Scalable Model Checking: A Practical Exploration

Cayden Lund (cayden.lund@utah.edu)

11 May 2023

1 Introduction

1.1 The Murphi Modeling Language

Model checking plays a vital role in ensuring the reliability and correctness of complex systems, ranging from hardware designs to software applications. As systems grow increasingly intricate and interconnected, the potential for errors and vulnerabilities also escalates. Model checking provides a systematic and formal approach to identify design flaws, uncover subtle bugs, and verify the adherence of systems to desired specifications. By exploring the state space of a model and analyzing the system's behavior against specified properties, model checking aids in identifying critical issues that might lead to malfunctions, security breaches, or safety hazards. The significance of model checking extends beyond the initial development phase, as it can also be employed during system evolution and maintenance to detect unintended consequences of modifications and ensure the preservation of system correctness. With its ability to provide rigorous analysis and verification, model checking plays a pivotal role in improving system dependability and reducing the risk of costly errors and failures.

To facilitate the model-checking process, several modeling languages have been developed, each designed to capture the essential characteristics of a system under scrutiny. One such language is the Murphi modeling language, which has gained popularity due to its intuitive syntax and powerful expressive capabilities.

The Murphi modeling language provides a formal framework for describing system behavior by specifying its state space and the rules governing state transitions. By defining the system's properties and constraints, engineers can create precise and detailed models that capture the essential aspects of the system's functionality.

Moreover, the rigidity of the Murphi modeling language has led to the development of various model-checking tools specifically tailored for Murphi models. These tools, such as `rumur` and `romp`, offer different strategies and algorithms to explore and analyze the state space of Murphi models. By leveraging these tools, engineers can efficiently verify the correctness and identify potential flaws in their systems.

1.2 `rumur` and `romp`

In the context of Murphi model checking, two notable tools have gained prominence: `rumur`[1] and `romp`[2]. These tools offer different approaches to model checking, catering to varying requirements and constraints of the system being analyzed.

`rumur` performs an explicit enumeration of the entire state space, exhaustively examining all possible system states. This exhaustive approach provides a definitive answer regarding the presence of flaws in the model. It is particularly well-suited for smaller models where the state space can be completely covered.

In contrast, `romp` adopts a parallel random walk strategy for model checking. By exploring the state space in a randomized and parallelized manner, `romp` can quickly cover an enormous state space. However, due to its probabilistic nature, it cannot offer strong guarantees of complete bug-freedom in the model.

The choice between `rumur` and `romp` depends on the characteristics of the system being analyzed. For smaller models, where it is feasible to exhaustively cover the state space, `rumur` can provide a definitive answer on the presence of flaws. On the other hand, for models with enormous state spaces where exhaustive enumeration is infeasible, `romp` offers a practical and scalable approach to detect potential bugs.

In this paper, we present a practical exploration of scalable model checking using `rumur` and `romp`. We demonstrate their effectiveness by examining two buggy models: a tic-tac-toe strategy and a model for the leader election protocol. Through these case studies, we evaluate the capabilities of `rumur` and `romp` in identifying and correcting flaws, shedding light on their respective strengths and limitations in different scenarios.

1.3 Supporting Files

In support of this paper, several files have been provided to facilitate the reproduction and understanding of the presented work. These files are located in the `Models/` directory and serve various purposes in the context of the case studies. The following supporting files are included:

- `tictactoe_buggy.m`: This file contains the initial version of the buggy tic-tac-toe strategy that was subjected to debugging and analysis in this paper. It serves as a starting point for the investigation and showcases the flaws in the original strategy.
- `tictactoe_fixed.m`: This file contains the corrected version of the tic-tac-toe strategy after the debugging process. It represents the improved strategy that addresses the identified flaw and ensures a winning or tied gameplay.

[1] `https://github.com/Smattr/rumur`.

[2] `https://github.com/civic-fv/romp`. Note that this is currently a private repository, so readers may receive a 404 error.

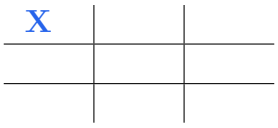

Figure 1: The player's first move.

- `leaderelection_buggy.m`: The `leaderelection_buggy.m` file contains the original version of the buggy leader election protocol. It is the basis for the analysis and correction of the protocol, and has a bug in its implementation that we identify and correct.
- `leaderelection_fixed.m`: This file includes the fixed version of the leader election protocol.
- `Makefile`: The `Makefile` is a utility file provided to automate the building and compilation of the model-checking executables. It simplifies the process of generating the necessary binaries for running the analysis on the provided models.

