible in systems with complexity, in systems where the behavior may be non-deterministic, in systems where the behavior may be reactive, in the face of concurrency and timing requirements, and the face of uncertainty in the surrounding environment. Because of this, some important errors might go unnoticed until the deployment time, when their effects can be disastrous.

- **Guarantees through Mathematical Rigor:** Formal verification is an entirely different direction to approach the identified problem since it begins with mathematical models and deductive logic to ensure all possible system behaviors. Model checking, theorem proving, and runtime verification are techniques which enable engineers to demonstrate properties such as safety (nothing bad happens) and liveness (something good eventually happens) and correctness with respect to a formal specification. Edge cases, race conditions and flawed designs may become evident through this exhaustive analysis and may be overlooked during simulation-based validation.

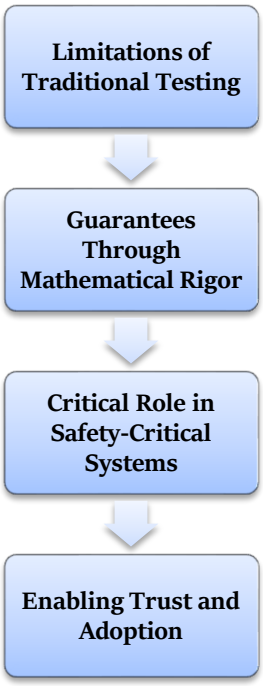

Fig 1: Need for Formal Verification

- **Critical Role in Safety-Critical Systems:** The margin of error is very slim in fields such as autonomous vehicle, UAVs and robotic surgery. Formal verification is becoming more significant (as a mandatory component of safety standards, e.g. ISO 26262 for automotives or DO-178C for avionics) or regulatory conformance (or in some cases, evidence to assess conformance with specifications). Formal methods used in such contexts do not simply mitigate the risk, but also offer auditable, traceable evidence on the correctness of systems.
- **Enabling Trust and Adoption:** Formal verification is an important aspect of ensuring that the general perception by society and regulations is widespread on autonomous systems. The more intelligent and autonomous these systems become, the more users and other stakeholders want to understand their actions and expect them to act in a predictable, but safe way. The demand is addressed in part by the use of formal methods to provide demonstrable assurances of software behaviour, which provides the basis of responsible, large-scale deployment.

1.2. Formal Verification of Autonomous System Software

The formal verification of autonomous system software is a verification that uses mathematically rigorous means to guarantee that the software can operate correctly with respect to every conceivable operating scenario. Traditional testing, because it represents testing specific spots or test cases only, fails to be exhaustive. Still, formal verification is exhaustive since both the system and specifications relating to it can be modeled in a formal language and verified as associated with desired properties. This is significant, in particular, to autonomous systems that are required to stay unattended in potentially dynamic, uncertain settings, and frequently are called upon in real time to make safety-critical decisions. The programming source code of their perception, planning and control systems should be dissimilar; even the slightest mistake in their programming can cause disastrous consequences. Here, formal proof methods are used that include model checking, theorem proving and run-time checking. Model checking refers to an automatic process of checking the properties of safety, liveness, reachability, etc, by determining all the possible states of a system. Theorem proving utilizes deduction and human construction of proof to prove the correctness of a system and provides high confidence in its complicated components.

Runtime verification supplements these methods by passive inspection of system behavior during execution to detect violations of the specified properties in real time. Formal verification of autonomous systems is special. Most of these components (particularly those relating to machine learning or sensor-based perception) are hard to model formally, because they are probabilistic and non-deterministic. However, evidence-making operations, control algorithms, communicational protocols, etc, are key processes which could be well represented through mechanisms like finite state machines, timed automata, or hybrid systems. Verification Tools like UPPAAL, Coq, SPIN, CBMC have been effectively used to test automated actions like collision prevention, planetar assignment, and controller production. Finally, the formal verification contributes to the credibility, safety, and certification capabilities of the software of any autonomous system, so it is an extremely valuable tool in the development cycle of future smart machines.

2. Literature Survey

2.1. Overview of Formal Methods

Formal methods are system specification, development, and verification techniques based on mathematics. Such approaches are correct and reliable because of the strict modeling of system behavior. They can all be classified into four approaches. [5-9] Model Checking Theory Model Checking is an automated discipline concerned with the verification of finite-state systems against temporal logic specification (enabling state space exploration). In Theorem Proving, proofs must be built manually, through logical inference, sometimes aided by an interactive proof assistant, e.g. Coq or Isabelle. Abstract Interpretation is a program analysis that studies the behavior of a program by modelling that behavior with an abstract state over-approximating the set of states a program may reach. Runtime Verification, by contrast, aims to verify system executions during execution to make sure that they obey desired properties, which is especially practical on dynamic and adaptive systems.

