Examination of Model Checking

Utsav Anantbhat,
Martin Chuddy, Alex Ho,
Martin Lau

12/6/2022

—

CMPT 477

—

Professor Wang

Introduction:

In the digital age, we rely highly on computer systems for almost every aspect of our lives. From software that supervises functions in our smartphones and cars to industrial warehouses of Amazon run mostly by autonomous robots, these systems are critical for the world's economy to operate smoothly. In order to ensure that these systems work as efficiently as possible, quality assurance testing of the hardware and software is crucial. This requires formal models that explain the core components of the systems at an abstract level. Model checking is a technique used to formally verify that a system meets required functionality given a specification. This technique has been used to find subtle errors in designs of various systems which often aren't as detectable during testing or simulation. Due to its practicality and compatibility with traditional design methods, model checking is a standard procedure for quality assurance in various systems.

[6] Inputs to model checkers comprise of a system's description and properties, often written as formulas or algorithms, that the system holds. The role of the model checker is to determine if the properties hold or if the properties are violated and report on it. In the event that the property is violated, the checker provides a counter-example. This can prove useful when looking for errors in specific areas, for example, in certain lines of code. Furthermore, the ability of a model checker to determine the lack of errors helps one understand how an error can be produced.

In this paper, we present the model checking technique and how it can be applied to different forms of analyses as well as discuss the various applications that utilize model checking.

Key Words:

- Temporal Operators, Atomic Propositions, Multi Agent Systems, Blockchain **(Paper 1)**
- Knowledge/experience using Uppaal-SMC, familiarity with SMC, Intersection Management **(Paper 2)**
- Autonomous Vehicles, Timed Automata, UPPAAL, Model Checking **(Paper 3)**
- Symbolic Logic, Boolean Expressions, State generalization and Tree Traversal Algorithms **(Paper 4)**
- Traversal Algorithms, System States, Concurrency, DFS/BFS, VeriSoft **(Paper 5)**

Problem Statement:

The papers we shall be investigating discuss different problems that are solved using different model checking tools. Some of these problems include errors in blockchain smart contracts, and safety in autonomous vehicle technology.

Approaches:

Paper 1: Formal Verification of Blockchain Smart Contracts via ATL Model Checking [1]

The paper aims to solve errors that come from blockchain smart contracts, specifically Ethereum smart contracts. The problems that occur in these smart contracts will be solved using an ATL (Alternating-time Temporal Logic) model checker. ATL details the logic of computations for multi-agent systems, thus the model checker that will be used is called MCMAS, which is a model checker for multi-agent systems.

MCMAS will be used to perform error checking in the smart contracts. The idea that is presented by the researchers involves interactions between user and smart contracts as a two-player game. The smart contracts which for Ethereum are written in the Solidity programming language will be translated into a game structure that is supported by the model checker. This is done by contriving translation rules that preserve the semantics of the Solidity language. Then this will be passed to MCMAS to be verified. In addition, properties are represented between two-players in an ATL formula and verify whether the whole system satisfies the properties by MCMAS.

A case study is then performed to test that this model checker can identify issues in smart contracts in the real world. One of them involves an attack in which users can withdraw more Ether than they own from a fund. The smart contracts of the attacker and fund are presented and then translated into the MCMAS language. An ATL formula is then produced for checking whether the attacker can withdraw more Ether than it possesses. The result comes back to be true which means that the attacker can withdraw from the fund and MCMAS produces a counter-example to show. A fix to the DAO's smart contract code is given and rechecked with the same ATL formula to MCMAS and that result comes back false indicating that the attacker cannot withdraw any more Ether.

Discussion of Paper 1:

An issue with this approach is that only a subset of the Solidity language syntax is supported when converting into MCMAS language. Therefore, some of the syntax it cannot deal with are function type variables, dynamic creation of contracts, and inheritance. However, this approach has proven to solve a real-world issue. As well, without being given an attacker smart contract, the most permissive model of an attacker is introduced which is able to verify the issue in the case study. This would mean that developers building smart contracts can use this to verify security issues using this model.

Paper 2: Formal Verification of Heuristic Autonomous Intersection Management Using Statistical Model Checking [2]

The goal of this paper is to prove the safety behind autonomous vehicle technology using statistical model checking on intersection management. The approach used to accomplish this task is accomplished using Statistical Model Checking (SMC) applied on the Heuristic Autonomous Intersection Management (HAIM) algorithm through a model checker named Uppaal. The system Uppaal will be verifying will be modeled using a group of Timed Automata (TA).