These supporting files, along with the accompanying explanations and discussions in the paper, enable readers to reproduce the experiments, examine the model structures, and gain a comprehensive understanding of the presented case studies.

2 Case Study 1: A Tic-Tac-Toe Strategy

Tic-tac-toe is a classic two-player game played on a 3×3 grid. The game is also known as *noughts and crosses* or *X's and O's*, with one player marking the squares with X's and the other with O's. The objective of the game is for a player to get three marks in a row of that player's type, either horizontally, vertically, or diagonally, before the opponent. If all the squares are filled without a winner, the game is considered a draw.

2.1 The Tic-Tac-Toe Strategy

The goal of this strategy is to never allow the opponent to win. This means that either the agent employing the strategy (from here on, *the player*) wins the game, or the game ends in a draw.

In this strategy, the player is assumed to go first. The player's mark is X. The opponent's mark is O.

The player's first move is in the top-left corner of the board, as shown in Figure 1. From there, the opponent may place a mark in any of the eight remaining locations on the board. Depending on the opponent's next move, the player will react accordingly.

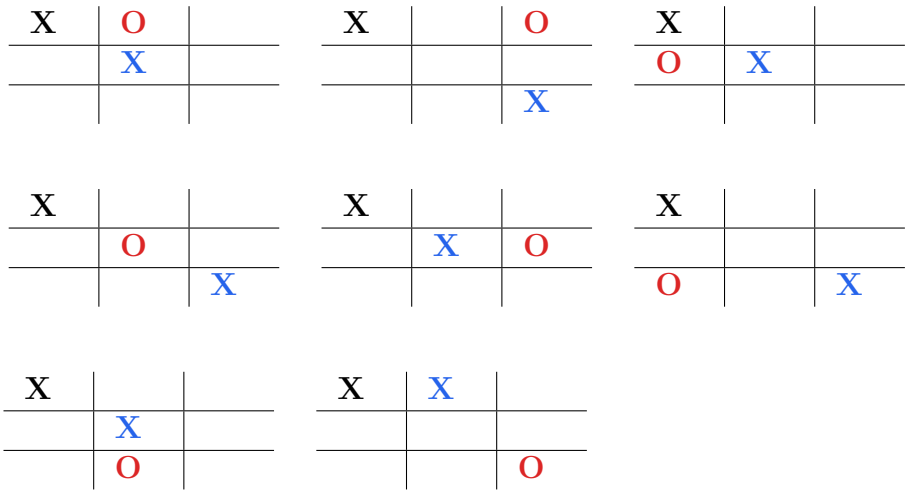

Figure 2: The player's second move.

Figure 2 shows the lookup table for the player's moves for the second turn of the game. For brevity, the entire strategy isn't described in the paper; see the accompanying file `tictactoe_buggy.m` for details.

2.2 The Flaw in the Strategy

Under a certain sequence of moves, the tic-tac-toe strategy, which is under examination in this case study, inadvertently allows the opponent to secure a winning position. Specifically, there exists a particular sequence of moves where the strategy fails to anticipate a strategic counter-move from the opponent, resulting in an unfavorable board configuration. This sequence is shown in Figure 3. The flaw in the strategy opens up an opportunity for the opponent to strategically place marks and create a winning pattern, exploiting the strategy's vulnerability. By identifying and addressing this specific sequence of moves, we aim to rectify the flaw and enhance the strategy's effectiveness in preventing the opponent from achieving a winning position.

2.3 Identifying the Flaw with `rumur` and `romp`

To conduct a thorough analysis of the tic-tac-toe strategy, we encoded it as a Murphi model. The process involved capturing the key elements and decision-making rules of the strategy within the formal framework of the Murphi modeling language. We defined the state space of the model, representing the various configurations of the tic-tac-toe game board (i.e., the positions occupied by X's and O's) and the current player's turn. Additionally, we encoded the strategy's logic for determining optimal moves based on the current game state. By encapsulating the tic-tac-toe strategy within the Murphi model, we created a formal