2.2. Historical Perspective

Formal methods pursuits go back to the 1980s due to the requirements of correctness in safety-related fields like aerospace and nuclear systems. Nevertheless, it was initially prevented by the inability to scale effectively and the absence of supportive tooling. This became much different in the 2000s when efficient SAT (Boolean-Satisfiability) and SMT (Satisfiability Modulo Theories) solvers were developed, which have made it possible to automate most verification tasks. Such developments made the field exciting, with the result that active interest and industry applications are seen in the fields of embedded systems, security and in the recent past, autonomous systems.

2.3. Applications in Autonomous Systems

Formal methods have become more and more relevant to the field of autonomous systems, to which reliability and safety are paramount concerns. As an example, the problem of timing constraints and safety properties of the autonomous vehicle systems has beenꟷibilized with model checking using UPPAAL, among others. Theorem proving has found applications in justifying the correctness of complicated UAV path planning algorithms, with Coq having particularly through its proposal by Venkat and Cristian in 2008 and by Gruska and Van Dooren in 2012, appeared mostly in this area. Additionally, multiple robots are using runtime verification, as it should be implemented in multi-robot coordination, where there is a need to verify dynamically executed behavior, such as with tools like RV-Monitor. These applications bring out the increased role of formal methods in securing credibility in autonomous technologies.

2.4. Challenges Identified

However, formal methods also have a number of critical challenges despite being strong. The best known is the state explosion problem in model checking, in which the number of states of a system can exponentially increase with the complexity of the system, rendering the activity computationally intensive or proving impossibly complex. Theorem proving is powerful and expressive, but in practice, it usually has a major manual component that is very labor intensive and requires expertise that severely limits its generality and ease of use. In addition to this, formal methods that exist are weaker at supporting probabilistic and learning-based systems that are more pervasive in contemporary autonomous systems, e.g. systems based on AI and machine learning. This disparity poses a critical challenge in testing the next generation of intelligent systems.

3. Methodology

3.1. Formal Verification Workflow

The professionalization of the verification process has the structure of a workflow that provides the idea of the correctness of the system by requiring a verified implementation. [10-13] The stages are cumulative and lead to systematic and rigorous reasoning concerning the behavior of the system.

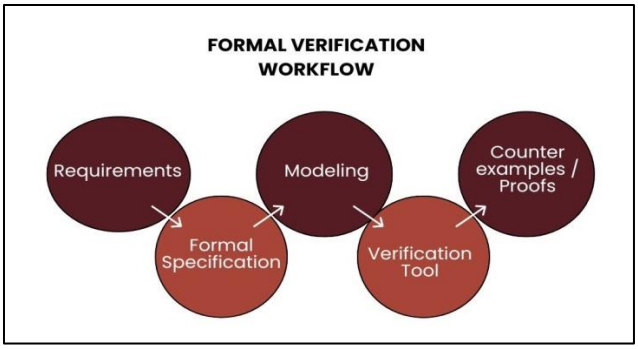

Fig 2: Formal Verification Workflow

- **Requirements:** The starting point of the workflow will therefore be the clear definition of the system requirements, which defines what the system should do. They may be either functional, e.g. safety constraints, timing properties or non-functional, e.g. performance, reliability. They are a basis of further verification. These requirements should be complete, unambiguous and formally expressible in formal verification, because any imprecision in this part can result in incorrect or incomplete results of verification.
- **Formal Specification:** When requirements have been grasped, they can be formalised and articulated as a formal specification, which represents the desired properties of the system over mathematical logic or formal language. The step suggests specification of invariants, safety requirements, liveness or time constraints in a formally correct and machine-readable manner. Official specifications are free of ambiguity and offer a formal basis to compare behavior of the system it is expected to have and not have.
- **Modeling:** During the modeling stage, the system is created using an abstract version. The system structure and behavior are described by this model with formal semantics in terms of state machines, process algebras, or timed automata. The abstraction simplifies the system be simplified so that essential behaviors are not unnecessarily eliminated, making analysis effective. A well-designed model serves as a control between the high-level specifications and the verification tool, which is important for scalability and correctness.
- **Verification Tool:** Then, having the model and specification, an adequate verification tool is used to conduct an automated or semi-automated analysis. Depending upon the approach (e.g., model checking, theorem proving, runtime verification), tools like UPPAAL, Coq, SPIN or CBMC are used. They involve the exploration of the state space of the system or the production of logical proofs that ensure that the specification is true in the model.
- **Counterexamples / Proofs**: The result of the verification is either a correctness proof or a counterexample that shows a specification violation. A proof warrants the fact that the model possesses all the formal properties regardless of scenarios. On the contrary, in case of a failure, if no error is found, the tool gives a counterexample, a trace or action sequence that follows and results in a violation. Such counterexamples are also crucial when debugging, as the developer can get the idea of the actual cause and make adjustments to the system model or specification.

