Automated and Efficient Test-Generation for Grid-Based Multiagent Systems

Comparing Random Input Filtering versus Constraint Solving

SINA ENTEKHABI and WOJCIECH MOSTOWSKI, Halmstad University, Sweden
MOHAMMAD REZA MOUSAVI, King's College London, UK

Automatic generation of random test inputs is an approach that can alleviate the challenges of manual test case design. However, random test cases may be ineffective in fault detection and increase testing cost, especially in systems where test execution is resource- and time-consuming. To remedy this, the domain knowledge of test engineers can be exploited to select potentially effective test cases. To this end, test selection constraints suggested by domain experts can be utilized either for filtering randomly generated test inputs or for direct generation of inputs using constraint solvers. In this paper, we propose a domain specific language (DSL) for formalizing locality-based test selection constraints of autonomous agents and discuss the impact of test selection filters, specified in our DSL, on randomly generated test cases. We study and compare the performance of filtering and constraint solving approaches in generating selective test cases for different test scenario parameters and discuss the role of these parameters in test generation performance. Through our study, we provide criteria for suitability of the random data filtering approach versus the constraint solving one under the varying size and complexity of our testing problem. We formulate the corresponding research questions and answer them by designing and conducting experiments using QuickCheck for random test data generation with filtering and Z3 for constraint solving. Our observations and statistical analysis indicate that applying filters can significantly improve test efficiency of randomly generated test cases. Furthermore, we observe that test scenario parameters affect the performance of the filtering and constraint solving approaches differently. In particular, our results indicate that the two approaches have complementary strengths: random generation and filtering works best for large agent numbers and long paths, while its performance degrades in the larger grid sizes and more strict constraints. On the contrary, constraint solving has a robust performance for large grid sizes and strict constraints, while its performance degrades with more agents and long paths.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Test Input Generation, Domain Specific Languages, Test Selection, Autonomous Agents, Multiagent Systems, Grid-based systems, Constraint Solving, Test Input Filtering

ACM Reference Format:
Sina Entekhabi, Wojciech Mostowski, and Mohammad Reza Mousavi. 2022. Automated and Efficient Test-Generation for Grid-Based Multiagent Systems: Comparing Random Input Filtering versus Constraint Solving. *J. ACM* 1, 1 (August 2022), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Testing typically accounts for more than half of the software development costs [35]. Test automation, e.g., using Model-Based Testing (MBT) [27] or Property-Based Testing (PBT) [10], mitigates this

Authors' addresses: Sina Entekhabi, sina.entekhabi@hh.se; Wojciech Mostowski, wojciech.mostowski@hh.se, Halmstad University, Halmstad, Sweden; Mohammad Reza Mousavi, King's College London, London, UK, mohammad.mousavi@kcl.ac.uk.

0004-5411/2022/8-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

problem by generating tests at low additional cost once a model or a suitable property specification is in place. However, for complex systems and specific application areas, several problems remain. In particular, in autonomous and AI-enabled systems, the input domain is a huge multi-dimensional data space and it is not always clear how an effective sampling can be made. Second, test execution can be very time- and resource-intensive, even if it is limited to a simulation environment, let alone in the hardware- and vehicle-in-the-loop settings. Finally, for autonomous systems it is challenging to find effective test-cases that can test their robustness in critical cases, where improvements to the system are still needed.

1.1 Context and Approach

In our earlier work [14], we proposed a domain specific language (DSL) for grid-based multiagent systems that enables the testing engineer to narrow down the test case context to cases that are more likely to uncover faults. Using an experiment, we have shown that such guided test generation can lead to significant improvements in terms of the time required to reach a fault in a system. In the current work, we extend our conference publication by studying and comparing two alternative test input generation approaches: one that uses random data selection followed by filtering, and one based on constraint solving using an SMT solver.

The general context of our work is to provide some criteria for choosing between the two aforementioned automated test input generation techniques, i.e., random generation with filtering versus constraint solving. To this end, we also show the dependency of efficiency on the parameters of the testing scenario and on the complexity of test selection constraints specified by our DSL. These provide the test engineer with criteria to choose a suitable method for generating test data and also opens the possibility for further investigation on how to optimize the testing process by combining the two above-mentioned approaches. The concrete context and the particular contribution towards this goal are detailed in the following.

1.2 SafeSmart Project

The specific context of our work is the SafeSmart project [47], which investigates the *Safety of Connected Intelligent Vehicles in Smart Cities* from different perspectives. These perspectives include vehicle-to-X (V2X) communication, localization of objects on the road, and the control of vehicles. The context of the project is dense urban traffic, and the primary technique to validate the developments is simulation. Our particular objective is the application of model-based techniques [27] for simulation-based testing in this domain. We started off by using Property-Based Testing (PBT) with automatic random test data generation [14], and we now move to other test data generation methods.

1.3 Contributions

This work is an extension and continuation of an earlier paper published at ICTSS 2021 [14]. In our previous work [14], we devised and formalized a locality-based test selection DSL for grid-based multiagent systems and proposed a methodology for its application to filtering randomly generated test cases. To show the impact of this approach, we had partially implemented the DSL in Erlang; the implementation was sufficient to conduct our intended experiments, and statistically analyzed their results. In particular, we pursued the following research questions in that work:

RQ1: Can random generation and filtering test cases make fault detection more efficient in grid-based multiagent systems?

RQ2: Can random generation and filtering test cases lead to a more efficient process for finding the most concise failing test case in grid-based multiagent systems?

In this paper, we consider one of the threats to the validity of the previous work and conduct additional experiments for different problem sizes and analyze the results in each case. We also consider analyzing the efficiency in terms of time, along with the number of SUT executions, which is the most time-consuming part of testing in our domain. Furthermore, we introduce constraint solving as an alternative method for generating test cases with the proposed DSL. We improve upon the experiment design and analysis for answering new research questions, namely:

RQ3: How does test case generation efficiency by random generation and filtering compare with test case generation by constraint solving in grid-based multiagent systems?

RQ4: How do problem domain and constraint complexity influence test case generation time with either of the two methods in grid-based multiagent systems?

The main goal of this paper is to define the criteria for suitability of random test data filtering versus data generation by constraint solving given the complexity of the planning problem (i.e., the number of agents, the grid size, and the agents' path length) and that of the constraint (i.e., its strictness). We methodically compare the efficiency of filtering random test data and constraint solving approach in generating efficient test cases considering domain constraints. We observe that, while both approaches have a general promise of making testing more effective (we provide an experiment setup and results to show this for the filtering of the random test data), they do indeed show distinct characteristics with respect to the particular testing parameters. For example, just increasing the grid size considerably affects the performance of random test data generation with filtering approach, while it does not affect the constraint solving approach in a significant way. The overall goal is to improve the testing efficiency by choosing the most efficient test data generation method, depending on the test scenario context. In addition, we fix some small inaccuracies in defining the semantics of the DSL in [14]. Moreover, for the PBT tool QuickCheck we provide a complete implementation of our proposed DSL for filtering random test cases, which in [14] was implemented only partially, and for direct generation of test cases, we provide a stand-alone implementation in Python by Z3 [34] in this work. The repository containing the code and experiment data is available at [13]. As admitted above, for constraint solving, for now, we have concentrated only on the test data generation performance and not yet on the overall performance of the test execution efficiency in terms of time to reach the fault. Incorporating Z3 into QuickCheck is not a trivial task, and we feel that devising a way to combine the filtering of random data approach with constraint solving is an effort better spent before the complete method is implemented properly in a tool.

1.4 Paper Structure

The rest of this paper is structured as follows: we start with giving some technical background of this work in Section 2 and discussing the related work in Section 3. We explain our testing methodology in Section 4 and propose our DSL for formalizing test selection constraints in Section 5. In Section 6, we represent two approaches for generating selective test input data. To answer the introduced research questions, we design two sets of experiments in Section 7 and present the results along with analytical discussion in Section 8. The threats to the validity of our work are discussed in Section 9, and the paper is concluded with a short summary of the results and our ideas for future work in Section 10.

2 BACKGROUND

QuickCheck and Z3 are the tools that we use in this work for implementing our approach and conducting the required experiments. They are briefly introduced in this section.

2.1 QuickCheck

In the context of the SafeSmart project, we use an advanced Property-Based Testing (PBT) tool QuickCheck[1] [3]. Automatic input data generation in QuickCheck is supported by dedicated random data generators for different data types (numbers, lists, vectors) and the ability to compose generators to build more complex data structures. Reaching more selective test cases is also possible in QuickCheck by filtering the automatically generated test inputs with a defined predicate. The implementation and specification language of QuickCheck is Erlang [6], which is a functional, weakly typed, inherently distributed, and platform-independent programming language.

In QuickCheck, when a generated test fails, to ease debugging, the tool will attempt to find a more concise failing test input. This process is called *shrinking*. Module 1 presents the pseudo-code of the shrinking process for a failed test input, its corresponding data generator, an SUT, and a test selection filter. If shrinking is possible, QuickCheck repetitively tries to find a "smaller" input than the previous candidate (line 4). This smaller input is achieved by modifying the previously determined candidate based on its corresponding data generator. This modification follows data type specific QuickCheck heuristics, e.g., positive numbers are made smaller, while lists are made shorter. After obtaining a smaller input, QuickCheck retries the test with that input (line 6). If the test failed for that input, QuickCheck has gotten one step closer to the most concise failing input; this is called a *successful shrinking attempt* (line 7). Otherwise, if a smaller input data did not lead to a failed test, the process would backtrack and try other ways of reducing the test input. This is called a *failed shrinking attempt* (line 9). This process continues until no more successful shrinking attempt is possible, and the last input is reported as the most concise failing test.

In the case of choosing a test selection filter, the same filter is also used in the shrinking process (line 5). By default, QuickCheck assumes that inputs not satisfying the filtering criterion would not make the test fail (or at least not produce the same failure as the original one). Therefore, if a modified input violates the filtering constraint, it will be just discarded (line 11). In case of having no filter, all modified inputs are considered for shrinking.

Module 1. QuickCheck shrinking process

```
1  shrink(input, generator, SUT, filter):
2      candidate = input
3      while ( shrinkMorePossible(candidate, generator) ):
4          temp = shrinkInput( candidate, generator)
5          if ( filter(temp) ):
6              if( test(SUT, temp) is failed):
7                  candidate = temp  //successful shrinking attempt
8              else
9                  skip                  //failed shrinking attempt
10         else
11             skip                  //discarded shrinking attempt
12     return candidate
```

2.2 Z3 SMT solver

Z3 is a state-of-the-art constraint solving technology from Microsoft Research [34]. It comes from a family of Satisfiability Modulo Theory (SMT) solvers, which allow for extending Boolean satisfiability checking with predicates from other theories (than just Booleans, such as integers or sequences/arrays or more advanced data types). It is implemented in C++, but it has APIs for

[1] http://www.quviq.com/products

several programming languages, such as Java and Python. Similar to other constraint solvers, Z3 takes intended variables, their domains, and the constraints among them as input. Then, it searches for a set of assignments to all of the given variables from their domain that satisfy all constraints and reports that as a solution. One of the features of Z3, elaborated on later in the paper, that proved useful in our application is the possibility of diversifying the produced solutions by using a random seed.

3 RELATED WORK

In designing test suites, scenario-based testing is commonly used for testing autonomous agents. Organizations such as ASAM [17], EuroNcap,[2] and DOT [36] have designed scenarios and specification languages for this purpose. Test scenarios can also be extracted by analyzing crash data [5, 37] or naturalistic driving data [20, 30, 42]. Usually, each test scenario targets a particular corner or critical case of the system. To gain more confidence, one might be interested in testing different configurations of a critical situation. However, running and evaluating autonomous agents in a real environment for a large number of such cases may not be practically feasible. The limitations and dangers of executing such systems in a real environment hinder testing many of the interesting test cases. Simulation environments such as SUMO [32], CARLA [11], Gazebo [29], OpenDS [33], and SVL[3] are proposed as safer and more efficient environments for executing tests for such systems. Testing by simulation has its challenges and disadvantages [4], but overall, it is an unavoidable prerequisite for physical and operational field tests. Considering simulation environments, in our work, we attempt to automatically generate and test different situations that are potentially critical. To this end, we take the test scenarios that are specified in an abstract way instead of concrete test scenarios. Our meaning of an abstract scenario is the scenario that can define different configurations of one general scenario. Such scenarios are used in our work to randomly generate different concrete critical case scenarios for testing. Currently, the considered feature of autonomous agents to test is narrowed down to the collision avoidance mechanism. Test generation for other features of autonomous agents is also discussed in the literature, like AsFault [21] that targets the lane-keeping feature of self-driving cars. Generating test suites for autonomous systems is considered extensively in the literature; below, we provide a survey of some of the closely related work.