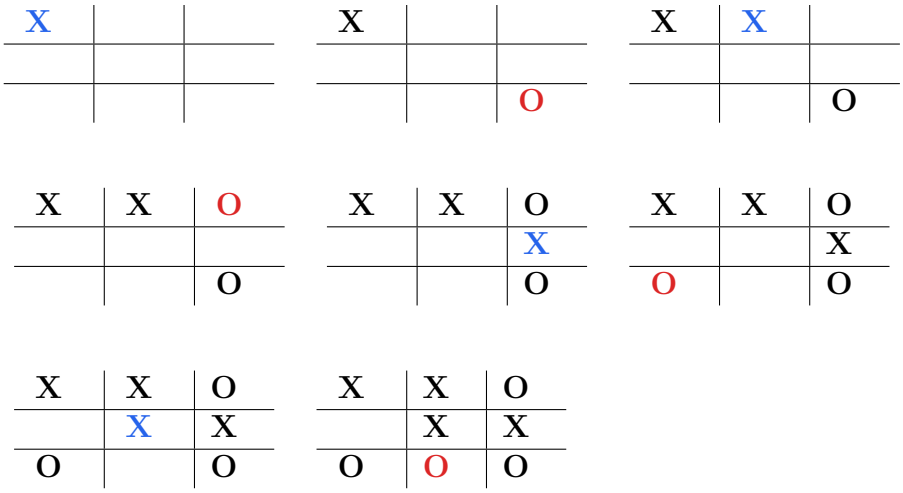

Figure 3: The losing sequence.

representation that allowed us to apply model-checking techniques, such as using `rumur` and `romp`, to systematically analyze and evaluate the strategy's effectiveness and identify any underlying flaws or vulnerabilities. This model can be found in the accompanying file `tictactoe_buggy.m`.

First, we analyze the model with `rumur`. `rumur` has a very user-friendly command-line interface: we first invoke the `rumur` command and supply it with three arguments: the filename of the model, `-o` (for "out"), and the output filename for a generated C file. `rumur` then parses the model and generates a C file that will enumerate the entire state space of the model and identify any underlying faults. After that, we compile the C file and run the generated binary.

Running the compiled program reveals that there was a sequence in which Player 2 wins the game. Further, it shows the complete trace of the losing sequence from the start state to the losing state, with all transitions. This sequence is shown in Figure 3.

Analysis of the model with `romp` is not quite as straightforward. First, we generate C++ code for the model by invoking the `romp` command with four arguments: `-S` (for "simple traces"), the filename of the model, `-o` for ("out"), and the output filename for a generated C++ file. `romp` then parses the model and generates a C++ program that will perform a parallel random walk of the model's state space. That is, several threads will simultaneously run the model from the starting state and perform random transitions from the starting state. This process is repeated several times, so an enormous state space is covered in a short amount of time. We compile the C++ program and run the generated binary with the `-t` flag (for "traces").

Upon being run, the generated program reports that there is a losing sequence in which the opponent wins the game. Because of the random nature of these parallel walks, any readers repeating the experiment here may need to run the binary a couple of times for it to come across a losing sequence. The program's invocation saves ninety-six different traces as

X	X	O
	#	X
O	#	O

Figure 4: The disadvantageous state. The opponent can win in two ways.

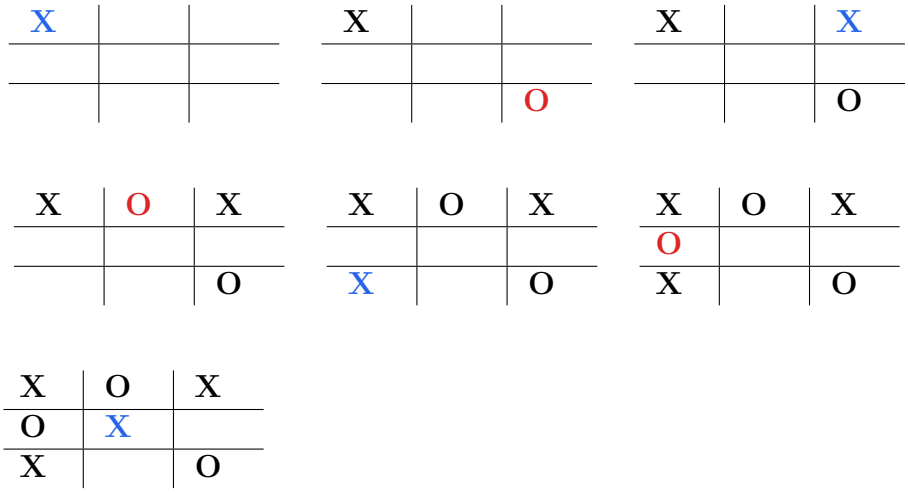

Figure 5: A winning sequence with the new strategy.