The HAIM algorithm will be applied to an intersectional model representing a 3-lane intersection with roads that travel in four directions with two-way traffic using only autonomous vehicles. The HAIM algorithm will utilize 3 schemes named First Enter First Serve scheme, Window scheme, and Reservation scheme which will utilize known potentially points of collision named conflict points (CP) (4 conflict points exists per lane-to-lane traversal) to determine the velocity of the vehicle along its journey under the restrictions of the model. The HAIM algorithm will be modeled in Uppaal using the tools SMC extension (UPPAAL-SMC). To model the intersection system two TA's will be utilized, namely a Traffic Automaton responsible for releasing new vehicles into the intersection system and a Master Automaton which using a dynamic template will be responsible for the initialization of vehicles, setting up the HAIM pipeline control, and assessing movement and collision checks.

Model checking verification is performed on the HAIM algorithm in Uppaal-SMC to verify three constituent layers followed by a verification of the complete model heuristic. The first layer is a Model with Lane Velocity as the Final Velocity which calculates lane velocity by setting the departure time of the currently measured vehicle to a value larger than the previous vehicle in the same lane. The second layer is a Model with FEFS Velocity as the Final Velocity that determines FEFS velocity by determining and solving conflicts for the first two of the four CP. The third layer is a Model with FEFS Velocity and Window Velocity that handles the last two CP and improves the probability of avoiding all intersection collisions. The complete model heuristic pairs all of the layers in our model which will produce a trip that is collision free when traversing through both the lane and intersection.

The validity of this model was further strengthened by generating five satisfiable invariant conditions using conflict points and the previously described schemas combined with artificial error injection testing.

Discussion of Paper 2:

The tools and verification methods used for this paper outline some issues that still need to be addressed going forward in regards to vehicle automation. The researches outlined there are limitations to other verification methods that could be applied to the problem set, such as Timed Petri-Nets (TPN) or Labeled Finite State Automata (LFSA) due to lack of tool sophistication (TPN) and modeling time-critical systems not being feasible (LFSA). Another problem with the problem being addressed, is often the tools do not always determine with certainty that all crashes can be avoided.

Paper 3: Formal Verification of an Autonomous Wheel Loader by Model Checking [3]

As construction environments are often hazardous, the industry is moving towards autonomy. Furthermore, since the environment is dynamic, it is important to ensure safety and efficiency with such machines. The purpose of this paper is to provide a timed automata (TA) description of an autonomous wheel loader's (AWL) control system in order to check the dependability of the loader. This includes algorithms to perform navigation and avoid collisions. The encoding is checked in UPPAAL with given safety and timing requirements.

The autonomous machines are tasked to operate in often harsh environments alongside other equipment and humans. Although some aspects of the environment are controlled, the machines' dependability is essential for increased safety and productivity. Therefore, formally verifying an abstraction of the machines' behavior is beneficial. The approach to verifying the autonomous machines in this paper encodes to A* algorithm for the initial path planning.

The A* algorithm is one of the best techniques used in path-finding traversals. Here, it is used to compute the initial path for the AWL. The purpose of A* is to find a path from the initial state to goal state with the lowest cost, similar to Dijkstra's algorithm. However, unlike Dijkstra's, A* uses a heuristic function to optimize traversal. The cost is calculated using the following function $f(n) = g(n) + h(n)$ with n being the current state, g(n) being the cost from the initial state to current state, and h(n) being the estimated lowest cost from current state to goal state. Therefore, the goal of A* is to produce a path with the lowest f(n) value.

The verification model of the AWL consists of a map which is modeled as a data structure. The map is abstracted into a 2D grid with (x,y) coordinates with various indicators to represent static and dynamic objects on the map. The map is defined as a 2D array in UPPAAL based on this abstraction. Each element of the array is a point on the map and is assigned 0 or 1. Upon applying the modeling process described in the paper, a formal model of the AWL can be created and its environment as a network of TA. The formal model consists of 12 TA with four data structures, two of them for the A* algorithm. The first A* is executed to check for obstacles around AWL and the second A* is used to generate an initial path.

Discussion of Paper 3:

In regards to the A* algorithm, it is effective when solving complex problems as it is complete and optimal. The algorithm is optimally efficient which means that it expands the fewest number of nodes compared to other optimal algorithms. However, A* is only complete if every action has a fixed cost and the given branching factor is finite. Furthermore, the speed of the execution of A* depends highly on the accuracy of the heuristic algorithm.

Paper 4: Automatic Verification of Knowledge and Time with NuSMV. [4]

The variability of software product lines is essential for a company to generate consumer demand through the expansion of their cross-product options, but how do the companies ensure the constraints are maintained when a new feature is implemented? The paper answers this question by proposing an efficient model checking method to verify the validity of the software product line behaviors against temporal properties with CTL and the NuSMV Solvers.