For specifying high-level scenarios, we propose a DSL with formal semantics which considers the locality of autonomous agents. There are other DSLs in the literature for specifying scenarios for cyber-physical systems such as Scenic [19], OpenScenario [17], MDSL[4], and GeoScenario [41]. Compared to other DSLs, we opted for our formally-defined and minimalistic DSL focusing on the locality constraints for multiagent grid-based systems. Our design principle was to provide a confined DSL in order to be able to carefully investigate the effect of the parameters in the efficiency of test-case selection mechanisms. We expect future studies will be needed to replicate our results for more complex DSLs, but our results will provide a guideline for how such future studies should be organized. In fact, we are carefully extending our DSL in our ongoing research with features using the basic idea of separation of concerns and limiting interaction.

Two main methods exist at the opposite ends for generating test cases based on their required amount of computation: random testing [25] and constraint solving [34, 38]. On the one end, random testing spends negligible computational effort in generating test inputs but has high uncertainty in providing a desirable result and hence, poor productivity in test selection; on the other hand, constraint solving involves considerable computational effort but provides a guarantee for capturing

[2]https://www.euroncap.com/en

[3]https://www.svlsimulator.com

[4]https://www.foretellix.com/open-language

domain constraints, if at all possible. Search-based approaches [22] are placed between these two ends. They require more computation effort than the random input filtering approach and less computation effort than the full constraint solving approach (notwithstanding the fact that SMT solvers use many meta-heuristic search approaches under the hood). In this work, we compare the efficiency of the two extremes of this spectrum. Identifying the performance characteristics of these two approaches give us a clear idea of the type of parameter characteristics one needs to consider in order to choose a suitable approach. Such characteristics can also be utilized in devising efficient search-based algorithms, for example, by integrating these approaches. Such an integrated approach may be applicable when neither random filtering nor constraint solving has satisfactory performance (for example, when path lengths and arena size of the required test cases are large, see Table 10) later on. As mentioned before, search-based approaches are not entirely different from the constraint solving approach because optimization engines in contemporary SMT solvers are used to support the solving process. The optimization engine, like in Z3, can also be accessed directly and can be used for generating test cases when the required test specification is formulated as an optimization problem. In addition to search-based testing, there are other approaches that build upon random testing and constraint-based testing and improve their performance and effectiveness. Our study has targeted the baseline approaches, and these enhancements, briefly surveyed below, may be used in future empirical studies.

Due to effectiveness issues in random testing [35], several approaches are proposed to enhance it. Adaptive Random Testing (ART) is one such approach [25], which uses diversity to improve fault detection. Our method for generating random paths is based on earlier experiment to ensure some level of diversity in the generated paths [14] and is hence aligned with the goals of ART. Using a formal measure of diversity is likely to improve the results of random testing in our experiments and warrants further investigation.

Constraint solvers directly provide solutions for constraints, but using them for testing and verification has a few drawbacks and challenges. First, constraint solvers are not very scalable, and finding a solution for a large problem with complex constraints can be prohibitively time- and resource-consuming. Second, when generating a diverse set of test cases is preferred, it is commonly required to find varying solutions to one constraint. Sampling SAT solutions is referred to as Constraint Random Verification (CRV) [38] in the domain of hardware design and is also known as SAT witnesses in other domains. Spur [1], QuickSampler [12], Smarch [39], and UniGen2 [7, 8] are some of the state-of-the-art samplers, where Spur aims to address both scalability and uniformity [24]. In our work, we use the Z3 solver [34] for constraint solving and rely on its random seed to reach a solution for the given constraint. Currently, we investigate the time of reaching the first solution by the solver, and analyzing the diversity of solutions by repeating solver calls is left for future studies.

Simplifying constraints and checking for mutual constraint inclusion are two approaches to improve scalability of constraint solving [2, 26]. Thanks to the formal foundation of our approach, we can apply these techniques to our DSL and improve the performance of the constraint-solving based approach in future studies.

In test generation by filtering random test scenarios [25], test cases that do not satisfy the expected criterion are discarded. Test case prioritization [28] is a related technique where test cases are reordered for execution based on a criterion. In this approach, instead of discarding low-priority test cases, they are reordered to be executed after the ones with higher priority. This method is commonly used in regression testing where a test suite is already designed for testing the previous version of the system [23, 40]. Based on the feedback taken from previous test runs, the test cases are reordered to improve the efficiency from one point of view, for instance, to detect faults faster. In our work, the focus is on generating test cases where all generated ones have the same execution

priority. Prioritizing test cases after their generation can be considered as a future line of work as well.

4 METHODOLOGY

In this section, we provide an overview of the type of subject systems targeted by our study as well as the testing process used for them. Finally, we briefly introduce our approach to test selection, which is based on a domain-specific language.

4.1 Intended Subject Systems

The Systems Under Test (SUT) considered in our study are grid-based multiagent systems, often used in planning for robotic and autonomous systems [43]. Each agent has starting and goal coordinates and an initially planned path between them, including several imposed delaying steps (to simulate a varying speed or intermediate agent tasks). This initial plan is a pre-calculation only and disregards any future observations when the agent is moving in the environment. Agents may update their movement plan during operation to avoid collisions with others. Section 7.2 provides a more detailed account of the agents' plans and their updates as implemented in our SUT.

The input to this system is the grid size (X, Y) and initially planned paths, i.e., the sequences of waiting and displacement steps, for each agent. The output of this system is a sequence of actual moves of each agent to reach its goal. The test oracle considers possible collisions (i.e., more than one agent residing in the same cell). Any collision is an indication of a failure in the agent's safety mechanism, as the agents are supposed to avoid collisions even if there are collisions in the initially planned paths, as depicted in Fig. 1.

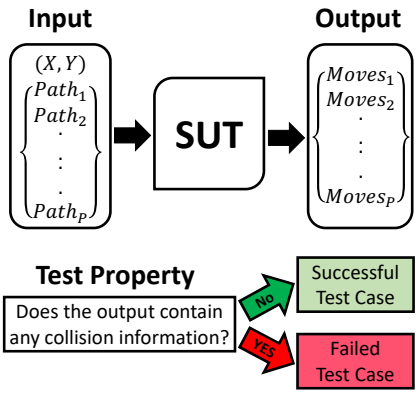


Fig. 1. The SUT of autonomous agents and the testing property

4.2 Test Process

To automate the testing process for this type of system, we aim to generate critical test cases, i.e., initially planned paths, that push the agents toward collisions. The ability of agents to deal with critical test cases by avoiding collisions provides more trust in the safety of the agents' control algorithm. For this, we first define the general structure of each test case. Then, using this structure and evaluating the parameters, we generate concrete random test cases. Since the agents can continuously revise their plan at run-time, the safety of the implementation cannot be tested by just analyzing the initial input. Thus, we need a discipline of dynamic testing to evaluate the safety

behavior of the SUT. Although this approach can reveal faults in unforeseen corner cases, a large number of test cases may be generated to include only a few effective, fault detecting, test cases. Executing all these test cases is prohibitively time-consuming, even in a simulation environment. Therefore, we propose exploiting the domain knowledge to distinguish effective tests and restrict test execution to the potentially effective ones. Such domain knowledge can also be utilized to measure test coverage in terms of critical scenarios and compare the reliability of different systems. For example, a system passing test cases of more challenging or more diverse types of scenarios can be considered more reliable than a system passing less challenging or fewer types of critical scenarios.

4.3 Test Selection

We propose a DSL to formalize (some aspects of) the domain knowledge. Having this DSL, complex testing scenarios can be easily specified by the composition of DSL elements. Our DSL can serve as a basis for future extensions capturing more domain elements, such as agents' dynamics and kinematics. In this paper, we consider using the DSL in two different ways for generating test inputs. In the first method, random test cases are generated first, and then the ones satisfying the domain constraint are filtered. In the second method, a constraint solver is used to derive test cases directly from the DSL.

5 TEST SELECTION DSL

In this section, we explain the syntax and semantics of our proposed DSL for specifying locality-based test selection constraints of autonomous agents in a grid-based multiagent system.

5.1 Syntax

The syntax of our DSL is shown in Mod. 2. A constraint in our DSL is either a locality-based condition specified about an area, or a Boolean expression built upon such conditions. A locality-based condition is of the form *"In Area Condition"*. An area can be either a circle or a square, where *"Circle x"* and *"Square x"* represent a circle and a square with radius and side length x, respectively. Conditions can, in turn, either be atomic conditions or a Boolean combination of conditions. There are two types of atomic conditions. *"Count d"* considers the condition of having at least d agents at one particular time in a pre-specified area. *"Intersection n d"* considers the condition of having at least n grid points in a pre-specified area that at least d agents cross sometime in their path. Different constraints can also be composed using Boolean operators of the DSL to make more complex constraints. As an example of a valid constraint specified by this DSL, *"In Circle* 1 *Count* 3*"* specifies a test input data that includes at least 3 agents residing at one time in a Circle with radius 1.

Module 2. DSL syntax for locality-based test selection constraint specification of autonomous agents

```
Constraint ->   In  Area Condition |
                And Constraint Constraint |
                Not Constraint |
                Or  Constraint Constraint

Area       ->   Circle Integer |
                Square Integer

Condition  ->   Count Integer |
                Intersection Integer Integer |
                And Condition Condition |
                Not Condition |
                Or  Condition Condition |
```

To illustrate the syntax, a few examples of constraints are shown below for the paths represented in Fig. 2. This example consists of the planned paths of four agents in a 7×7 grid. The agents start to move at the same time $t = 0$ and stop at the same time $t = 6$. Based on the traffic situation, the agents are supposed to autonomously adapt their actual moves while running and avoid possible collisions with the others if needed. Thus, the constraints always refer to the planned paths, and not to the actual paths.

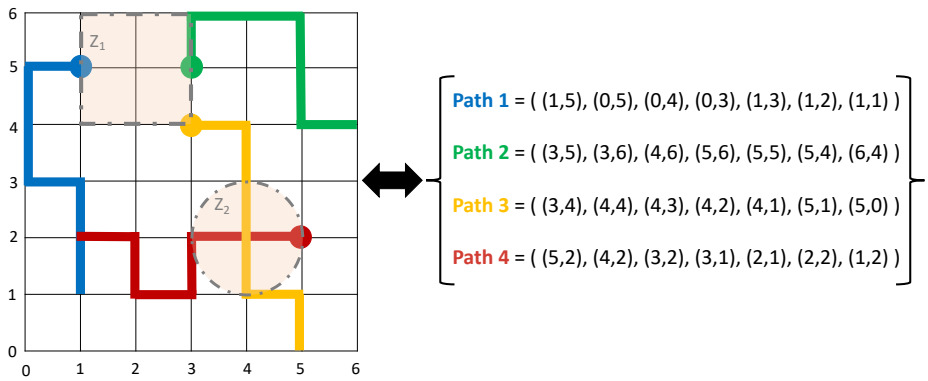


Fig. 2. Test input example including the planned paths of four autonomous agents

- **`In Square 2 Count 3`**: This constraint is satisfied for the test input since there are three agents (i.e., agents 1, 2, 3) that in a particular time ($t = 0$) stand in positions that are included in a square with side length 2 (the Z_1 area).
- **`In Circle 1 Intersection 1 2`**: This constraint is satisfied for the test input since there is an occurrence of two agents (agents 3 and 4) crossing a particular point (i.e., point $(4, 2)$) which is included in a circle with radius 1 (the Z_2 area). In fact, for this condition, the area defined in the constraint is effectively irrelevant (this condition occurs in a single point that is included in any area).
- **`In Square 2 (Intersection 1 2 And Count 3)`**: This constraint is not satisfied for the test input since there is no square area with side length 2 in which both conditions

"`Intersection 1 2`" and "`Count 3`" are satisfied. An area almost satisfying this constraint would be the square defined by $(1, 2)$–$(3, 4)$ corners; however, the three agents present in this square are not present there at the same time.

5.2 Semantics

For defining the DSL semantics, we use the internal data types that are shown in Mod. 3. The type `Point` refers to a grid point made of an integer tuple, and `Path` refers to a sequence of several adjacent or repetitive `Points` in the grid. `AreaInstance` refers to one particular instance of an area type in the grid containing its center point (currently, we only consider circle and square area types in the DSL that have a grid aligned center). `CheckCase` is for holding the required information for checking the satisfaction of a condition; the one starting with `CasesC` contains the `Count` condition related information, and `CasesI` contains the `Intersection` related one.