JSON files, and reports which one of the traces contains the losing sequence. At every state in the sequence, the trace lists every rule, including the rules that don't apply to the state, before selecting and applying an appropriate rule. Thus, this trace is very large and difficult to read by hand. We recommend using some kind of filtering utility like `sed` to trim down the trace file to a readable size, holding only relevant information.

2.4 Correcting the Flaw

By analyzing the sequence, we see that this sequence of moves allows the opponent to reach a state where there are two unblocked two-in-a-rows. That is, after the opponent takes a turn, there are two spaces on the board where, if the opponent were to take another turn, the opponent would win. This guarantees that the opponent will win, as the player can only place a mark in one of these two spaces. This state is shown in Figure 4, with # representing a blank space that the opponent can fill to win the game. The player can only fill one of these two spaces on the player's turn.

To correct the flaw in the strategy, we need to prevent this from happening. We change the second move of the game so that the player, instead of putting a mark in the top-center space, will put a mark in the top-right space. By making this change, the opponent is never able to reach the advantageous state where there are two blank spaces that the opponent can fill to win the game. A sample game played with the fixed strategy is shown in Figure 5.

This corrected strategy can be found in the accompanying file `tictactoe_fixed.m`.

3 Case Study 2: The Leader Election Protocol

3.1 The Leader Election Protocol

In this case study, we examine the leader election protocol[3] implemented in the Murphi modeling language. The leader election protocol aims to identify the process with the highest value among a group of four hundred processes. These processes are arranged in a circle, where each process can only receive messages from its left and send messages to its right. Each process initially starts in the `ACTIVE` state and is assigned a unique value. The protocol operates based on a set of rules and message passing between the processes.

The leader election protocol defines two main states for the processes: `ACTIVE` and `PASSIVE`. Initially, all processes are in the `ACTIVE` state. A process transitions to the `PASSIVE` state when it learns that its value cannot be the maximum value among all processes.

In the `PASSIVE` state, a process only receives messages marked `ONE` or `TWO` from its left and forwards the messages to its right. If it receives a message marked `FINAL`, then it records the value, transitions to the `FINISHED` state, and then forwards the message to the process on its right.

In the `ACTIVE` state, each process follows a set of rules for exchanging messages and determining the maximum value. First, each `ACTIVE` process sends its own value to the right and then awaits the value of the closest `ACTIVE` process on its left. This message, tagged as `ONE`, is saved for later use as a comparison to determine the maximum value. After saving the `ONE`-tagged message, the `ACTIVE` process then forwards that message to the process on its right with the tag `TWO`.

If an `ACTIVE` process receives a value that is different from the one it sent, it waits for the value of the second-closest `ACTIVE` process on its left. This message, tagged as `TWO`, is then compared with the process's value and the previously-saved value of the closest `ACTIVE` process. If the value of the current process is the largest of the three (i.e., the current process's value, the value of the closest `ACTIVE` process on the left, and the value of the second-closest `ACTIVE` process on the left), the current process retains its value and remains `ACTIVE`. Otherwise, it transitions to the `PASSIVE` state.

When an `ACTIVE` process receives the same value it sent, it concludes that it is the only `ACTIVE` process with the maximum value. This is a safe conclusion because each process has a unique value, and because a process receiving its own value means that the process is its own closest active process, meaning that it is the only `ACTIVE` process. When this happens, the process sends its value once more, tagged as `FINAL`, and transitions to the `FINISHED` state.

[3]Developed by Dolev, Klawe and Rodeh; outlined by Edmund M. Clarke.

If a process receives a message tagged as `FINAL`, it saves the received value as the maximum value, forwards the message to its right, and transitions to the `FINISHED` state.

When a process in the `FINISHED` state receives a message tagged `FINAL`, then all processes should be in the `FINISHED` state. At this point, the assertions are made that all processes are in the `FINISHED` state; that there are no pending, unreceived messages; and that the saved maximum value is correct for all processes.

3.2 The Flaw in the Model

We created a Murphi model of the leader election protocol. This model has a rule named "Active process: awaiting `ONE`: found max value". The condition upon which this rule can be applied is incorrect; the model does not check whether the received value tagged `ONE` is equivalent to the current process's value. That is, any `ACTIVE` process receiving a value tagged `ONE` can apply this rule, mark itself as `FINISHED`, and send a message tagged `FINAL`, even if the received `ONE`-tagged value is not equivalent to the process's actual value.