3.2. Modeling Autonomous Behavior

Modeling of autonomous behavior means the abstraction of complicated and sometimes even non-deterministic behavior of autonomous systems by precise mathematical models. This is necessary in undertaking formal verification schemes since a patterned frame of existence is given to the behavioral exploration and rationalization of systems. Behavior may be reactive, i.e. autonomous systems always react to the stimuli of the surrounding environment, as in self-driving cars, drones, and robot swarms. A formal model like a state-transition system, a finite automaton, or a hybrid system typically describes the latter. A state-transition system describes the system as a collection of states and transitions between those states, with events or inputs provoking the transitions. This enables engineers to model the transformation of the system over time in response to different conditions. A more restricted set (appropriate to systems that have a finite number of discretely representable states) is the finite automata, used mostly in protocol verification and control logic design. They feature effective representation of deterministic behavior and can be readily analyzed with an automatic toolset, including model checkers. The problem is that most autonomous systems have to work in conditions of the occurrence of both discrete and continuous changes.

An example is that a drone may be discrete in its decision-making (e.g. take off, land, avoid obstacles) but continuous in its motion through physical space. In these instances, hybrid systems are applied, containing discrete transitions of states together with continuous dynamics, usually in the form of differential equations. Hybrid automata are modest extensions of classical state machines, in that they have time-varying variables. Hybrid automata are very suitable models of embedded controllers, robot

dynamics, and physical interaction. The modeling process relies on a selection of methods to be used based on the complexity, needs and character of the system under analysis. To verify the system, it is very important to create a well-constructed model not only so that the system can be verified accurately, but also to find ambiguities or inconsistencies in the system design. Therefore, the integration of modeling forms the junction between high-level requirements and the rigorous verification process.

3.3. Specification Languages

Specification languages in formal verification. It is in formal verification that imperative systems specify their desired properties and expected behavior of any machine within formal languages. [14-18] These are languages whose syntax and semantics combine mathematical use to outline system goals, constraints, and patterns of interaction. Some of the most popular flavors of specification languages include temporal logics, process algebras and contract-based specifications that characterize various features of a system's behavior and verification approaches.

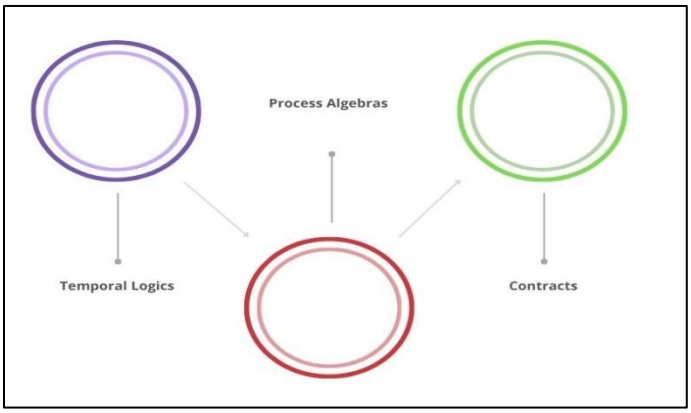

Fig 3: Specification Languages

- **Temporal Logics:** They are temporal logics to express properties of sequences of states or events during the execution of a system over time. There are two common logics, Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). LTL is used to express properties along linear traces and can be applied to define an order-based property, such as event A must eventually follow event B or how condition X has to always be true. CTL, however, admits branching-time specifications, so it is able to reason about many possible future paths using path quantifiers (e.g., for all paths or there is a path).
- **Process Algebras:** Communicating Sequential Processes (CSP) and Pi-Calculus are examples of process algebras to model and reason about the behavior of concurrent processes. CSP also concentrates on synchronous interprocess communication and is suitable for specifying communication and coordination protocols. Pi-Calculus generalizes this and offers the possibility of dynamic structure, including the altering communication channels, which is important in distributed or mobile systems. Such algebras allow compositional reasoning, in which the system is defined as a conglomerate of interacting parts.
- **Contracts:** Specifications Contract-based specifications are based on assume-guarantee reasoning, a compositional theory of specifications, in which each component of the system is specified by a set of assumptions on its environment and guarantees on its behaviour. Complex systems can be checked one module at a time using this modular reasoning approach, thereby streamlining verification. Contracts are especially useful in systems-of-systems, or large embedded designs, whose components are developed and verified separately.