Module 3. DSL internal data types

```
Point        -> (Integer, Integer)
Path         -> [Point]
AreaInstance -> AreaIC Area (Real, Real)
CheckCase    -> CasesC [[Point]] | CasesI [Point] Integer
```

The proposed DSL semantics, defined in Def. (1)–(4), assumes a $\mathcal{G} \times \mathcal{G}$ grid containing $\mathcal{M}$ agents where each agent has a total of $\mathcal{L}$ number of movement steps (including the imposed waiting steps). The evaluation of a constraint for a given list of paths is defined by the *eval* function in Def. (1). This function uses the auxiliary function *evalCon* defined in Def. (2). Based on the condition type, *evalCon* exploits *getCases* function in order to extract the desired information from the given paths, defined in Def. (3). According to this information, the existence of some features in a given area is checked by the function *areaContains*, defined in Def. (4). The Composition of *Constraint*s and *Condition*s by Boolean operators is defined in Def. (1) and (2).

$$
\begin{array}{l}
eval :: Constraint \rightarrow [Path] \rightarrow Boolean \\
eval\ (\textbf{\textit{In}}\ e\ c\ ) \quad P := \exists x, y \in \mathbb{R}.\ 1 \leq x, y \leq \mathcal{G}.\ evalCon\ c\ (\textbf{\textit{AreaIC}}\ e\ (x\ , y))\ P \\
eval\ (\textbf{\textit{Not}}\ f) \quad P := \neg eval(f\ P) \\
eval\ (f_1\ \textbf{\textit{And}}\ f_2)\ P := (eval\ f_1\ P) \wedge (eval\ f_1\ P) \\
eval\ (f_1\ \textbf{\textit{Or}}\ f_2) \quad P := (eval\ f_1\ P) \vee (eval\ f_1\ P)
\end{array}
\tag{1}
$$

$$
\begin{array}{l}
evalCon :: Condition \rightarrow AreaInstance \rightarrow [Path] \rightarrow Boolean \\
evalCon\ (\textbf{\textit{Count}}\ d) \quad a\ P := \exists z \in (getCases\ (\textbf{\textit{Count}}\ d)\ P).\ areaContains\ a\ z \\
evalCon\ (\textbf{\textit{Intersection}}\ n\ d)\ a\ P := \exists z \in (getCases\ (\textbf{\textit{Intersection}}\ n\ d)\ P).\ areaContains\ a\ z \\
evalCon\ (\textbf{\textit{Not}}\ c) \quad a\ P := \neg\ evalCon(c\ a\ P) \\
evalCon\ (c_1\ \textbf{\textit{And}}\ c_2) \quad a\ P := \exists z \in (getCases\ c_1\ P).\ (areaContains\ a\ z)\ \wedge (evalCon\ c_2\ a\ P) \\
evalCon\ (c_1\ \textbf{\textit{Or}}\ c_2) \quad a\ P := \exists z \in (getCases\ c_1\ P).\ (areaContains\ a\ z)\ \vee (evalCon\ c_2\ a\ P)
\end{array}
\tag{2}
$$

$$
\begin{array}{l}
getCases :: Condition \rightarrow [Path] \rightarrow [CheckCase] \\
getCases\ (\textbf{\textit{Count}}\ n) \qquad P := (\textbf{\textit{CasesC}}\ S)\ where \\
\quad S = \{\ s \mid t \in \{0,\ldots,\boldsymbol{\mathcal{L}}\},\ Q = \{P[1][t],\ldots,P[\boldsymbol{\mathcal{M}}][t]\},\ s \subseteq 2^Q,\ |s| = n\ \} \\
getCases\ (\textbf{\textit{Intersection}}\ n\ d)\ P := (\textbf{\textit{CasesI}}\ S\ n)\ where \\
\quad S = \{(x,y) \mid \exists Q \subseteq P.\ |Q| = d,\ \forall q \in Q.\ \exists i \in \{1,\ldots,\boldsymbol{\mathcal{M}}\}.\ \exists t \in \{1,\ldots,\boldsymbol{\mathcal{L}}\}.\ (x,y) = q[i][t]\ \}
\end{array}
\tag{3}
$$

$$
\begin{array}{l}
areaContains :: AreaInstance \rightarrow CheckCase \rightarrow Boolean \\
areaContains\ (\textbf{\textit{AreaIC}}\ (\textbf{\textit{Circle}}\ r)\ c)\ (\textbf{\textit{CasesC}}\ S) \quad := \exists Q \in S. \\
\quad \forall q \in Q.\ (q_x - c_x)^2 + (q_y - c_y)^2 \leq r^2 \\
areaContains\ (\textbf{\textit{AreaIC}}\ (\textbf{\textit{Circle}}\ r)\ c)\ (\textbf{\textit{CasesI}}\ S\ n) \quad := \exists Q \subseteq S.\ |Q| = n \\
\quad \forall q \in Q.\ (q_x - c_x)^2 + (q_y - c_y)^2 \leq r^2 \\
areaContains\ (\textbf{\textit{AreaIC}}\ (\textbf{\textit{Square}}\ r)\ c)\ (\textbf{\textit{CasesC}}\ S) \quad := \exists Q \in S. \\
\quad \forall q \in Q.\ |q_x - c_x| \leq \frac{r}{2}\ \wedge\ |q_y - c_y| \leq \frac{r}{2} \\
areaContains\ (\textbf{\textit{AreaIC}}\ (\textbf{\textit{Square}}\ r)\ c)\ (\textbf{\textit{CasesI}}\ S\ n) := \exists Q \subseteq S.\ |Q| = n \\
\quad \forall q \in Q.\ |q_x - c_x| \leq \frac{r}{2}\ \wedge\ |q_y - c_y| \leq \frac{r}{2}
\end{array}
\tag{4}
$$

6 SELECTIVE RANDOM TEST CASE GENERATION

In this section, we present two methods of exploiting the domain knowledge in generating effective test cases. In the remainder of the paper, these methods are compared in a rigorous fashion.

6.1 Filtering Randomly Generated Data

In the first step of this approach, test cases are generated randomly. Then, the required test selection constraint is applied for each generated test case to check whether it satisfies the required criteria. If the constraint is satisfied, the test case will be used in test execution, and otherwise, it will be discarded.

For testing our SUT, random paths can be generated in different ways [14]. In this paper, for generating a random path with displacement length d, we first pick one random starting point and one candidate random endpoint in the grid that are reachable from one another with at most d horizontal and vertical moves. Then, we generate a random set of horizontal and vertical moves to reach the candidate end point from the starting point with the minimum number of moves m. If this path includes less than d moves, it will be compensated by adding random pairs of {*Left*, *Right*} and/or {*Up*, *Down*} to the path. In this approach, if either d or m is even and the other is odd, adding random pairs of {*Left*, *Right*} and/or {*Up*, *Down*} to a path with length m will not generate a path with length d (it will have the length $d-1$ or $d+1$). In other words, the generated path reaches the adjacent points of the candidate end point from the starting point in this case. This issue is simply resolved by randomly selecting an adjacent point of the candidate endpoint as the target endpoint of the generating path, which is d moves far from the chosen starting point. In case of having mandatory waiting moves with length w, w *wait* actions are added to random positions in the generated path. Finally, all moves in the path are shuffled at the end to add more randomness.

This method of randomly generated paths, which is illustrated in Fig 3, is called *Targeted Data Generation* in [14].

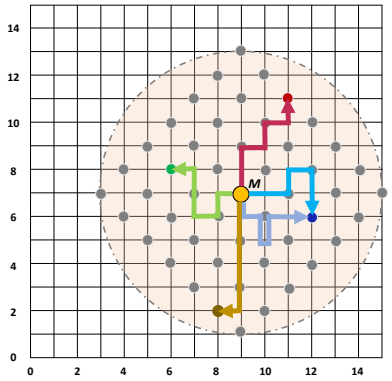


Fig. 3. The possible end points and some random paths with length 6 starting from point M

In our forthcoming experiments, Erlang and QuickCheck are used for generating random paths, filtering them based on the given constraint, and finally executing the tests for the selected ones on the target SUT, as shown in Mod. 4. This test specification written in Erlang takes the grid size, number of agents, number of displacement and waiting steps, and test selection constraints (specified by our DSL) as input. Then, for each list of randomly generated paths (in line 3), it checks if the required constraint is satisfied (in line 4). For the paths that satisfy the constraint, the test is executed, and the existence of collisions is checked afterwards.

Module 4. QuickCheck module for testing the SUT of autonomous agents

```
1  testCollision(X,Y,AgentsNum, ActionSteps, WaitSteps, Constraint)->
2    ?FORALL(AgentsPaths,
3            pathGenerator(X, Y, AgentsNum, ActionSteps, WaitSteps),
4            ?IMPLIES(eval(Constraint, AgentsPaths),
5                     begin
6                       Trace = sut:run(AgentsPaths, X, Y),
7                       not anyCollision(Trace)
8                     end)).
```

6.2 Constraint Solving

In this approach, a constraint solver is used for generating paths that satisfy the constraint specified in our DSL. To generate paths, the path-construction constraints (i.e., the adjacency of the points in the parth) are defined in the input language of the solver. Similarly, inter-path constraints stemming from our DSL are also translated into the input language of the solver, based on our formal semantics.

Z3 is a state-of-the-art constraint solver that we use in this work. We mechanized the translation of our DSL to Z3 constraints format in the following way. We define four classes of variables X, Y, D and W for each agent i at each time t of their movement to store the following information:

- $X_{i,t}$: the X position of the agent i in time t in the grid
- $Y_{i,t}$: the Y position of the agent i in time t in the grid

- $D_{i,t}$: the number of passed displacement moves of the agent i up to time t
- $W_{i,t}$: the number of passed waiting moves of the agent i up to time t

To generate simple paths (with no inter-path constraints) for agents, we specify the Z3 constraints based on these variables. To begin with, we define that the position (X, Y) of each agent is always bounded to the grid size $\boldsymbol{G}$. Similarly, we define that W and D for each agent are always equal or greater than zero and less or equal to the required number of waiting and displacement steps. In the beginning, W and D are both zero for each agent, and at the end, they are equal to the given required number of steps. The constraints of the agents' movement are specified by defining that at each time, either W or D must be incremented (with respect to the previous time), and the other must remain intact.

The translation from our formal semantics to the input language of Z3 is straightforward. For example, to translate the constraint of "In Circle 2 Count 3", we define new variables C_x and C_y as the center of a circle and specify that (C_x, C_y) are bounded in the grid. Then, we define that the position of at least 3 of the previously declared agents at a time is inside the circle with radius 2 and center (C_x, C_y).

It has been shown that diversifying test suites improves the fault detection ability, even in test suites with small sizes [15]. We considered two ways of building some diversity into our path generation. First, in some constraint solvers, solving a set of constraints always leads to one particular solution, i.e., the solving process is deterministic. As a result, reaching diverse test cases can be a challenge with some solvers. To have diversified solutions, one can first call the solver to reach the first solution. Supposing that in the first solution the value of variables $V_1, V_2, \ldots, V_n$ are determined to be $c_1, c_2, \ldots, c_n$, respectively, to reach a different solution in the next attempt, the constraint $(V_1 \neq c_1) \vee (V_2 \neq c_2) \vee \cdots \vee (V_n \neq c_n)$ can be added to the solver. One could strengthen this further by replacing inequality to a stronger notion of diversity. Secondly, in this work, we use the Z3 solver and rely on its internal mechanism for the diversifying solutions. This is achieved by letting Z3 choose a random seed for each run.

7 EXPERIMENT DESIGN

We design and conduct experiments in this section to investigate answers to the following research questions defined in Section 1:

RQ1: Can random generation and filtering test cases make fault detection more efficient in grid-based multiagent systems?

RQ2: Can random generation and filtering test cases lead to a more efficient process for finding the most concise failing test case in grid-based multiagent systems?

RQ3: How does test case generation efficiency by random generation and filtering compare with test case generation by constraint solving in grid-based multiagent systems?

RQ4: How do problem domain and constraint complexity influence test case generation time with either of the two methods in grid-based multiagent systems?

For analyzing the experiment results, we use statistical hypothesis testing. In this approach, a *null hypothesis*, which is represented by H_0, and its opposite, *alternative hypothesis* which is represented by H_a, are defined first. Then, acceptance of the null hypothesis (and rejection of the alternative) or vice versa would be evaluated by considering a particular confidence level of the corresponding statistical test. If the confidence level of rejecting a null hypothesis exceeds a specified threshold, the alternative hypothesis will be accepted (and the null hypothesis will be rejected). Otherwise, the alternative one would be rejected (and the null hypothesis would be accepted). We will accept an alternative hypothesis in this paper if the confidence level of accepting it is greater than 95%, i.e., the p-value is less than 0.05.

In the remainder of this section, we first provide a more detailed account of the subject systems and then explain the two experiments designed to answer our research questions.

7.1 Subject System

The SUT in our experiment is called *SafeTurtles* [13] which is a program containing several autonomous agents. In SafeTurtles, there are a fixed number of agents, called *turtles*,[5] that move in a two-dimensional grid, i.e., the possible movement directions are up, down, left, and right. A movement in each direction is allowed only if it does not push the agent beyond the grid boundaries. All turtles are able to stay at their current points or move to an adjacent point, and the movement speed of all turtles is the same. Each turtle has an identifier (a number), a starting point, a goal point, and an initially planned path between these two points. In the beginning, all turtles are situated outside the arena. Upon launch, the turtles try to occupy their starting positions and move toward their goal positions. After reaching the goal position, the turtle goes out of the arena again. All turtles have full environmental observability and are aware of the current positions of all others. In addition, through communicating with each other, the turtles get aware of the planned immediate next move of all others too, including the ones that may potentially collide with them in their next move. Each turtle evaluates this information before every move, revises its plan if needed to avoid possible collisions, and moves one step forward according to the (potentially revised) plan. This is notwithstanding the faults that are injected to evaluate and compare the fault detection capabilities of different approaches explained below.

