Locality-based Test Selection for Autonomous Agents

Sina Entekhabi[1], Wojciech Mostowski[1], Mohammad Reza Mousavi[2], and Thomas Arts[3]

[1] Halmstad University, Halmstad, Sweden
{sina.entekhabi, wojciech.mostowski}@hh.se
[2] King's College London, London, England
mohammad.mousavi@kcl.ac.uk
[3] Quviq AB, Gothenburg, Sweden
thomas.arts@quviq.com

Abstract. Automated random testing is useful in finding faulty corner cases that are difficult to find by using manually-defined fixed test suites. However, random test inputs can be inefficient in finding faults, particularly in systems where test execution is time- and resource-consuming. Hence, filtering out less-effective test cases by applying domain knowledge constraints can contribute to test effectiveness and efficiency. In this paper, we provide a domain specific language (DSL) for formalising locality-based test selection constraints for autonomous agents. We use this DSL for filtering randomly generated test inputs. To evaluate our approach, we use a simple case study of autonomous agents and evaluate our approach using the QuickCheck tool. The results of our experiments show that using domain knowledge and applying test selection filters significantly reduce the required number of potentially expensive test executions to discover still existing faults. We have also identified the need for applying filters earlier during the test data generation. This observation shows the need to make a more formal connection between the data generation and the DSL-based filtering, which will be addressed in future work.

Keywords: Test Input Generation · Domain Specific Languages · Test Selection · Autonomous Agents · Scenario-based Testing · Model-Based Testing

1 Introduction

It is well-known [22] that testing and debugging account for more than half of the development costs. Test automation, e.g., using Model-Based Testing (MBT) [16], mitigates this problem by generating tests at low additional cost once a model is in place. However, in some application areas, test execution is very time- and resource-intensive. In particular, this applies to our research project SafeSmart[4], where we consider cooperative (semi-)autonomous vehicles

[4] https://hh.se/safesmart

supported by V2X communication [29]. In our project, testing cooperative behaviour in complex urban traffic scenarios requires full simulation cycle for each test and is hence, extremely resource consuming. Our main objective is thus to identify *interesting* tests, i.e., ones that can effectively provide challenging scenarios that may stress the system under test and reveal severe faults by triggering failures.

To address this objective, we introduce a Domain Specific Language (DSL) for defining locality-based constraints for our autonomous agents moving on a grid. We implement this DSL for filtering the test cases randomly generated by the QuickCheck tool that we use in our project. To evaluate our approach, we implement and use a downsized form of the project case study, namely a set of autonomous agents moving in a grid, which we call *SafeTurtles* [8]. We use SafeTurtles for conducting an experiment with a few filtering constraints, defined by our DSL, and analyse and compare the results of testing with and without filters. This analysis is mainly in terms of the most expensive task in our testing approach, i.e., the number of test executions until a failure is found. Using this experiment, we answer the following two research questions:

Q1 Can filtering test inputs make fault detection more efficient?
Q2 Can filtering test inputs lead to a more efficient process for finding the most concise failing test input?

The answers to these research questions have a significant impact on reducing the test execution time: Q1 implies that we can find challenging test cases more efficiently and Q2 implies that we can close up on the "causes" for such failures more efficiently.

Finally, we also discuss the need of having a tailored data generator in the first place for the approach to be meaningful, while the natural step of deriving test cases directly from the DSL is a topic for future work.

In the remainder of this paper, we present a brief overview of our context in Sec. 2. Our testing methodology and our proposed DSL for formalising test selection constraints are explained in Sec. 3 and Sec. 4, respectively. To evaluate our approach, an experiment is designed, carried out, and its results are analysed in Sec. 5. Finally, related work is discussed in Sec. 6, and the paper is concluded in Sec. 7.

2 Context

2.1 SafeSmart Project

The wider context of our work is the SafeSmart project [29] that investigates *Safety of Connected Intelligent Vehicles in Smart Cities* from different angles, including vehicle-to-X (V2X) communication, localisation of objects on the road, and control of vehicles. These topics are investigated in the context of dense urban traffic, and the primary technique to validate the developments is simulation. Our particular objective is the application of model-based techniques [16]

for testing systems in this domain. We start off by using Property-Based Testing (PBT) with random test data generation.

2.2 QuickCheck

In the context of the SafeSmart project, we use an advanced PBT tool QuickCheck[5] [1]. Random input data generation in QuickCheck is supported by dedicated data generators for different data types (numbers, lists, vectors) with capping and distribution parameters, and the ability to combine the generators to build more complex data structures.

In QuickCheck, when a generated test fails, to ease debugging, the tool looks for and reports the most concise test failing input to report by modifying a failing test input into a "smaller" input and retrying the test. The way the data is modified is inherent in each particular data generator following a data type specific heuristic. This process is called *shrinking*. If a smaller test still fails, QuickCheck has gotten one step closer to the most concise failing input; this is called a *successful shrinking step*. Otherwise, if a smaller input data does not lead to a failed test, the process may back-track and try other ways of reducing the test input. This is called a *failed shrinking step*. This process continues up to the point, where no more successful shrinking is possible, and the last modified input is reported as the most concise input.