This model can be found in the accompanying file named `leader_election_buggy.m`.

3.3 Identifying the Flaw with `romp`

The state space of this model is enormous.

Initially, we utilize the `romp` command to generate C++ code for the model, using the same four arguments as we did for the tic-tac-toe strategy. The `-S` flag is specified to generate "simple traces" of the model. We also provide the filename of the model and use the `-o` flag to indicate the desired output filename for the resulting C++ file. Upon invocation, `romp` parses the model and generates a C++ program that will perform a parallel random walk through the state space. Multiple threads concurrently walk through the model from the initial state making random transitions, repeating this process several times to cover an extensive state space quickly. After compiling the generated C++ program, we execute the resulting binary with the `-t` flag, which instructs the program to record traces from the model's execution.

The generated program upon execution detects the presence of several sequences where the assertions are violated. That is, either there are unreceived messages, not all processes are in the `FINISHED` state, or the processes haven't recorded the correct `FINAL` value. The program's execution produces ninety-six distinct traces, saved as JSON files, and identifies the traces that demonstrate violated assertions. These trace files are substantive, and tedious to interpret manually; as such, we suggest using a filtering utility like `sed` to reduce the trace file to a manageable size for easier analysis.

These trace files show that processes are incorrectly entering the `FINISHED` state and sending a `FINAL` message. We can see that the "Active process: awaiting `ONE`: found max value" rule is being applied when not appropriate. Upon viewing the condition upon which

this rule can be applied, we see that the rule is not checking that the received value is equivalent to the current process's actual value; we tighten this condition and the resulting model (`leader_election_fixed.m`) successfully passes all checks.

3.4 A Word on `rumur`

The leader election protocol involves a considerable state space due to the presence of four hundred processes and various possible combinations of their states. Unfortunately, the size of this state space surpasses the capabilities of `rumur`, rendering it ineffective for comprehensive analysis. `rumur` relies on explicit enumeration, systematically exploring every potential state of a model. However, the enormous scale of the state space in the leader election protocol makes it impractical to exhaustively cover all possible states using `rumur`. Consequently, we are unable to obtain meaningful results from `rumur` regarding the correctness of the model or the identification of flaws. To overcome this limitation, we employ the parallel random walk strategy offered by `romp`, which allows us to navigate the extensive state space efficiently and detect potential bugs in the leader election protocol.

4 Results and Discussion

4.1 Comparing the Performance and Effectiveness of `rumur` and `romp`

For the tic-tac-toe strategy model, both `rumur` and `romp` successfully identified the flaw in the strategy that allowed the opponent to win under a certain sequence of moves. The exhaustive enumeration approach employed by `rumur` provided a definitive answer, confirming the presence of the flaw. On the other hand, `romp` utilized its parallel random walk strategy to efficiently explore the state space and identify the same flaw. Both tools proved effective in detecting and localizing the bug, enabling the correction of the flawed strategy.

In the case of the leader election protocol model, the size of the state space proved too large for `rumur` to perform a comprehensive analysis. As a result, we could not obtain conclusive results from `rumur` regarding the correctness of the model or the identification of flaws. `romp`, however, with its ability to handle large state spaces through parallel random walks, successfully identified a bug in the model. The flaw was rectified after its detection, strengthening the correctness of the leader election protocol. After this flaw was corrected, `romp` detected no more issues.

4.2 The Trade-offs Between Exhaustive Enumeration and Parallel Random Walks

The comparison between `rumur` and `romp` highlights the trade-offs associated with the choice between exhaustive enumeration and parallel random walks in the context of model checking.

`rumur`, with its exhaustive enumeration approach, guarantees that every possible state is explored, providing a definitive answer regarding the presence of flaws. This method is particularly suitable for smaller models with manageable state spaces, where it is feasible to exhaustively cover all states. When confronted with models characterized by vast state spaces, however, such as the leader election protocol, the exhaustive enumeration approach becomes impractical due to the rapid growth in the number of states to be explored.

On the other hand, `romp`, leveraging parallel random walks, offers a scalable and efficient approach to model checking. By simultaneously exploring multiple states and performing random transitions, `romp` covers an extensive state space in a short amount of time. This approach is well-suited for models with large and complex state spaces, where exhaustive enumeration is infeasible. The probabilistic nature of parallel random walks introduces a level of uncertainty, however, as it cannot provide strong guarantees of complete bug-freedom in the model.