All turtles should follow two safety rules to avoid collisions. First, for the next step, no turtle is allowed to move to a position that is occupied by another turtle in the current step. Second, the turtles in the neighboring cells should synchronize such that if more than one turtle plans to move to the same position, the turtle with the smallest identifier has the highest priority to go there. If executing the current plan of a turtle violates these safety rules, that turtle is supposed to update its movement plan. Plan update for each turtle starts with choosing one random position among the possible safe positions to move to in the next step. The safe positions for each turtle consist of the turtle's current position and all adjacent positions that are currently not occupied and that no turtle with a higher priority wants to occupy in the next step. To complete updating the movement plan after picking the next move, the turtle randomly chooses one shortest path to reach its goal position from there (the shortest path between two points in a grid can be built with different combinations of the required horizontal and vertical moves).

7.2 Experiment I

This experiment is designed to evaluate the effect of applying filters on randomly generated test cases to answer our research questions **RQ1** and **RQ2**. To do that, we test our simple SUT of autonomous agents that includes a few injected faults. The injected faults affect the agent's movement decisions and actions, which is representative of real fault types of multiagent systems. However, from the perspective of complexity, the faults are simple (due to having a simple SUT), and they have a higher occurrence rate than the faults of realistic multiagent systems. For testing our SUT, we use QuickCheck for random generation of inputs with and without test selection filters. For generating random paths, we implement the *targeted data generator* explained in Section 6.1. In the shrinking process, we attempt to shorten the agent's paths by changing their goal positions, while keeping their initial points intact.

[5]The name is a legacy of our earlier work when we experimented with an implementation of autonomous turtles in the Robot Operating System.

We analyze testing efficiency in both fault detection and failed test case shrinking processes. Our analysis is mainly based on the observed number of SUT executions up to detecting the SUT fault, which is the most resource- and time-consuming task of our testing. Additionally, we measure fault detection efficiency by counting the total number test steps (agent moves) before the first fault is detected in the test suite: for test cases that do not uncover any fault, this is the maximum number of steps taken by any agent in the test case, while for failing test-cases, it is the maximum path lengths of the agents that collided. This criterion is a more specific indication of the required timing cost or the size of the test suite used for detecting faults in our SUT. Therefore, we also apply similar statistical tests to compare the results based on this criterion to answer **RQ1**.

We constructed a symbolic model of the safety rules and a correct implementation for agents respecting the safety rules to avoid collisions. To validate the correctness of this implementation, we tested the system with 10,000 random test cases, and the agents reached their goals as expected, with no observed collision or deadlock. Subsequently, we injected the SUT with three types of faults inspired by actual faults. To decide on the fault types, we interviewed two lecturers of the graduate course Design of Embedded and Intelligent Systems (DEIS) at Halmstad University [45] about the common faults of autonomous robots designed in a project akin to SafeTurtles by students in the course. The lecturers have been responsible for this course for the last five years and have supervised about 5 graduate-student projects on average each year. Three recurring fault types were reported: (i) self-localization faults leading to incorrect estimates of the position of the robot in the environment due to the accumulated sensor errors, (ii) faults leading to incorrect actions due to actuation mistakes and miscalibration (effectively over-speeding and/or over-braking), and (iii) perception and communication faults leading to incorrect information about other robots and their movement plans.

We injected these fault types into *SafeTurtles* as follows. For the self-localization fault, we allow the agent to assume itself in a position that is adjacent to its actual position. For the actuation fault, we allow the turtle to overshoot a movement by one additional position going in the same direction or move one position when the turtle is supposed to stay put. The third injected fault disables synchronization on the plans of another neighboring agent. In the faulty version of the SUT, there is an independent and uniform distribution of the probability of each fault type or no fault at all (i.e., 25% fault probability) during the execution of each turtle move.

This experiment is designed for the following parameter sizes and filtering constraints. We also repeat testing the SUT with and without each filter 100 times to get better statistics.

- **Grid sizes**: $\{10 \times 10, 15 \times 15, 20 \times 20, 50 \times 50\}$
- **Number of agents**: $\{5\}$
- **Path length**:
  Maximum displacement steps: $\{5\}$
  Maximum wait steps: $\{5\}$
- **Test selection constraints**:
  $\boldsymbol{F_1}$: In Circle 3 Count 2
  $\boldsymbol{F_2}$: In Circle 1 Count 2
  $\boldsymbol{F_3}$: In Circle 1 Intersection 1 2

In defining the experiment sizes, we are considering two issues. First, we would like to see the effect of filtering on fault detection. In our case, the fault happens in a situation when the agents are getting close to each other and trying to visit a point at the same time. Therefore, if the grid size is increased and the other parameters are kept fixed, the probability of collision will decrease. Similarly, decreasing the number of displacement steps or decreasing the number of agents each

has the same effect. This provides us with the above-given design space for our experiment by varying the grid size, path length, and agent number.

Our second consideration in designing this experiment is to see the effect of the strictness of a constraint. We call a constraint less strict than another one if the accepted test cases by the stricter constraint are a subset of the less strict one. For this purpose, we added $\boldsymbol{F_1}$ filter to this experiment, which is less strict than the filter $\boldsymbol{F_2}$. We would also like to observe how both *Count* and *Intersection* constraint condition types affect the result. For this purpose, we specified $\boldsymbol{F_3}$ filter along with the filters $\bar{\boldsymbol{F_1}}$ and $\bar{\boldsymbol{F_2}}$. It should be restated that we are not looking for the best filter that detects faults faster in our SUT or all considered multiagent systems. The effectiveness of a filter essentially relies on the domain knowledge of test engineers. Here, filters $\boldsymbol{F_1}$, $\boldsymbol{F_2}$, and $\boldsymbol{F_3}$ are examples of filters with different complexities. It would be an interesting future research problem to find a correlation between the effectiveness of filters and the fault types.

For our experiment, we produced four data sets: one without a filter and three others for filters $\boldsymbol{F_1}$ to $\boldsymbol{F_3}$. These are referred to as $\boldsymbol{D_0}$, $\boldsymbol{D_1}$, $\boldsymbol{D_2}$, and $\boldsymbol{D_3}$, respectively. We compare the result of applying no filter with each and every filter by applying the corresponding statistical tests. To pick a suitable statistical test, we need to check if the data sets are normally distributed. To check the normality of the data set distribution, the *Shapiro-Wilk* statistical test [44] is the most suitable choice. The null and alternative hypotheses of this test applied to some data set D are defined in Statement (5).

$$\begin{aligned} \boldsymbol{H_{0_1}} &: D \text{ is distributed normally.} \\ \boldsymbol{H_{a_1}} &: D \text{ is not distributed normally.} \end{aligned} \tag{5}$$

Once the normality of the data sets is determined, we can pairwise compare the data sets. First, however, we can check if at least one of the data sets has significantly different values than one of the other ones. If no significant difference is detected, there is no need to put further effort into comparing the pairs. The statistical tests used for that are the following. *Anova* [16] and *Kruskal-Wallis* [31] are the statistical tests that check if the mean of one data set among a group of data sets is significantly different than another one. Annova is suitable when all data sets in the group are normally distributed, and Kruskal-Wallis can be applied regardless of the data sets' distribution. The null and alternative hypotheses of checking a significantly different data set in the group are defined in Statement (6). We apply Anova test if $\boldsymbol{H_{0_1}}$ is accepted for all $\boldsymbol{D_i}$-s (all $\boldsymbol{D_i}$ are normally distributed) and apply Kruskal-Wallis test otherwise.

$$\begin{aligned} \boldsymbol{H_{0_2}} &: \forall i, j \in \{0,1,2,3\}.\ \mu(\boldsymbol{D_i}) = \mu(\boldsymbol{D_j}) \\ \boldsymbol{H_{a_2}} &: \exists i, j \in \{0,1,2,3\}.\ \mu(\boldsymbol{D_i}) \neq \mu(\boldsymbol{D_j}) \end{aligned} \tag{6}$$

In this test, the acceptance of $\boldsymbol{H_{a_2}}$ means that the mean value of one data set is significantly different than one other data set. However, it would not clarify which one is greater than the other. Since we are interested in comparing the filtered results with the non-filtered one, when $\boldsymbol{H_{a_2}}$ is accepted, we need further investigation by pair-wise comparison of the corresponding data sets. In other words, by accepting$\boldsymbol{H_{a_2}}$, we need to pair-wise compare $\boldsymbol{D_0}$, $\boldsymbol{D_1}$, $\boldsymbol{D_2}$, and $\boldsymbol{D_3}$. However, if $\boldsymbol{H_{0_2}}$ is accepted instead, we conclude that none of the filters significantly affect the testing result compared to having no filter at all, and we avoid making any further comparisons.

For pair-wise statistical comparison, the following are the suitable tests. The *t-test* [46] and *Mann-Whitney-Wilcoxon u-test* [48] check if the mean values of two data sets are significantly different from each other. The t-test is suitable when both data sets are normally distributed, and the other test can be applied regardless of the data sets' distribution. In our experiment, when $\boldsymbol{H_{0_1}}$ is

accepted for two of our data sets (they are distributed normally), we apply t-test for their pair-wise comparison and apply Mann-Whitney-Wilcoxon u-test otherwise.

We apply the statistical test with the details defined in Statement (7) to compare these data sets with each other. This test is *one-tailed* and when $\boldsymbol{H_{a_3}}$ is rejected, either $\mu(\boldsymbol{D_0}) = \mu(\boldsymbol{D_f})$ or $\mu(\boldsymbol{D_0}) < \mu(\boldsymbol{D_f})$ are possible to be accepted. In order to clarify that, we apply another one-tailed pair-wise test with the details defined in Statement (8).

$$\begin{aligned} \boldsymbol{H_{0_3}} &: \mu(\boldsymbol{D_0}) \leq \mu(\boldsymbol{D_f}) \\ \boldsymbol{H_{a_3}} &: \mu(\boldsymbol{D_0}) > \mu(\boldsymbol{D_f}) \end{aligned} \tag{7}$$

$$\begin{aligned} \boldsymbol{H_{0_4}} &: \mu(\boldsymbol{D_0}) \geq \mu(\boldsymbol{D_f}) \\ \boldsymbol{H_{a_4}} &: \mu(\boldsymbol{D_0}) < \mu(\boldsymbol{D_f}) \end{aligned} \tag{8}$$

Along with the statistical analysis, we also want to calculate the average fault detection time for different grid sizes supposing different SUT execution times, i.e., the practical performance gain from our improvements. To do that, we observe the time of filtering one test case by our filters in 100 filtering attempts. Then, assuming negligible time for random test data generation, we use the formula in Def. (9) and (10) to calculate the average filtering and fault detection time based on the average value of the other parameters.

$$\textit{Filtering Time} := (|\textit{Executed cases}| + |\textit{Discarded Cases}|) \times (\textit{Unit Filtering Time}) \tag{9}$$

$$\textit{Fault Detection Time} := \textit{Filtering Time} + |\textit{Executed cases}| \times (\textit{SUT Execution Time}) \tag{10}$$

7.3 Experiment II

This experiment is designed to evaluate and compare the efficiency of constraint solving versus random input filtering as means to generate test data. Specifically, this experiment addresses our research questions **RQ3** and **RQ4**. In this experiment, the time of generating a valid test input for different domain parameters is calculated for both test data generation approaches. Similar to the previous experiment, QuickCheck is used for generating random test cases and their filtering, and Z3 is used for solving constraints. For generating random test cases, the targeted data generator (explained in Section 6.1) is implemented. To get a diverse set of solutions by Z3, we use its internal strategy for diversification using random seeds.