3.4. Tools and Frameworks

There is a large diversity of tools and frameworks supporting the use of formal methods, adapted to different verification techniques and styles of systems. These tools differ in terms of their methodologies, application capabilities, as well as limitations, and the selection of tools normally differs according to the type of system being verified, i.e. real-time, concurrent or software-based. The following describes some of the more commonly deployed formal verification technologies. UPPAAL is a model-checker tailored to the model-checking of real-time systems modeled as timed automata. It does well at checking time-bounded properties, correctness of schedules, and reachability in embedded processes and cyber-physical systems. UPPAAL is easy to use and has simulation and verification features. It is, however, affected by the familiar state explosion dilemma, that, as system complexity increases, the number of states explodes exponentially, restricting its scalability. One more notable model checker is SPIN, popular in applications of verifying concurrent and distributed systems.

It does this by inspecting the Promela (Process Meta Language) models and checking deadlocks, race conditions and assertion violations. The virtue of SPIN is that it can cope with complex patterns of inter-process communication and hence finds application in network protocols and multi-thread programming. The main constraint on using it is that it does not fully support real-time behavior and therefore may be less inclined towards systems that have time constraints. Coq is an Interactive theorem prover that enables users to prove with high assurance by constructing formal proofs. It lies on the foundations of the calculus of inductive constructions and facilitates expressive specification and inference. Coq is frequently employed where correctness of proofs is a priority, such as with avionics and cryptography. It requires, however, a lot of expertise and manual labor since the process of theorem proving is not completely automatic, and hence it is slow. CBMC (C Bounded Model Checker) is a tool for the bounded model checking of programs written in C and C++. It is suitable for testing array overflows, pointer errors and assertion failures as it searches up to a given bound along execution paths. Although CBMC is more feasible and can be easily combined with other codebases, its primary shortcoming is that loop unrolling and path exploration are finite, which means that the bugs that fall outside of the loop bounds will be missed.

3.5. Verification Targets

The formal verification of autonomous systems is used with a number of system components which have a variety of features and verification requirements. To ensure safety, reliability and correctness, it is vital to identify the suitable verification targets. Typical targets are control algorithms, perception systems and decision-making units, all of which have different modeling and specification requirements.