The main problem verifying a set of software product lines is the *"state space explosion problem"* caused by the exponential scaling of system states between the features and the products. The paper suggests encapsulating all the products through a compact model called *"Featured Transition Systems (FTS)"* rather than generating a model for each individual product to decrease the chance of a state space explosion problem from occurring. In addition, symbolic logic is used to encode the FTS to take advantage of the scaling benefit of symbolic logic algorithms when exhaustive solutions are failing. The encoding for the product is represented by a Boolean function in terms of all the features supported by the function. With the FTS encoded, *"feature Computation Tree Logic (fCTL)"*is introduced to create formulas using the FTS as product quantifiers to specify all the products that hold under the fCTL formulae. The semantics of the fCTL formula consists of propositional logic operators, and four unique path operators: next, until, finally, and generally). The path operators designate a constraint that must hold for a given state.

The model checking algorithm used to verify the satisfiability of a fCTL expression converts the expression into a parse tree and decomposes the parent formulae into multiple sub-formulas. The rationality behind using a parse tree is to recursively verify the tree from bottom up. The states in the leaf nodes are evaluated before the states in the parent nodes for satisfiability, and the output of the algorithm contains all the states that satisfies the fCTL formula.

The practical implementation of the fCTL symbolic model checker is done using the built-in model checking functionalities of the NuSMV solver. The fCTL formulas are encoded using the fSMV language, which provides the user capability to generate new variables, overriding the definition of existing variables and changing the values of variables in real-time. Through the NuSMV solver, all state violations incurred by a product due to a particular feature is displayed by the verification tool, thus providing clear insight on software cross-system errors.

Discussion for Paper 4:

Although the combination of symbolic encoding and fCTL formulas improves the scale and complexity of verifying the satisfiability of a set of product line systems, the state space explosion problem has not been resolved for all product line systems. This is due to the number of products represented as states being bottlenecked by the memory capacity of the hardware used to run the model checking models. As a result, symbolic model checking presents only a temporary triage to the underlying issue behind product line verification while the memory capacity of a computer remains fixed at a steady state.

Paper 5: Software Model Checking: The VeriSoft Approach [5]

The objective of this paper is to analyze the applications of model checking in testing software with a focus on concurrent systems. In this context, model checking is used as a testing method that can find difficult to catch errors rather than attempting to prove their absence. This approach is of use due to the massive difficulty of thoroughly testing concurrent systems using traditional testing methods.

The paper considers two technical approaches to model space exploration. The first approach is accomplished using state space exploration. Such exploration begins from the initial state of the system and conducts a recursive search of every successor of all states found within the search. This algorithm will always reach a conclusive answer if there is a finite set of states even in the presence of cycles. However, this approach is not feasible when applied to our problem as there is generally significantly too much information to be encoded and saved in memory.

The second approach is accomplished using a state less search algorithm. This approach uses state-space caching, a technique where encountered states are used to create a restricted cache to track states that have been visited. This technique by itself is extremely inefficient and performs worse than the initially considered algorithm due to regularly redundantly visiting previous states, and we must account for termination of looping behavior caused by acyclic systems in order for the algorithm to terminate for finite inputs. For this approach to work it must be bolstered by the usage of two additional techniques relating to persistent sets and sleep sets. Persistent sets are defined as follows: “A set T of transitions enabled in a state s is persistent in s iff, for all nonempty sequences of transitions”. Sleep sets are sets computed using a technique that takes advantage of information obtained from past components of state searches and utilizes it to reduce the state space of the system, often reducing repeat visitations to only the initial exploration of the state.

The researchers implemented both these algorithms in a model checker application called VeriSoft. Verisoft was chosen for this task due to two unique aspects of the tool that are not utilized by any other model checker, namely: Verisoft does not require the usage of any programming language or specific modeling to accomplish its task and Verisoft is capable of performing state-less search algorithmic approaches.

Discussion of Paper 5:

There are still some issues with the approach used in the second proposed algorithm. The biggest hurdles would be a limitation in the tools available to execute such algorithms, and the algorithm would still need further optimization to work with applications that generate a massive state set. Furthermore, the tool itself exacerbates this problem due to containing an intrinsic non deterministic method in how it evaluates the dataset. These two issues combined mean the tool is not always able to operate on all input problems, and in certain cases is more prone to generate errors.

Discussion of Model Checkers:

The papers have used different model checkers to solve their respective problems. Below we will discuss the model checkers that have been used.

The NuSMV symbolic model checker is designed using both binary decision diagrams and SAT based techniques. The model checker supports both CTL and LTL expressions, and provides concise error logs on the states causing the verification failures. Many of the current model checkers are an extension of the NuSMV model checker with additional features added to efficiently solve their targeted problem. One such example is the MCMAS model checker developed for multi-agent systems.