We monitor the performance of both methods for generating a valid test case for the values of the following parameters:

- **Grid sizes**: $\{10 \times 10, 15 \times 15, 20 \times 20, 50 \times 50\}$
- **Number of agents**: $\{10, 15, 20\}$
- **Path lengths**:
  Displacement steps:$\{1, 5, 10, 15, 20\}$
  Wait steps:$\{0\}$
- **Test selection constraints**:
  "In Square 2 Count x", where x $\in \{3, 5, 8\}$
  "In Square 1 Intersection 1 x", where x $\in \{3, 5, 8\}$
- **Time-out (in seconds)**: $\{15, 30, 60\}$

In designing this experiment, we aim to compare the performance of the two methods by changing the parameters of the design space specified above. For different path lengths, we could vary the length of both displacement and wait steps. However, since varying wait steps has an

effect similar to varying the number of displacement steps, we decided to fix the number of wait steps (to zero) in the experiment design. We use the constraints in our experiment with the form of "In Square 2 Count x" and "In Square 1 Intersection 1 x" (note again, that for the condition of the form "Intersection 1 x", the area of the constraint is irrelevant since the single intersection of agents' paths occurs in a single point which is included in any area). In addition, because the time of generating a valid test case can be prohibitively long, we need to impose a time-out for each method for generating a valid test case. For the sake of completeness, we apply different time-outs of 15, 30, and 60 seconds to judge whether the running time can affect the results. Moreover, since we are using a randomness factor in both of our methods, test case generation time for a single constraint and test configuration can vary in different attempts. For example, in 30 attempts of generating a single test case in a test set-up (grid size: 10×10, number of agents: 15, displacement steps: 1, wait steps: 0, time-out: 60 seconds, constraint: 'In Square 2 Count 3'), the mean and standard deviation of the generation times that we observed were 34.61 and 23.59, respectively. Therefore, to gain statistical confidence, we repeat calculating the performance of each method for each experiment configuration 30 times. In this repetition, we also consider that Z3's solution for one set of constraints can possibly be cached. Therefore, we avoid successive calls to Z3 for the same experiment configuration. Specifically, between two calls of Z3 for the same configuration, we call Z3 for all other experiment configurations once.

To figure out the correlation between one test scenario parameter and the corresponding test generation time, we apply a statistical correlation test to measure the linear association between these two factors. *Pearson* [18] and *Spearman* [9] methods can be used for this. The first method is suitable when both data sets are normally distributed, and the second one can be applied regardless of the data set's distribution. Since all test scenario parameters are defined uniformly by us, we know that one side of the correlation test is not distributed normally. Thus, we use the Spearman method for correlation testing. Naming test generation time with $\boldsymbol{T}$ and test scenario parameter with $\boldsymbol{P}$, the null and alternative hypotheses of this test are defined in Def (11).

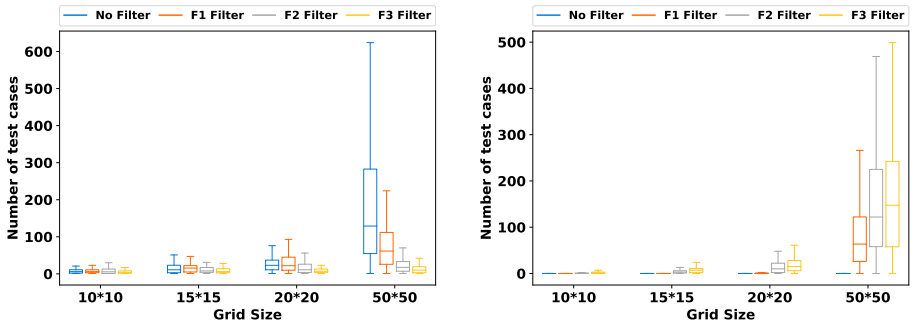
$$\begin{aligned} \boldsymbol{H_{0_5}} &: \boldsymbol{T} \text{ and } \boldsymbol{P} \text{ do not have a linear correlation.} \\ \boldsymbol{H_{a_5}} &: \boldsymbol{T} \text{ and } \boldsymbol{P} \text{ have a linear correlation.} \end{aligned} \tag{11}$$

We also calculate the value of $\boldsymbol{r}$ in this test, which is the linear coefficient value between test generation time and a test scenario parameter. This coefficient value is a number in the range $[-1, +1]$, where $+1$ implies a very strong direct correlation, 0 implies no association, and -1 implies a very strong inverse correlation between the two variables.

8 EXPERIMENT RESULTS

8.1 Experiment I Results

8.1.1 Fault detection time. In the case of having filters, the total fault detection time comprises path generation, checking the filtering constraint satisfaction, and test execution. Since the SUT execution time is significantly large in our domain, the number of executions has a significant contribution to the total fault detection time. In case of having no filter, the test generation time is negligible and filtering time is zero, but all generated test cases before failure are executed on the SUT for detecting the fault. Test filters can reduce the total fault detection time by reducing the number of test executions. The number of executed and discarded test cases in our experiment are shown in Fig. 4. In Fig. 4a, we see that as the grid size increases, the number of SUT executions to catch the fault increases for each filtering case. This is simply because increasing the grid size while fixing the other parameters decreases the possibility of randomly generating a critical scenario to



(a) Executed cases (b) Discarded cases

Fig. 4. The number of executed (accepted) and discarded (rejected) test cases up to detecting a failure in SafeTurtles

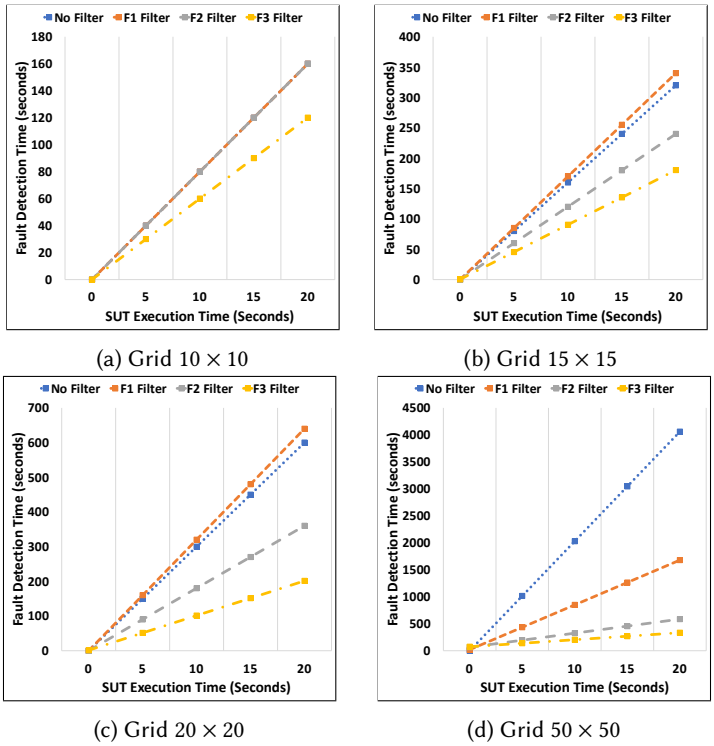
Table 1. The p-value of applying hypothesis tests on the required number of test executions

	Grid 10 × 10				Grid 15 × 15				Grid 20 × 20				Grid 50 × 50			
	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}
No Filter	0.00	0.34	—		0.00	0.00	—		0.00	0.00	—		0.00	0.00	—	
F1 Filter	0.00		-	-	0.00		0.82	0.17	0.00		0.70	0.29	0.00		**0.01**	1
F2 Filter	0.00		-	-	0.00		0.07	0.92	0.00		**0.00**	0.99	0.00		**0.00**	1
F3 Filter	0.00		-	-	0.00		**0.00**	0.99	0.00		**0.00**	1	0.00		**0.00**	1

catch a fault. To measure the effect of filtering on the number of test executions, statistical tests are applied to the data, and the results are summarized in Table 1.

According to Table 1, when the grid size is 10×10, there is no significant difference in the number of test executions by the applied filters (H_{a_2} does not have an acceptable confidence level and is rejected). In this case, the grid is small enough, and the intended critical scenarios are generated with high probability. As a result, random test cases are effective enough in detecting the SUT fault even with no help of the proposed filters. However, by increasing the grid size, a significant difference is detected in all other three grid sizes (H_{a_2} is accepted for them). In 15×15 grid, ***F3*** significantly reduces the number of test executions (H_{a_3} is accepted for ***F3***). However, the other two filters do not lead to a significant improvement in this grid size (H_{a_3} is rejected for both of ***F1*** and ***F2***). In other words, ***F3*** is guiding the random test cases towards our SUT fault in a 15×15 grid with higher effectiveness than the other two filters. By increasing the grid size to 20×20, both ***F2*** and ***F3*** result in a significant reduction of the number of test executions (H_{a_3} is accepted for both), but ***F1*** which has a less strict constraint than ***F2*** does not improve the efficiency of random test cases significantly. However, in the 50×50 grid, all three filters show a significant improvement in the number of test executions (H_{a_3} is accepted for all three filters). Increasing the grid size increases the room for randomly generated paths to diverge from each other, leading to less effective test scenarios. In this case, even small guidance to the generated inputs can significantly improve effectiveness, as we can see in the results for the 50×50 grid.

The fact that ***F3*** is effective in smaller grid sizes can be explained as follows. For a small arena and small path sizes, an intersection constraint is likely to lead to a possible collision with relatively large number of agents. This explains why in our experiment, we see better performance for ***F3*** compared to ***F1*** and ***F2*** in revealing SUT faults in smaller grid sizes.



(a) Grid 10 × 10

(b) Grid 15 × 15

(c) Grid 20 × 20

(d) Grid 50 × 50

Fig. 5. Average fault detection time

Table 2. The p-value of applying hypothesis tests on the required number of execution steps till detecting a fault

	Grid 10 × 10				**Grid 15 × 15**				**Grid 20 × 20**				**Grid 50 × 50**			
	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}
No Filter	0.00	0.33	—		0.00	0.00	—		0.00	0.00	—		0.00	0.00	—	
F1 Filter	0.00		-	-	0.00		0.80	0.19	0.00		0.70	0.29	0.00		**0.01**	1
F2 Filter	0.00		-	-	0.00		0.06	0.93	0.00		**0.00**	0.99	0.00		**0.00**	1
F3 Filter	0.00		-	-	0.00		**0.00**	0.99	0.00		**0.00**	1	0.00		**0.00**	1

To make a more precise quantitative and platform-independent estimate of the fault detection time, we also used the total number of execution steps (until failure) in SUT executions for comparing fault detection efficiency in our experiments. Then, we apply similar statistical tests and present the results in Table 2. Here, we see that the results are very similar to the ones in Table 1 in all filtering cases and all grid sizes. A similar p-value analysis explained for the results of Table 1 can be restated for the results of Table 2. The results of Table 2 also confirm that our initial assumption regarding the similarity of SUT execution time for generated test cases with and without filters is valid, and the number of test executions is a good proxy to compare testing efficiency in our experiment.

To answer **RQ1**, in addition to the number of test cases, and the total number of steps, we also calculate the average fault detection time with and without filters. Fig. 5 shows how average fault detection time is affected by having different test execution times (assuming a fixed time for each

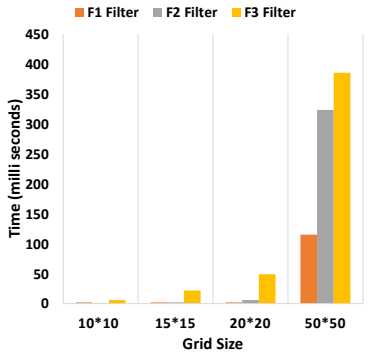


Fig. 6. Average time of filtering a test case

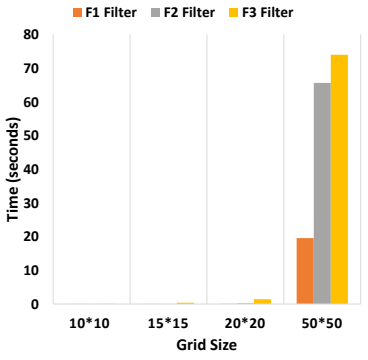
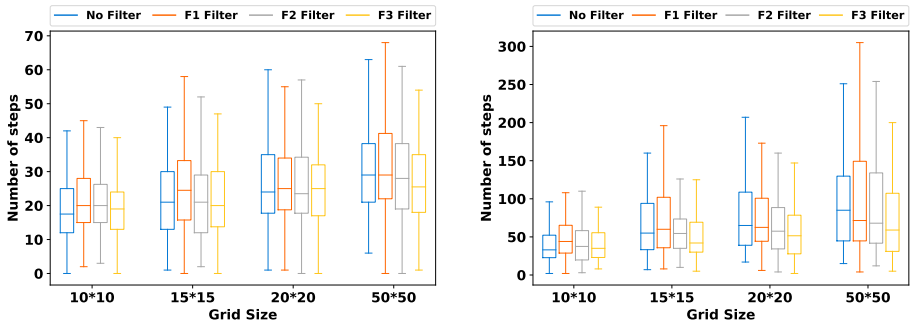


Fig. 7. Average time of filtering test cases up to detecting a failure in SafeTurtles

test case execution and using the average value of all other parameters defined in Def. (9) and (10) in the required calculations). As mentioned in Def. (10), fault detection time has a linear relation with SUT execution time. Applying a test selection filter is an attempt to reduce the coefficient of this linear relation in exchange for filtering cost. In other words, filtering would be favorable to utilize with random test cases as long as (i) the filters effectively guide test cases towards challenging scenarios, which leads to reducing the required number of test executions for detecting faults, and (ii) SUT execution time is significantly larger than the time of filtering a test case. In our test selection scenarios, when the grid size is 10×10, the number of executions will not be significantly affected by our filters (see Table 1). However, by increasing the grid size, we see the role of different filters, especially ***F3***, in guiding the test cases toward potentially faulty cases. It results in reducing the number of test executions and, as a result, reducing the fault detection time. The anticipated improvement in the efficiency by having filters depends on SUT execution time. If SUT is executed very fast, the filtering time may not be compensated by reducing the number of test executions. But as long as the SUT execution time is large, the effect of having filters and reducing the number of test executions is shown with higher clarity. In more realistic scenarios, following Def. (10), a similar linear relation is expected between SUT execution time and fault detection time. However, as much as (i) input generators generate more faulty cases, (ii) test selection filters detect more faulty cases, (iii) the filtering cost is lower, and (iv) SUT execution is more time-consuming, we would see a linear filtering plot with a smaller slope and a smaller vertical-intercept.