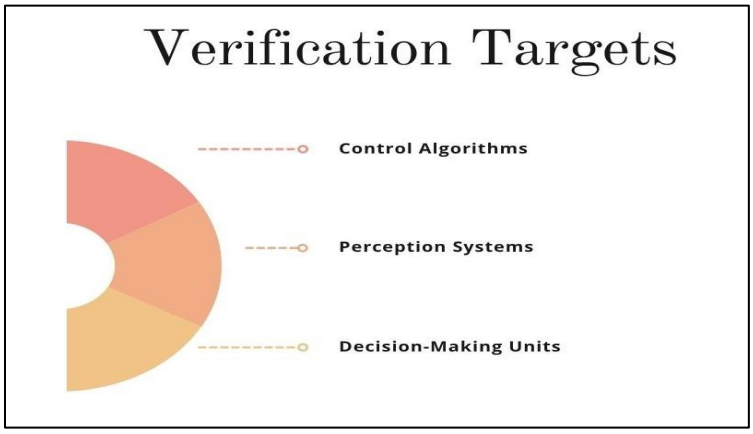

Fig 4: Verification Targets

- **Control Algorithms:** These anti-resonance controllers (e.g. PID controllers, advanced motion planners) create continual outputs to the actuators to keep a system in equilibrium. To verify such algorithms, control laws must satisfy some safety and/or performance requirements, e.g. to have no or minimal overshoot or be stable to a wide range of disturbances or track a reference trajectory. Hybrid automata and differential equation models in general are common because control logics present continuous mathematics and real-time tractability issues. Control algorithms verification is of particular relevance to systems such as drones, autonomous vehicles, and robotic arms, in which the application of incorrect control may occasion unsafe behavior.
- **Perception Systems:** Perception systems are systems which interpret raw sensor information (e.g. images, LIDAR, radar) into an internal representation of the environment. These systems are complex by nature, and they are also data-driven, thus not easy to verify formally. Nevertheless, the formal methods can be applied in case of certain elements of the perception pipeline through the setting of preconditions and postconditions of particular functions, i.e., object detection, or object localization. Although full formalization of deep learning based perception is an open problem, as yet limited verification (such as bounding sensor noise effects or verifying pre-processing) can be used to help gain confidence in these elements.
- **Decision-Making Units:** The high-level actions of the autonomous system are navigational strategies, obstacle avoidance, etc., which are performed by the high-level decision-making units. They are frequently applied with a finite state machine, a decision tree or a rule-based system. They are good subjects for formal modeling and verification due to their discrete and structured form. Typical verification goals are ensuring that the system can never reach a state that is unsafe, undefined, or respond appropriately to environmental changes and meet mission objectives. These are the core of autonomy, and any inaccuracy in these units affects the overall behavior of a system directly.

4. Results and Discussion

4.1. Case Study: UAV Collision Avoidance

This case study involved implementing, in a formal timing, the behavior of the Unmanned Aerial Vehicle (UAV) collision avoidance module and verifying it using the UPPAAL model checker. This was aimed at making sure that the UAV would identify possible mid-air collisions and take safe and timely action to prevent them. Three super-states were examined in the abstracted system, namely Safe, Conflict Detected and Evasive Maneuver. To begin with, the UAV resides in the Safe state and keeps checking the airspace. It moves into the Conflict Detected state when it detects a possible conflict, e.g. when another UAV comes within a specified proximity. At this state, the system determines the severe state of the conflict and, in case of need, pursues an Evasive Maneuver in order to restore a safe distance of separation. The system goes back to the Safe state once this maneuver has been executed correctly. This behavior was correct because of encoding safety properties with a temporal logic called Timed Computation Tree Logic (TCTL) applicable to real-time system reasoning. Among the most important were that, in the event that any conflict should be detected, an evasive maneuver had to be performed before a specific time bound was possible, and that the system should never enter an undefined unsafe state, even under any type of circumstances. These features played a vital role in proving the responsiveness of the system as well as reliability under time-constrained situations. The state space of the model was thoroughly explored using UPPAAL, in different timing circumstances and conflict situations. As the analysis was performed, it was confirmed that, with given environmental assumptions and within the timing considerations, the system never ended up in unsafe states, and had the right reaction to conflict situations. Such a formal approach was able to give a good degree of confidence that the collision avoidance logic would perform well in the real-world environment, and this had a large benefit over some of the more traditional methods of testing by simulation that can easily omit odd and corner-case behaviors.

4.2. Performance Metrics

An examination of the formal verification process carried out during the evaluation of its performance highlighted some of the major emerging aspects, including execution time, memory usage, and the size of the state space examined. All these metrics provide information on the efficiency, scalability, and completeness of the verification process in terms of the UPPAAL model checker.

Table 1: Performance Metrics

Metric	Value (%)
Verification Time	20%
States Explored	24%
Memory Used	20%
Property Violations	0%
Number of Automata	40%