The choice between `rumur` and `romp` depends on the characteristics of the model and the specific requirements of the analysis. For smaller models where exhaustive coverage is possible, `rumur` can provide definitive results. Conversely, for models with enormous state spaces where exhaustive enumeration is infeasible, `romp` offers a practical and scalable approach to detect bugs.

4.3 Areas for Future Research

The exploration of scalable model checking using `rumur` and `romp` has provided valuable insights into their effectiveness and trade-offs. However, there are several areas for future research that can further enhance model-checking techniques.

One direction for future research is the development of hybrid approaches that combine the strengths of exhaustive enumeration and parallel random walks. Such approaches could intelligently switch between strategies based on the characteristics of the model or employ heuristics to determine the most suitable approach at different stages of the analysis. A hybrid approach could even be used to make exhaustive guarantees for granular functions and rules, and then parallel random walks could be used to analyze the other functions and rules with too large a state space to entirely enumerate. By leveraging the benefits of both exhaustive enumeration and parallel random walks, these hybrid techniques could improve the efficiency and accuracy of model checking, especially for models with varying levels of complexity.

Another direction for future research is to minimize the state space through induction. The leader election protocol is a good example of how this could be effective: if the protocol works for, say, three processes, four processes, and five processes, then one can apply induction to show that the protocol will work for any arbitrary number of processes beyond that. Such an approach could detect this and exhaustively enumerate the states of a model with three, four, and five processes, and then apply induction to prove that the protocol will work for four hundred processes, or for even higher magnitudes.

Finally, there is room for future research in regard to scalar sets; the case studies of the tic-tac-toe strategy and the leader election protocol use primitive C-style arrays to represent all parts of the state. We have done no experimentation on models with scalar sets.

5 Conclusion

5.1 Summary of the Main Findings

In this paper, we conducted a practical exploration of scalable model checking using the `rumur` and `romp` tools for the Murphi modeling language. We presented two case studies: a tic-tac-toe strategy and a model for the leader election protocol. Through these case studies, we evaluated the performance and effectiveness of `rumur` and `romp` in identifying and correcting flaws in the models.

In the tic-tac-toe case study, we encoded the strategy as a Murphi model and used both `rumur` and `romp` to identify a flaw that allowed the opponent to win under a certain sequence of moves. After correcting the flaw, both tools confirmed the model's correctness. This demonstrated the ability of both `rumur` and `romp` to effectively detect and rectify flaws in smaller models.

In the leader election protocol case study, we analyzed a model with a large state space, making it infeasible for `rumur` to perform exhaustive enumeration. `romp` successfully detected a bug in the model, highlighting its scalability and efficiency for models with enormous state spaces. The bug was corrected, ensuring the model's adherence to the desired specifications.

5.2 Final Remarks

Our exploration of scalable model checking using `rumur` and `romp` has provided valuable insights into their performance, effectiveness, and trade-offs. The choice between using an exhaustive state-space enumeration and a parallel random walk depends on the characteristics of the model and the scale of the state space. `rumur` excels in smaller models, offering exhaustive enumeration and definitive results, while `romp` is more suitable for larger models, providing a scalable approach with efficient coverage of enormous state spaces.

Model checking plays a crucial role in ensuring the reliability, correctness, and safety of

complex systems. By systematically exploring the state space and analyzing system behavior against specified properties, model checking aids in identifying critical flaws and vulnerabilities. The practical examination of scalable model checking techniques using `rumur` and `romp` contributes to the advancement of formal verification methods and provides engineers and developers with powerful tools to enhance system dependability.

The exploration of scalable model checking using `rumur` and `romp` has provided valuable insights into their effectiveness and trade-offs. Future research directions include the development of hybrid approaches that combine exhaustive enumeration and parallel random walks to leverage the strengths of both strategies, enhancing the efficiency and accuracy of model checking for models with varying levels of complexity. Additionally, research can focus on minimizing the state space through induction, allowing for the application of proven protocols to larger-scale systems. Furthermore, investigating the use of scalar sets in models presents an area for future research, as the current case studies have primarily used primitive C-style arrays. These research avenues will further advance model-checking techniques and expand their applicability in various domains.

In conclusion, scalable model checking techniques, exemplified by `rumur` and `romp`, offer valuable insights and capabilities for system verification. By leveraging these tools and exploring future research directions, we can ensure the correctness and reliability of increasingly complex systems, mitigating risks and enabling the development of robust and dependable software and hardware solutions.