Providing the filtering cost overhead of our experiment can complement the results shown in Fig. 4. As mentioned in Def. (9), filtering has the cost of checking the satisfaction of the filtering constraint for all generated test cases. To estimate the filtering time overhead in our experiment, first, we monitored the number of accepted (executed) and rejected (discarded) test cases, represented in Fig. 4. Then, we observed the time of filtering one test case in 100 filtering attempts, represented on average in Fig. 6. This figure shows that, on average, filtering a test case by ***F1*** takes less time than ***F2***, and it takes less time than ***F3***. It also shows that increasing the grid size raises the filtering time, which happens due to the increased searching time in a bigger space. Based on the average time of filtering a test case and the average number of generated test cases up to detecting the fault, which is shown in Fig. 4a, the average filtering time of fault detection can be calculated (according to the Def. 9), which is represented in Fig. 7.

8.1.2 Shrinking time. In the shrinking process of QuickCheck (see Section 2.1), the same filter that is used in the test selection phase is also applied in the shrinking process. Filtering constraints



(a) Successful attempts (b) Failed attempts

Fig. 8. The number of shrinking attempts

Table 3. The p-value of applying hypothesis tests on the number of successful shrink attempts

	Grid 10 × 10				Grid 15 × 15				Grid 20 × 20				Grid 50 × 50			
	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}
No Filter	0.00		—		0.00		—		0.00		—		0.00		—	
F1 Filter	0.00	0.16	-	-	0.00	0.16	-	-	0.00	0.74	-	-	0.00	0.32	-	-
F2 Filter	0.01		-	-	0.00		-	-	0.03		-	-	0.35		-	-
F3 Filter	0.00		-	-	0.22		-	-	0.05		-	-	0.03		-	-

Table 4. The p-value of applying hypothesis tests on the number of failed shrink attempts

	Grid 10 × 10				Grid 15 × 15				Grid 20 × 20				Grid 50 × 50			
	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}	H_{a_1}	H_{a_2}	H_{a_3}	H_{a_4}
No Filter	0.00		—		0.00		—		0.00		—		0.00		—	
F1 Filter	0.00	0.09	-	-	0.00	0.02	0.60	0.39	0.00	0.01	0.60	0.39	0.00	0.00	**0.01**	1
F2 Filter	0.00		-	-	0.00		0.21	0.79	0.00		0.09	0.90	0.01		**0.00**	1
F3 Filter	0.00		-	-	0.00		**0.00**	0.99	0.00		**0.00**	0.99	0.00		**0.00**	1

prune the search space of the possible shrunk test case candidates. Filters can potentially contribute to making the shrinking process more efficient, because they filter out those intermediate tests that are unlikely to cause a failure and hence, reduce the number of failed attempts while shrinking. Additionally, the initial test cases that pass the filtering phase are likely to be better starting points for the shriknking process, than randomly generated inputs. On the other hand, an ineffective constraint that filters out the most shrunk test input will mislead the shrinking process to continue its search with other test inputs, which may further affect the number of failed and successful shrinking attempts. Hence, we hypothesize that proper filtering constraints that recognize the cases that do not potentially reveal the SUT faults, may reduce the number of failed shrinking attempts.

The number of successful and failed shrinking attempts in our experiment are shown in Fig. 8. To check if the existence of filters can reduce the number of successful and failed shrinking attempts, we applied statistical test on the results, shown in Tables 3 and 4.

According to Table 3, the p-value of H_{a_2} is greater than 0.05 in all of the grid sizes, and, as a result, H_{a_2} is rejected for all of them. It means that filtering does not make a significant impact on the number of successful shrink attempts in all of the grid sizes. However, we observe a different picture by looking at Table 4. According to Table 4, no significant difference is seen in a 10×10 grid;

i.e., for the number of failed shrinking attempts H_{a_2} is rejected. Similarly, no significant changes are seen by having ***F*1** and ***F*2** filters in 15×15 and 20×20 grids, but ***F*3** shows good performance and significantly reduces the number of failed shrinking attempts in these grid sizes. In the 50×50 grid, all three filters reduce the number of failed shrinking attempts significantly.

Since we do not have access to the source code of the shrinking heuristics in QuickCheck, our discussion of the results is speculative and based on our intuition. We believe the insignificance of changes in the number of successful shrinking steps can be explained by the fact that filtering does not directly guide QuickCheck on how to choose successful attempts in the shrinking process to reach the most shrunk failed test case. Filters are mostly used to discard the test cases from test execution to begin with, when their possibility of revealing the SUT faults is considered to be low.

This explanation is consistent with the positive effect of filtering on reducing the number of failed shrinking steps. When the arena size is not big, the shrinking process is likely to detect faults even with no help from filters. The number of failed shrinking attempts is typically small in this grid size, and the effect of filtering is not significant as a result. When the possibility of detecting the fault in a modified test case is low, the modified paths in the shrinking process will have more chance to stray away and fail to detect the SUT fault. In such cases, even small hints can result in a considerable benefit. This is the reason that we see more significant results for larger arena sizes. In this experiment, we also see a better performance for ***F*3** rather than ***F*2** in reducing the number of failed shrink attempts. This happens because the test cases that ***F*3** accepts have more potential for detecting a fault of our SUT than ***F*2** (and ***F*2** more than ***F*1**). Therefore, there would be a higher probability with ***F*3** to reach a failed test case rather than the other two. As a result, among the modified inputs to consider in the shrinking process, ***F*3** will accept a smaller portion of those inputs to execute to reach the most shrunk test case. This leads to a better performance in the number of failed shrink attempts for ***F*3** rather than ***F*2**.

8.2 Experiment II Results

Tables 5 and 6 show the average time of generating a valid test case for a test selection constraint of the form "In Square 1 Intersection 1 X" with 60 seconds time-out. It can be seen in Table 5 that increasing grid size and constraint strictness increases the test generation time of the filtering approach. However, increasing path length and the number of agents decreases the test case generation time of this approach. On the other hand, as shown in Table 6, constraint solving looks robust to the changes of grid size and the constraint strictness, but increasing the number of agents and path length increases its test case generation time. Thus, the two approaches have significantly different characteristics.

Tables 7 and 8 show the average time of generating a valid test case by QuickCheck and Z3 for test selection constraint of the form "In Square 2 Count X" with 60 seconds time-out. As presented in Table 7, grid size and constraint strictness have a direct correlation with the test generation time of the filtering approach. A smaller number of agents sometimes degrade the performance, and it looks like path length does not considerably affect this approach's performance. On the other hand, as shown in Table 8, increasing the number of agents and path length directly degrade the performance of the constraint solving approach. However, it looks as if the grid size and constraint strictness do not affect the performance of this approach significantly.

In order to find out the correlation between test scenario parameters and test generation time, we apply statistical correlation test to check if there is a linear association between both sides; this is shown in Tables 9 and 10. In the case of constraint solving, the results show with a good confidence level (the p-values are below 0.05) that, for both constraint types 'Count' and 'Intersection', the parameters path length and the number of agents have a direct correlation with the test generation time. It also shows that for both constraint types, the path length has a higher impact on test

Table 5. The average random input filtering times (by QuickCheck) to generate a valid test input with 60 seconds time-out for the constraints of the form “In Square 1 Intersection 1 $\boldsymbol{X}$"

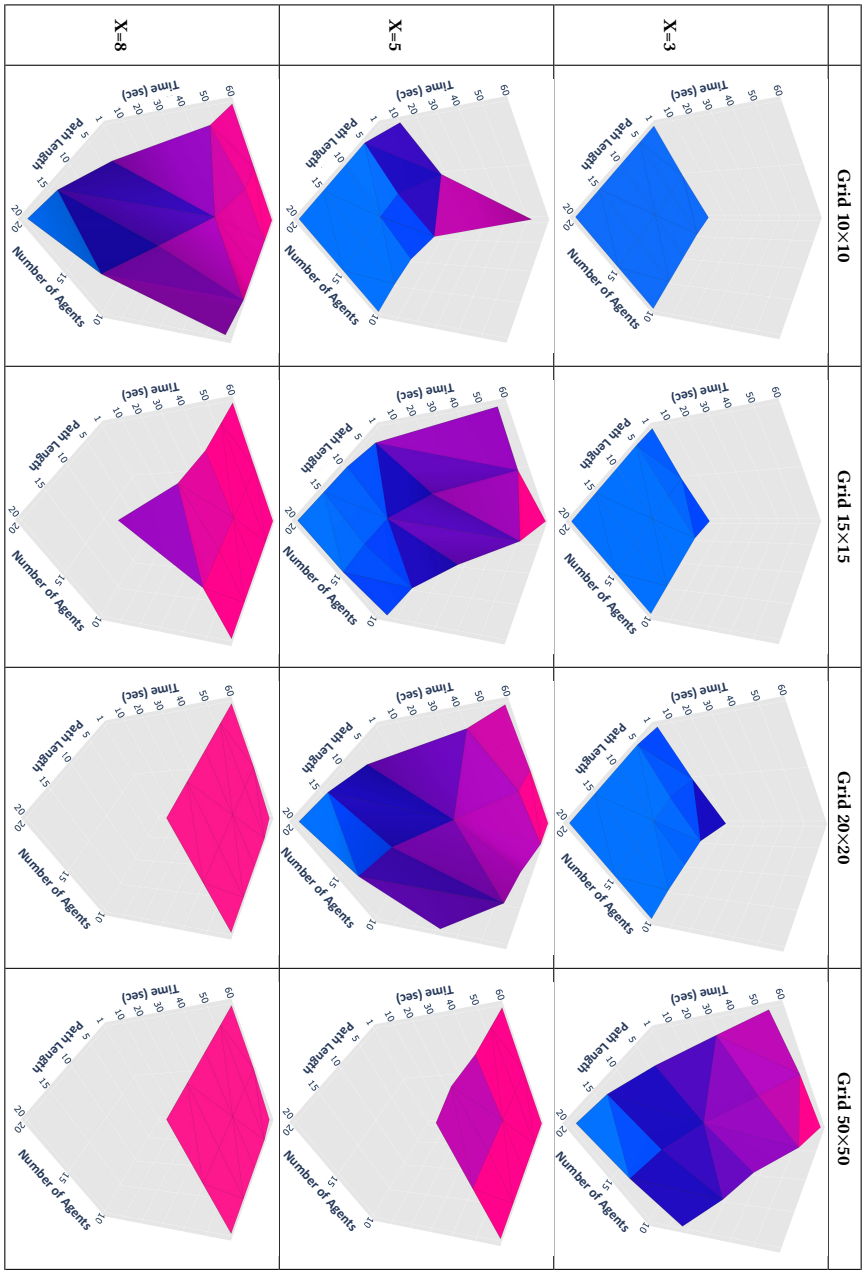


Table 6. The average constraint solving times (by Z3) to generate a valid test input with 60 seconds time-out for the constraints of the form "In Square 1 Intersection 1 X"