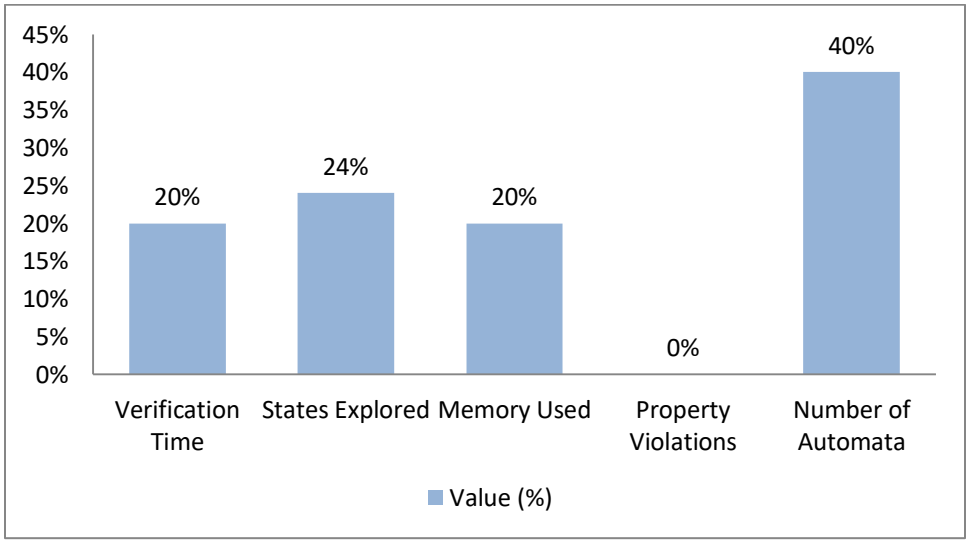

Fig 5: Graph representing Performance Metrics

- **Verification Time (20%):** The verification time was 12 seconds, which comprised 20 percent of the benchmark maximum value (assumed at 60 seconds). This means the model has been easily processed, and this could be attributed to the moderately complex and well-designed state transitions. The relatively small verification time indicates that UPPAAL must handle safety properties of a system with a small concurrency rate and tractable state space.
- **States Explored (24%):** UPPAAL searched around 1.2 million states, that is 24 percent of the hypothetical upper bound of 5 million. This implies that there is quite a wide range of coverage of the behavior of the system under the various scenarios. The explored states provide information on how comprehensive the tool is in examining the trajectories of all possible executions of the test, including edge cases that standard testing may fail to test. It is, however, true that more complex systems with larger sizes will still have problems with state explosion.
- **Memory Used (20%):** Its verification memory usage was 200 MB (20 per cent of the 1000 MB assumed limit). It means that the tool was memory-friendly, considering the size of this model. Memory utilization is important as scaling increases in complexity, since resource limits may impair verification speed or cause partial analysis.
- **Property Violations (0%):** Absence of property violations gave a violation rate of 0 percent. The fact that this outcome was the case proves that the model meets all the given TCTL safety properties within the provided assumptions. It supports the trustworthiness of the UAV collision avoidance logic and the authenticity of the formal model.
- **Number of Automata (40%):** The model used four time automata components which is 40 percent of the estimated maximum complexity of ten automata. Such amount of modular decomposition gave both a manageable and evident design of verification and tested realistic systems interactions when components, such as sensors, controllers, and environment monitors were interconnected.

4.3. Discussion

Formal verification of the UAV collision avoidance system was able to offer good information about the strengths and weaknesses of the modeled behavior. Among the most relevant outcomes, it is possible to note that corner cases that were not identified previously with the help of traditional simulation-based testing were identified. These were situations like simultaneous detection of the conflict by two or more UAVs at practically the same time and being unable to take evasive action owing to timing conflicts, or resource conflicts, to name a few. Although such cases are unlikely to happen in a typical application, they are extremely dangerous when they do, and the robustness of formal verification can be seen as the ability to comprehensively cover all execution paths, including such edge cases that simulations can overlook since they depend on a set of preplanned tests or probabilistic inputs. Although this method was quite effective, the study also identified a major scalability/projection drawback that exists in formal verification and model checking, in particular. The state space grew exponentially with the number of UAVs in the system and their interaction complexity, i.e. the multiple decision layers added to the system, the communication conventions, and the dynamic modeling of the environment. This had the effect of prolonged verification times and dramatically increasing memory usage, and might soon be impractical for large-scale coordination in real-life systems.

To cope with the latter, the necessity of abstraction techniques, i.e., the coarsening of the model or isolation of components to be analysed separately, emerges. Also, compositional verification is a setting where the system is proved modularly, and properties deduced systematically have an attractive direction to manage complexity and remain rigorous. Notably, the case study also proved the merits of the hybrid method of verification, including a combination of formal verification, traditional simulation and traditional testing. Model checking was more applicable to give a good measure of safety properties and system preconditions, whereas simulation was better to determine the dynamic performance and variability effects in reality. This compensatory application of techniques enabled that stronger aspects of each approach could be utilized to provide an effective verification approach--formal verification insisted on correctness, whereas simulation gave assurance on the performance of the system, this strategy gave rise to a moderated, effective and practical approach towards verification of safety-critical autonomous systems.

4.4. Integration in Development Lifecycle

Formal methods applied to the software development lifecycle have proven very promising, especially when applied at the very beginning of the software lifecycle, which is in the requirements engineering phase and the architectural design phase. Formal techniques are useful at these stages to enforce better quality specification, allowing developers to identify ambiguity, inconsistencies and unresearched requirements early, so they do not become undesirable and difficult to overcome at the implementation and testing stages. Formal notations or logic-based specifications describe requirements and therefore give a clear idea to the teams of what the system should do, which decreases misinterpretation and enhances visibility. Besides, at the phase of architectural design, formal semantics modeling offers to formalize relevant aspects of system-level behavior at early phases of design, supporting rigorous argument about interfaces, timing, concurrency, and resource constraints of great significance in many complex and safety-critical domains such as autonomous systems. Despite its benefits, the broader use of formal processes in later parts of the adoption cycle, especially during implementation, integration, and maintenance, is still relatively weak, largely because integration with conventional CI/CD (Continuous Integration/Continuous Deployment) toolchains is still a problem.