[7] The ATL model checker MCMAS which is used for multi-agent systems. It uses ordered binary decision diagrams for symbolic representation of state space. This is important in helping with performance as it can speed up computations and use less memory. Like other advanced model checkers, it offers a number of features such as counterexamples and witness generation. MCMAS has been compared with other well-known model checkers like NuSMV and the performance of MCMAS is in line with NuSMV.

The Verisoft model checker is a tool used for exploration of state spaces of concurrent systems obtained through observation and controlled execution of visible operations of the system. Some of the capabilities of Verisoft include checks for violation of deadlocks, assertions, divergences and livelocks and offering information for resolving these possible conflicts. Verisoft is unique in that in its time it offered two new aspects to the tool not seen in other model checkers. Verisoft did not require the usage of any specific model/programming language to tackle the presented problem ,the problem itself could be fit to be displayed using a chosen approach. Verisoft was also different in the regard that it was the first model checker to support the approach of state-less search.

[8] UPPAAL is a tool box for modeling, simulation and verification of real-time systems, based on constraint-solving and on-the-fly techniques. UPPAAL SMC is a model checking approach in UPPAAL that allows us to reason on networks of complex real-time systems with a stochastic semantics. UPPAAL has many advantages. It works with models developed in Timed Automata formalism, supports dynamic instantiation of templates and graphical modeling, offers high-level data structures and functions, and is updated frequently with an active community of users. UPPAAL isn't without its downsides, however. The SMC extension only allows UPPAAL to make an executable model based on a bounded count of simulations rather than exhaustively exploring all possible states. Due to this limitation using the SMC extension does not always guarantee absolute certainty for non-deterministic problems. This also means using the SMC extension may lead to additional errors and cannot support especially large inputs. UPPAAL SMC does not provide exhaustive model checking but rather the probability of satisfying the queries.

Even though many of these model checkers are customized on an application basis, there is a problem that is consistent between all modern model checkers. The problem is known as the "state explosion problem". The state space explosion problem is caused by the memory capacity constraint of the system used to run the model checkers. As a result, it is impossible for any model checker to present a permanent solution to the problem given there is always a limit on hardware memory capacity. To make the issue transparent for users that are planning to use the model checkers, a threshold or "steady state" is provided for the user to set a limit on the size of the model that is going to be verified.

Conclusion:

In this paper, we discussed the applications of model checking in various fields and compared the various model checkers used in these scenarios, weighing their advantages and disadvantages. The applications of model checking are vast and the examination of these model checkers allow us to understand the importance of these systems. Despite its appealing qualities, model checking also has some drawbacks such as limited scalability and the fact that the analysis of the given object is an abstract model. Therefore, further reviews are needed to ensure that the abstract model is representative of the tangible system so that the properties of the system may hold.

Division of Work:

Here is what each member of the group contributed to the project:

- Martin C: Paper 2, Paper 5, parts of Discussion
- Martin L: Paper 1, parts of Discussion, Problem Statement
- Utsav: Paper 3, parts of Discussion, Introduction, Conclusion
- Alex: Paper 4, parts of Discussion, Key words

References:

[1] Nam, W., & Kil, H. (2022b). Formal Verification of Blockchain Smart Contracts via ATL Model Checking. *IEEE Access*, *10*, 8151–8162. https://doi.org/10.1109/access.2022.3143145

[2] Chouhan, A. P. (n.d.). *Formal Verification of Heuristic Autonomous Intersection Management Using Statistical Model Checking*. MDPI. https://www.mdpi.com/1424-8220/20/16/4506

[3] Gu, R., Marinescu, R., Seceleanu, C., & Lundqvist, K. (2018). Formal verification of an autonomous wheel loader by model checking. *Proceedings of the 6th Conference on Formal Methods in Software Engineering*. https://doi.org/10.1145/3193992.3193999

[4] Alessio Lomuscio, Charles Pecheur, & Franco Raimondi. (2007). Automatic verification of knowledge and time with NuSMV. *International Joint Conference on Artificial Intelligence, 1384–1389.*

[5] Godefroid, P. (2005, March 1). Software Model Checking: The VeriSoft Approach. *SpringerLink.* https://link.springer.com/article/10.1007/s10703-005-1489-x?error=cookies_not_supported&code=bb6359d9-1cdb-4308-ae92-195b3f39917b

[6] Merz, S. (2001). Model Checking: A Tutorial Overview. *Modeling and Verification of Parallel Processes*, 3–38. https://doi.org/10.1007/3-540-45510-8_1

[7] Lomuscio, A., Qu, H., & Raimondi, F. (2015). MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, *19*(1), 9–30. https://doi.org/10.1007/s10009-015-0378-x

[8] Larsen, K. G., Pettersson, P., & Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, *1*(1–2), 134–152. https://doi.org/10.1007/s100090050010