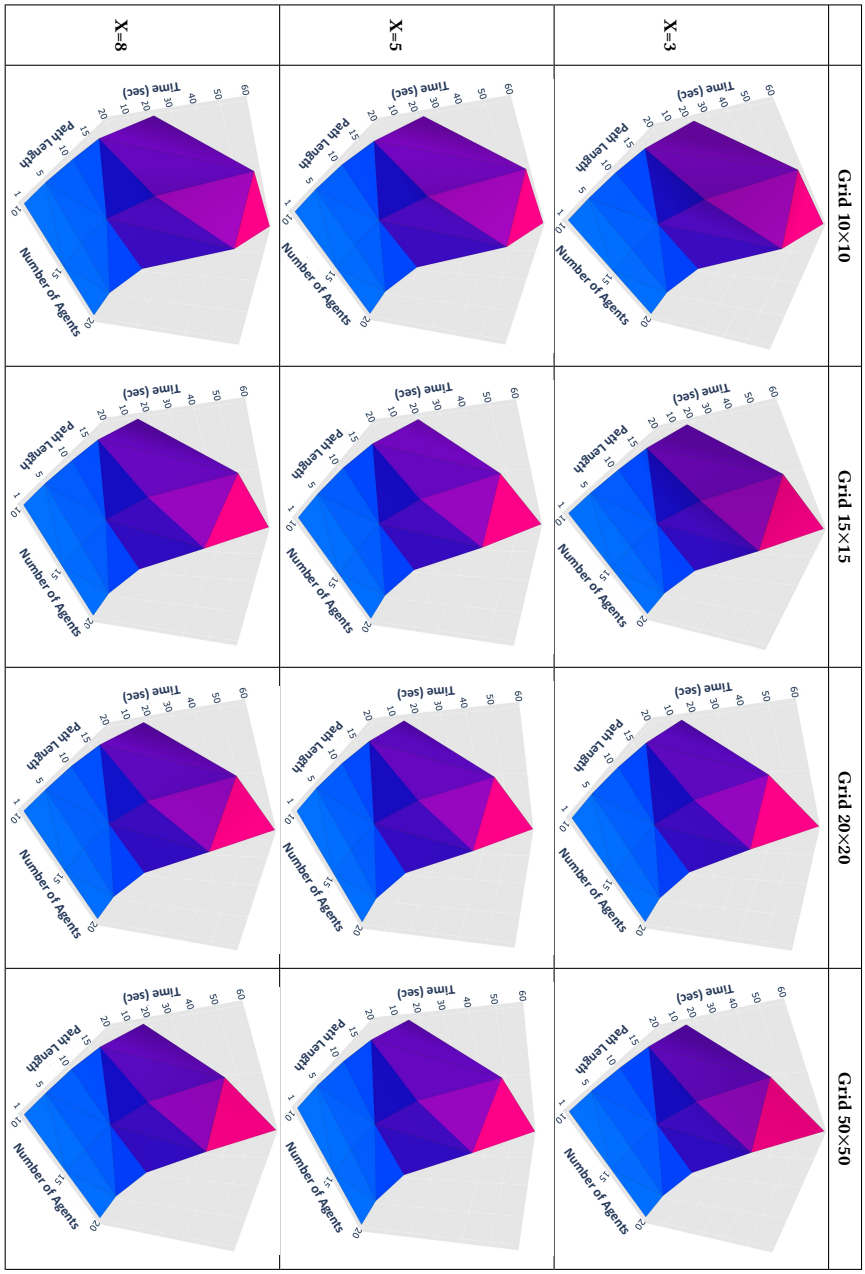


Table 7. The average random input filtering times (by QuickCheck) to generate a valid test input with 60 seconds time-out for the constraints of the form “In Square 2 Count X"

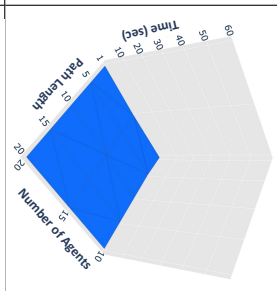
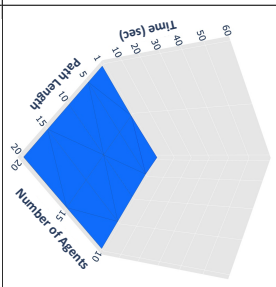
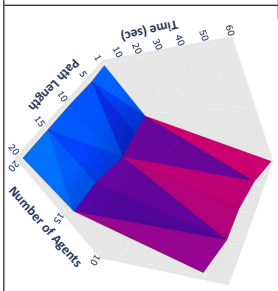
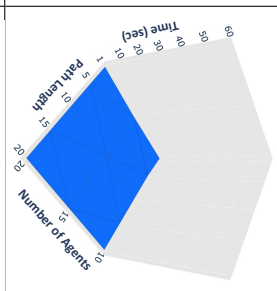
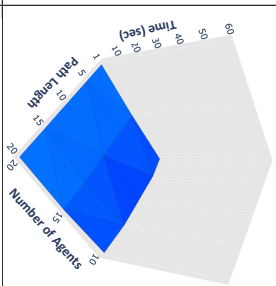
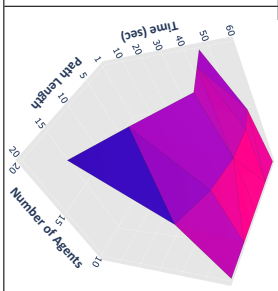
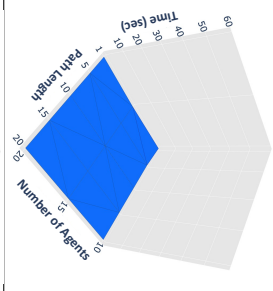
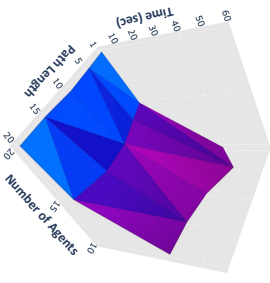
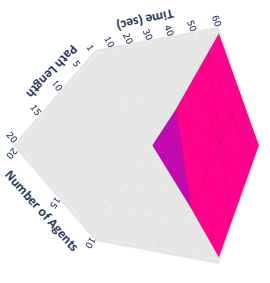
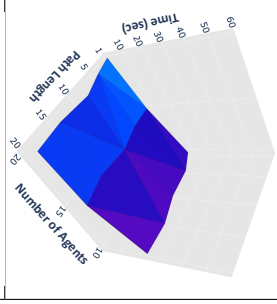
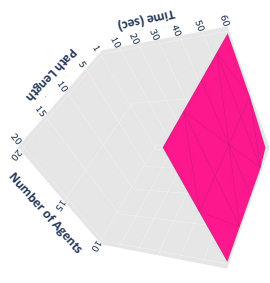
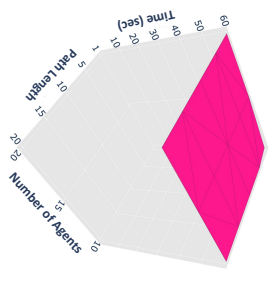
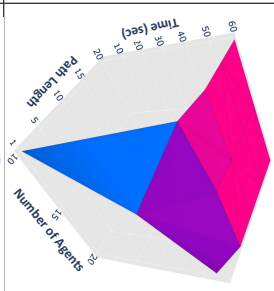
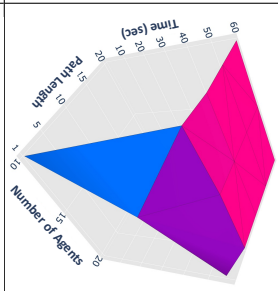
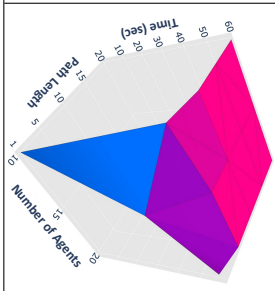
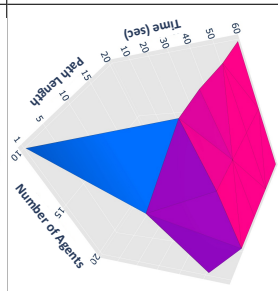
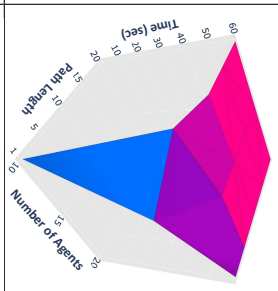
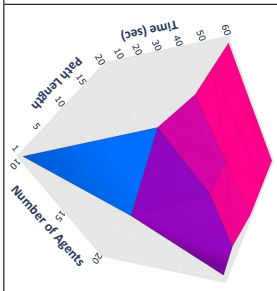
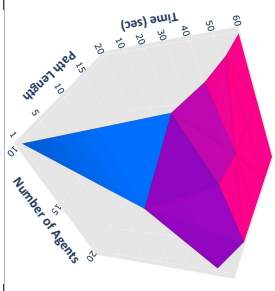
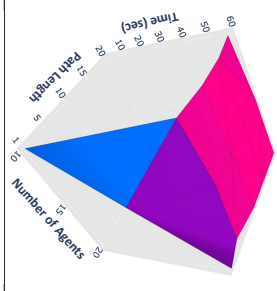
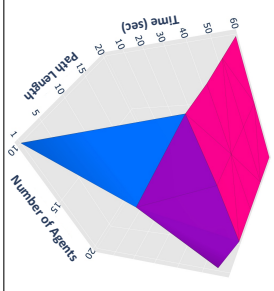
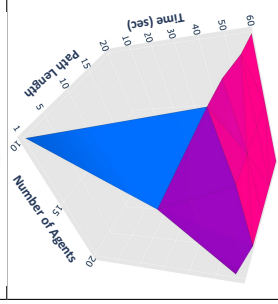
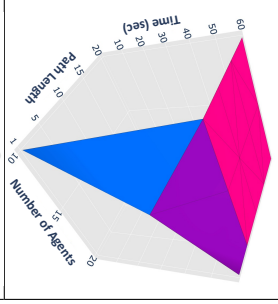
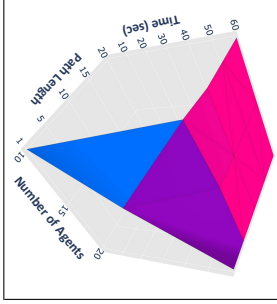
	X=3	X=5	X=8
Grid 10×10			
Grid 15×15			
Grid 20×20			
Grid 50×50			

Table 8. The average constraint solving times (by Z3) to generate a valid test input with 60 seconds time-out for the constraints of the form "In Square 2 Count X"

	X=3	X=5	X=8
Grid 10×10			
Grid 15×15			
Grid 20×20			
Grid 50×50			

generation time than the number of agents (the r factor is higher for the path length). On the other hand, for both constraint types, the results show that the grid size and constraint strictness do have a slight, but not high, impact on the test generation time. For the sake of completeness, we have repeated all experiments for time-out values of 15- and 30 seconds. The results, which are rigorously analyzed next, are very similar for different time-outs, which shows the running time does not considerably affect the results of constraint solving, i.e., for our class of constraints, if the problem is not solvable in a shorter time, it is unlikely to be solvable in a longer time.

The performance of Z3 is affected by the number of its input constraints. In our case, increasing path length and number of agents increase the number of constraints and, as a result, degrades the performance of Z3 in reaching a solution. However, changing the grid size only changes the domain of some variables, which does not considerably affect the performance. Changing the constraint strictness results in a change in a Z3 function input, which does not lead to a significant change in its performance either.

According to Tables 9 and 10, in case of random test case filtering, grid size and constraint strictness directly affect the test generation time, with an acceptable confidence level for both constraint types (the p-values are below 0.05). The results remain invariable for different time-out values of 15, 30, and 60 seconds. In random input filtering, constraint strictness has more impact on the performance than the grid size (the r factor is higher for the constraint strictness). An inverse correlation between the number of agents and test generation time is also indicated in random input filtering. However, the confidence level of this correlation for both constraint types is not high. For the 'Intersection' constraint type, an inverse correlation is shown between path length and time, where its impact is less than grid size and constraint strictness. However, the inverse correlation between the time and path length is not significant for the 'Count' constraint type in random input filtering. Similar to constraint solving, these results are very similar for different time-outs, which shows the running time does not considerably affect the results of random input filtering. This result is not surprising; when the probability of satisfying the test selection criterion through random selection is very low, increasing the number of attempts 2 or 4 times, will not increase this probability as much.

Two factors affect the performance of the random input filtering approach: the number of discarded test cases, and the required computation to check the satisfaction of a constraint. In our case, increasing each of the four test scenario parameters increases the problem size and, as a result, the required computation. The number of discarded test cases depends on the possibility of random paths in satisfying the constraint. This possibility is increased by tailoring random path generation definition (see Section 6.1) for the aimed filtering constraints. In the targeted path generator used in this experiment, increasing the grid size reduces the possibility of satisfying our constraints. Increasing our constraints' strictness also reduces this possibility. On the other hand, increasing path length and number of agents increases this possibility. However, by increasing the number of agents, the trade-off with the computation time growth does not allow to have a significant performance improvement in test generation. A similar effect is indicated by increasing the path length in the case of having the 'Count' constraint type. However, in the case of the 'Intersection' constraint type, the positive effect of increasing path length on the performance overcomes the negative effect of the increased computation. This is the reason that we see a significant inverse correlation between path length and test case generation time in Table 9 for the random input filtering approach.