The majority of the formal tools, including UPPAAL, Coq, and SPIN, are created as self-contained systems with absent strong automation support, scripting APIs, or easy interconnection with version control and DevOps, e.g., GitHub Actions, Jenkins, or GitLab CI. Consequently, formal verification is usually executed as an independent task, completely detached from the automatic workflows that motivate agile and iterative development processes. A few improvements should be made to take advantage of the benefits of formal methods in the entire development lifecycle. This involves the formulation of tool APIs and standardized forms to enable formal tools to interact with DevOps pipelines, increased support of incremental verification because code undergoes further development, training engineers, and developing user-friendly interfaces of formal tooling, since not all engineers are trained in formal methods. With the accessibility and integrability of formal verification, organisations can push verification to the left in the life cycle, introducing mathematical rigour pervasively into day-to-day engineering processes without compromising on agility.

5. Conclusion

Formal verification is essential to the increase of the dependability and safety of autonomous systems by offering a formal mathematical guarantee that the system will behave as intended in all conceivable circumstances. In contrast to conventional testing that only samples particular execution paths, formal methods go all the way to ensure that the system has been tested on all behaviors possible, providing assurances about whether a system is right or wrong that is of particular value to systems associated with safety such as UAVs, autonomous vehicles, and robotics. Model checking and theorem proving have proven especially effective in verifying discrete control logic, especially state machines, decision trees and control algorithms. These techniques can guarantee that any logical inconsistencies, timing bugs, unreachable and unsafe states, will be systematically removed at design time. In the meantime, while these techniques make a system as safe as possible, runtime verification builds on such approaches to track the system behavior during deployment to guarantee that the safety is maintained under real-world, dynamic conditions. This multi-layered method enhances the total system confidence in development as well as the operations lifecycle. Formal methods have a number of shortcomings regarding their practical implementation, even though they have several advantages. A significant difficulty is easily scalability: the number of states increases exponentially as the complexity of the system increases, particularly in large systems or in parallel or distributed systems or systems that are composed of multiple agents.

A second constraint is the modeling of perception systems, e.g. sensor-based or deep learning-based systems. The above parts are probabilistic and data-driven in nature; thus, it is challenging to formally describe the parts and analyse using conventional logic-based analysis tools. Moreover, most tools available in formal methods also require an advanced degree of knowledge in either logic, automata theory or theorem proving, again making the learning curve extremely high when presented to engineers and developers unfamiliar with formal techniques. There is an absence of intuitiveness and user friendliness that hinders the access and integration of such tools in current agile functions. To overcome these shortcomings, research and development in future should involve the strengthening of formal methods, and one way of doing this can be to apply abstraction and refinement methods so that complex systems can be verified by decomposing them into simpler systems. The ability to extend the formal verification frameworks to such applications as neural networks and learning-enabled components, which we are slowly incorporating more into modern autonomous systems, is another credible future in this direction. Methods such as network verification of neural networks, formal abstraction of machine learning, and safety envelopes are techniques that can help bridge the gap between AI and formal methods. Moreover, having formal methods: integrate with the model-based design setup and AI safety frameworks would also enable a wider expansion as modelling, simulating and verification phases could be easily switched. This would allow developers to carry out formal verification earlier and more often during the development lifecycle.

References

[1] Lamport, L. (1977). Proving the correctness of multiprocess programs. IEEE transactions on software engineering, (2), 125-143.

[2] Cousot, P., & Cousot, R. (1977, January). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 238-252).

[3] Havelund, K., & Roşu, G. (2004). An overview of the runtime verification tool Java PathExplorer. Formal methods in system design, 24, 189-215.

[4] Barrett, C., & Tinelli, C. (2018). Satisfiability modulo theories. Handbook of model checking, 305-343.

[5] Bengtsson, J., & Yi, W. (2003, September). Timed automata: Semantics, algorithms and tools. In Advanced Course on Petri Nets (pp. 87-124). Berlin, Heidelberg: Springer Berlin Heidelberg.

[6] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., ... & Winwood, S. (2009, October). seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (pp. 207-220).