9 THREATS TO THE VALIDITY

In this work, we conducted our experiments on an SUT of autonomous agents that is an abstraction of a realistic multiagent system. The injected faults are artificial but are representative of real

Table 9. The linear coefficient value of and p-value of applying correlation test for the constraint of the form "In Square 1 Intersection 1 X"

	Constraint Solving (Z3)						Random Input Filtering (QuickCheck)					
	Time-out 15 seconds		Time-out 30 seconds		Time-out 60 seconds		Time-out 15 seconds		Time-out 30 seconds		Time-out 60 seconds	
	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}
Path Length	**0.93**	0.00	**0.93**	0.00	**0.94**	0.00	**-0.25**	0.00	**-0.23**	0.00	**-0.22**	0.00
Number of Agents	**0.25**	0.00	**0.26**	0.00	**0.26**	0.00	-0.13	0.06	-0.14	0.05	-0.12	0.09
Grid Size	-0.02	0.73	-0.02	0.70	-0.03	0.65	**0.46**	0.00	**0.45**	0.00	**0.46**	0.00
Constraint Strictness X	-0.02	0.69	-0.02	0.77	-0.00	0.91	**0.71**	0.00	**0.73**	0.00	**0.75**	0.00

Table 10. The linear coefficient value and p-value of applying correlation test for the constraint of the form "In Square 2 Count X"

	Constraint Solving (Z3)						Random Input Filtering (QuickCheck)					
	Time-out 15 seconds		Time-out 30 seconds		Time-out 60 seconds		Time-out 15 seconds		Time-out 30 seconds		Time-out 60 seconds	
	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}	r	H_{a_5}
Path Length	**0.59**	0.00	**0.65**	0.00	**0.68**	0.00	-0.00	0.93	-0.00	0.93	-0.01	0.80
Number of Agents	**0.27**	0.00	**0.33**	0.00	**0.27**	0.00	-0.14	0.05	-0.13	0.06	-0.13	0.06
Grid Size	0.00	0.97	0.00	0.94	0.02	0.74	**0.55**	0.00	**0.56**	0.00	**0.58**	0.00
Constraint Strictness X	0.06	0.40	0.05	0.46	0.04	0.50	**0.70**	0.00	**0.70**	0.00	**0.69**	0.00

systems and real faults in multiagent systems. Our experiments were based on a single-fault assumption, i.e., the occurrence of one fault in our experiment excludes the occurrence of the other faults at that time. Extending the experimental setup to a multiple-fault assumption, with independent random variables for each fault, is a possible generalization of our results. More research and experiments can be done to mitigate the threat of the generalizability of our fault model by analyzing fault types and frequencies of faults in other real-world robotic projects.

The proposed DSL for test selection specification only captures the basic actions of a grid-based multiagent system. This is a setting that is rich enough to compare filtering vs. constraint-based test selection; moreover, we represent the complexity of constraints by the size of the formulae representing them. We expect that the results will transfer to more complex DSLs since for other types of constraints, SMT solvers are likely to face similar issues with large formulae. This assumption may pose a threat to the generalizability of our results. To address these threats, we are currently adding other domain concepts into our DSL. Our early results indicate that this abstraction is suitable for complex multiagent systems, and also, the complexity of the constraints seems to have similar effects as those observed in the current paper.

Our extended DSL is inspired by model-based agent and environment abstractions provided by Russel and Norvig [43]. Using this extended DSL, we would be able to specify different test oracles and test selection constraints on the target environment and agents. For instance, we can define different sets of valid actions in different locations, a minimum safe distance between agents, or measure the severity of collisions in the test oracles. For environmental constraints, we can specify, for example, different assumptions about the observability of the environment and stochastic changes in the environment. Along with specifying inter-agent constraints, we consider the dynamics and kinematics of the agents in the new DSL. We can specify the agents' configurations with properties such as maximum speed, acceleration, and deceleration. We plan to

provide abstractions for more complex movement plans, such as visiting a set or a sequence of sub-goals. We will provide further information and analysis about our extended DSL in a forthcoming paper once its design and implementation are completed.

For generating random paths, we define our specific data generator and use it along with a handful of filtering constraints. Therefore, our experiment results cannot be generalized to other random path generators and filters. We would like to consider a wider range of data generators and filtering constraints and study the relative effect of them in our future work.

In our experiments, we used a particular range of values for the experimental parameters. We defined these ranges to see the trend of performance changes based on input parameter changes. Therefore, the experiment results cannot be generalized to the parameter values much smaller or larger than our conducted experiments. We also used time-outs in our second experiment due to the limited available resources (even with time-out, conducting the experiments takes about 26 days). Experimenting with other, longer, timeouts would give us a broader perspective on the results, and it would improve the accuracy of statistical analysis. In addition, for each parameter in our experiment, we just picked coarse-grained values from the considered range of that parameter. Certainly, doing the experiment with higher parameter resolution (ideally all the values in the range) would provide a clearer picture of the results, but the timing limitations persuaded us to make that decision in the experiment design.

For evaluating the performance of constraint solving and random input filtering approaches, we rely on the performance of our experiment tools QuickCheck and Z3. Each tool has its own configurations that can be optimized and affect the final result to some extent. This threat can be addressed by conducting the same experiments with other tools to improve the reliability of the results. There can be some code optimization threats in our implementation too. Although we tried our best in coding, there might be room for performance improvement (for example, by using different functions of Z3) in our implementation that influences the efficiency of our code and the experiment's results.

10 CONCLUSIONS

In this paper, we designed a DSL with formal semantics for capturing the domain knowledge in test case specification for grid-based multiagent systems. We used this DSL as a means for comparing two test case generation techniques, random test case generation with filtering versus test case generation by constraint solving. While both approaches have a promise of making testing more effective, they show distinct characteristics with respect to the parameters of the system under test.

In our experiments, we observe that the grid size and constraint strictness (in the right order) increase test case generation time for the filtered random data approach. On the contrary, these parameters do not seem to severely affect the effectiveness of the constraint solving approach. On the other hand, the number of agents and path length increase test case generation time for the constraint solving approach while they have some insignificant positive effect on the filtered random data approach. Our results suggest a clear complementarity of the two approaches based on the problem parameters and call for follow-up research on how to combine the two techniques in a suitable way.

As an immediate next step, we would like to scale up our case study toward our demonstrator within the SafeSamrt project. We can use the existing Robot Operating System (ROS) version of our case study that has a more elaborate decision making algorithm of agents or use Apollo[6], which is an open-source (ROS-based) system of autonomous agents for this purpose. Furthermore, we can use the simulation environment of Apollo, or SUMO/Veins simulations of communicating vehicles

[6]https://github.com/ApolloAuto/apollo

(V2X) for simulating our methods. In order to do that, we plan to extend our DSL to generate test cases for more realistic systems. The DSL could also consider the severity and likelihood of the undesired situations along with the possibility of failures. On the implementation level, we would also consider when to use constraint solving or random input filtering approaches, or when and how to combine them, to generate test cases efficiently based on the lessons we learned from this work. A complete implementation for the testing framework of QuickCheck would then follow.

ACKNOWLEDGMENTS

First and foremost, we would like to thank Thomas Arts for his contributions to the first instance of this work [14] and his continued support for our work. We thank the anonymous reviewers of ACM TOSEM for their constructive reviews. Our research has been partially funded by the Knowledge Foundation (KKS) in the framework of "Safety of Connected Intelligent Vehicles in Smart Cities – SafeSmart" project (2019–2023). Mohammad Reza Mousavi has been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2.

REFERENCES

[1] Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. 2018. Fast sampling of perfectly uniform satisfying assignments. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 135–147.

[2] Andrea Aquino, Francesco A Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. 2015. Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 305–315.

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. 2–10.

[4] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. 2021. Digital Twins Are Not Monozygotic – Cross-Replicating ADAS Testing in Two Industry-Grade Automotive Simulators. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 383–393.

[5] Oliver Carsten, Natasha Merat, Wiel Janssen, Emma Johansson, Mark Fowkes, and Karel Brookhuis. 2005. Human machine interaction and safety of traffic in Europe. *HASTE final Report* 3 (2005).

[6] Francesco Cesarini and Simon Thompson. 2009. *Erlang programming: a concurrent approach to software development*. O'Reilly Media, Inc.

[7] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On parallel scalable uniform SAT witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 304–319.

[8] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2014. Balancing scalability and uniformity in SAT witness generator. In *2014 51st acm/edac/ieee design automation conference (dac)*. IEEE, 1–6.

[9] Wayne W. Daniel. 1990. Spearman Rank Correlation Coefficient Applied Nonparametric Statistics. *Ed* 2 (1990), 358–365.

[10] John Derrick, Neil Walkinshaw, Thomas Arts, Clara Benac Earle, Francesco Cesarini, Lars-Åke Fredlund, Víctor M. Gulías, John Hughes, and Simon J. Thompson. 2009. Property-Based Testing - The ProTest Project. In *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6286)*, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel (Eds.). Springer, 250–271. https://doi.org/10.1007/978-3-642-17071-3_13

[11] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[12] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 549–559.

[13] Sina Entekhabi and Thomas Arts. 2023. SafeSmartTurtle. https://doi.org/10.5281/zenodo.8208127

[14] Sina Entekhabi, Wojciech Mostowski, Mohammad Reza Mousavi, and Thomas Arts. 2022. Locality-Based Test Selection for Autonomous Agents. In *ICTSS 2021: Testing Software and Systems*. Springer International Publishing, Cham, 73–89.

[15] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 223–233.

[16] Ronald A. Fisher. 1919. XV.—The Correlation between Relatives on the Supposition of Mendelian Inheritance. *Transactions of the Royal Society of Edinburgh* 52, 2 (1919), 399–433. https://doi.org/10.1017/S0080456800012163

[17] Association for Standardization of Automation and Measuring Systems. 2022. *ASAM OpenSCENARIO v2.0.0.* ASAM e.v.

[18] David Freedman, Robert Pisani, and Roger Purves. 2007. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York* (2007).

[19] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–78.

[20] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Automatically reconstructing car crashes from police reports for testing self-driving cars. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 290–291.

[21] Alessio Gambi, Marc Müller, and Gordon Fraser. 2019. Asfault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 27–30.

[22] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–12.

[23] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 523–534.

[24] Ruben Heradio, David Fernández-Amorós, José A Galindo, and David Benavides. 2020. Uniform and scalable SAT-sampling for configurable systems. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*. 1–11.

[25] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A survey on adaptive random testing. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2052–2083.

[26] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 177–187.

[27] Bengt Jonsson, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer. https://doi.org/10.1007/b137241

[28] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93.

[29] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. IEEE, 2149–2154.

[30] Friedrich Kruber, Jonas Wurst, and Michael Botsch. 2018. An unsupervised random forest clustering technique for automatic traffic scenario categorization. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2811–2818.

[31] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[32] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. 2018. Microscopic Traffic Simulation using SUMO, In The 21st IEEE International Conference on Intelligent Transportation Systems. *IEEE Intelligent Transportation Systems Conference (ITSC)*. https://elib.dlr.de/124092/

[33] Rafael Math, Angela Mahr, Mohammad M Moniri, and Christian Müller. 2013. OpenDS: A new open-source driving simulator for research. *GMM-Fachbericht-AmE 2013* 2 (2013).

[34] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[35] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.

[36] Wassim G Najm, Raja Ranganathan, Gowrishankar Srinivasan, John D Smith, Samuel Toma, Elizabeth Swanson, August Burgett, et al. 2013. *Description of light-vehicle pre-crash scenarios for safety applications based on vehicle-to-vehicle communications*. Technical Report. United States. National Highway Traffic Safety Administration.

[37] Wassim G Najm, Samuel Toma, John Brewer, et al. 2013. *Depiction of priority light-vehicle pre-crash scenarios for safety applications based on vehicle-to-vehicle communications*. Technical Report. United States. National Highway Traffic Safety Administration.

[38] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. 2007. Constraint-based random stimuli generation for hardware verification. *AI magazine* 28, 3 (2007), 13–13.

[39] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. 84–87.

[40] Francis Palma, Tamer Abdou, Ayse Bener, John Maidens, and Stella Liu. 2018. An improvement to test case failure prediction in the context of test case prioritization. In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering*. 80–89.
[41] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2019. GeoScenario: An open DSL for autonomous driving scenario representation. In *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 287–294.
[42] Christian Roesener, Felix Fahrenkrog, Axel Uhlig, and Lutz Eckstein. 2016. A scenario-based assessment approach for automated driving by using time series classification of human-driving behaviour. In *2016 IEEE 19th international conference on intelligent transportation systems (ITSC)*. IEEE, 1360–1365.
[43] Stuart J Russell and Peter Norvig. 2021. *Artificial intelligence a modern approach*. Pearson Education, Inc.
[44] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.
[45] Galina Sidorenko, Wojciech Mostowski, Alexey Vinel, Jeanette Sjöberg, and Martin Cooney. 2021. The CAR approach: Creative applied research experiences for Master's students in autonomous platooning. In *2021 30th IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*. IEEE, 214–221.
[46] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.
[47] Johan Thunberg, Galina Sidorenko, Katrin Sjöberg, and Alexey Vinel. 2021. Efficiently Bounding the Probabilities of Vehicle Collision at Intelligent Intersections. *IEEE Open Journal of Intelligent Transportation Systems* 2 (2021), 47–59. https://doi.org/10.1109/OJITS.2021.3058449
[48] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.