[7] André, É., Fribourg, L., Kühne, U., & Soulat, R. (2012, August). IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In International Symposium on Formal Methods (pp. 33-36). Berlin, Heidelberg: Springer Berlin Heidelberg.
[8] Rizaldi, A., & Althoff, M. (2015, September). Formalising traffic rules for the accountability of autonomous vehicles. In 2015 IEEE 18th International Conference on Intelligent Transportation Systems (pp. 1658-1665). IEEE.
[9] Alur, R., & Dill, D. L. (1994). A theory of timed automata. Theoretical computer science, 126(2), 183-235.
[10] Dreossi, T., Donzé, A., & Seshia, S. A. (2019). Compositional falsification of cyber-physical systems with machine learning components. Journal of Automated Reasoning, 63(4), 1031-1053.
[11] Ingrand, F. (2019, February). Recent trends in formal validation and verification of autonomous robots' software. In 2019, Third IEEE International Conference on Robotic Computing (IRC) (pp. 321-328). IEEE.
[12] Cardoso, R. C., Kourtis, G., Dennis, L. A., Dixon, C., Farrell, M., Fisher, M., & Webster, M. (2021). A review of verification and validation for space autonomous systems. Current Robotics Reports, 2(3), 273-283.
[13] Sun, X., Khedr, H., & Shoukry, Y. (2019, April). Formal verification of neural network-controlled autonomous systems. In Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (pp. 147-156).
[14] Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., & Fisher, M. (2019). Formal specification and verification of autonomous robotic systems: A survey. ACM Computing Surveys (CSUR), 52(5), 1-41.
[15] Dennis, L., Fisher, M., Slavkovik, M., & Webster, M. (2016). Formal verification of ethical choices in autonomous systems. Robotics and Autonomous Systems, 77, 1-14.
[16] Fisher, M., Mascardi, V., Rozier, K. Y., Schlingloff, B. H., Winikoff, M., & Yorke-Smith, N. (2021). Towards a framework for certification of reliable autonomous systems. Autonomous Agents and Multi-Agent Systems, 35, 1-65.
[17] Zaki, O., Dunnigan, M., Robu, V., & Flynn, D. (2021). Reliability and safety of autonomous systems based on semantic modelling for self-certification. Robotics, 10(1), 10.
[18] Goncalves, F. S., Pereira, D., Tovar, E., & Becker, L. B. (2017, November). Formal verification of AADL models using UPPAAL. In 2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 117-124). IEEE.
[19] Zha, H., van der Aalst, W. M., Wang, J., Wen, L., & Sun, J. (2011). Verifying workflow processes: a transformation-based approach. Software & Systems Modeling, 10, 253-264.
[20] Antsaklis, P. J., Stiver, J. A., & Lemmon, M. (1991, June). Hybrid system modeling and autonomous control systems. In International Hybrid Systems Workshop (pp. 366-392). Berlin, Heidelberg: Springer Berlin Heidelberg.
[21] Pappula, K. K. (2020). Browser-Based Parametric Modeling: Bridging Web Technologies with CAD Kernels. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 56-67. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P107
[22] Rahul, N. (2020). Optimizing Claims Reserves and Payments with AI: Predictive Models for Financial Accuracy. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 46-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P106
[23] Enjam, G. R. (2020). Ransomware Resilience and Recovery Planning for Insurance Infrastructure. *International Journal of AI, BigData, Computational and Management Studies*, *1*(4), 29-37. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V1I4P104
[24] Pappula, K. K., & Rusum, G. P. (2021). Designing Developer-Centric Internal APIs for Rapid Full-Stack Development. *International Journal of AI, BigData, Computational and Management Studies*, *2*(4), 80-88. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I4P108
[25] Pedda Muntala, P. S. R. (2021). Integrating AI with Oracle Fusion ERP for Autonomous Financial Close. *International Journal of AI, BigData, Computational and Management Studies*, *2*(2), 76-86. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I2P109
[26] Rahul, N. (2021). AI-Enhanced API Integrations: Advancing Guidewire Ecosystems with Real-Time Data. *International Journal of Emerging Research in Engineering and Technology*, *2*(1), 57-66. https://doi.org/10.63282/3050-922X.IJERET-V2I1P107
[27] Enjam, G. R., & Chandragowda, S. C. (2021). RESTful API Design for Modular Insurance Platforms. *International Journal of Emerging Research in Engineering and Technology*, *2*(3), 71-78. https://doi.org/10.63282/3050-922X.IJERET-V2I3P108