3 Methodology

While automated random testing can be useful in generating unforeseen scenarios, the test execution cost can become prohibitive in using it for embedded autonomous systems. Hence, we propose to exploit domain knowledge in selecting tests along with having the randomness factor. In our methodology, we first define a random data generator and use a DSL to formalise the domain knowledge for filtering the uninteresting cases from test execution. Filtering, hence, aims to increase the fault detection capability of the generated test cases.

Our target domain for SUTs is the domain of autonomous agents moving on a grid. Each of these agents has a goal coordinate on the grid and plans a path to reach the goal, including some forced waiting steps. However, the agents do not have to follow the planned path, if they need to avoid a collision. The input of the test process is the grid size (X, Y), the number of agents within the grid, and the number of the their waiting and displacement (action) steps. The output is the sequence of actual movement steps of the agents in response to the planned paths. The testing property of concern is the existence of a collision event in the system execution output, see Fig. 1.

3.1 Testing module

In our methodology, the testing module generates random inputs, filters them based on the given constraints, and executes the selected tests (after filtering)

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

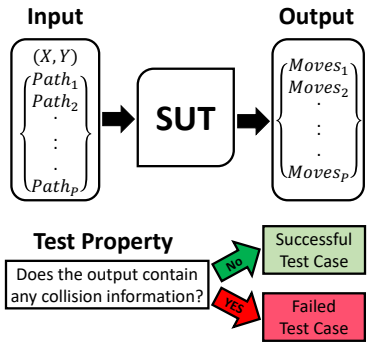


Fig. 1: The SUT of autonomous agents and the testing property

on the SUT. After executing each selected test input, the existence of collisions is checked. If no collision is detected another set of inputs is generated in the next attempt.

The module presented in Mod. 1.1 is the specification for testing our SUT using QuickCheck. The parameters of this module are respectively the grid dimensions, the number of agents in the gird, the number of displacement and waiting steps of each agent, and the filtering function (predicate) that is used for selecting the generated set of test inputs. In this module, first, a set of random paths is generated for the agents by a function which we named `pathGenerator` (line 3). Then, the SUT is executed to move all the agents based on the suggested generated paths in the given grid (line 6), and the existence of collisions is checked in the output trace afterwards (line 7).

Module 1.1: QuickCheck module for testing the SUT of autonomous agents

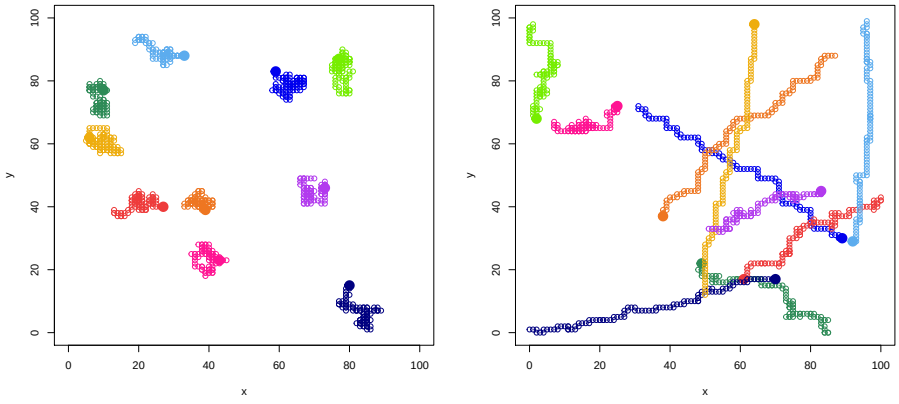
```
1 testCollision(X,Y,AgentsNum, ActionSteps, WaitSteps, FilterFunc)->
2   ?FORALL(AgentsPaths,
3           pathGenerator(X, Y, AgentsNum, ActionSteps, WaitSteps),
4           ?IMPLIES(FilterFunc(AgentsPaths),
5                    begin
6                      Trace = sut:run(AgentsPaths, X, Y),
7                      not lists:keymember({event, collision}, 1, Trace)
8                    end)).
```

3.2 Random Data Generation

Although randomness enables contrived corner case discovery, the whole process is still likely to statistically produce much more passing test scenarios rather than failing ones. Therefore, the way random inputs are generated can have a significant effect on fault detection capability and efficiency. In this work, a random data generator is specified for paths of a given length (line 3 in Mod. 1.1). In the remainder of this section, we discuss two approaches to generating random paths.

Uniform random data generator A random path with displacement length n can be generated by picking a random initial position in the grid, such as (x, y), and picking n sequential random actions (by uniformly random generators) from the *Action* list $\{Up, Down, Left, Right\}$. Although at first this may be a natural way of generating paths, in practice this method generates *clustered* paths making them unsuitable for triggering collisions.

By definition, having a set of elements and a uniform random generator, in random selection of n elements from this set when n is very large, the number of each element of the set in the selected sequence is statistically the same. Therefore, when n is very large, the number of *Up*-s and *Down*-s and the number of *Left*-s and *Right*-s would be equal in a randomly selected sequence, the agents would end up close to their initial position at the end of their travel. Therefore, by testing the SUT with these generated paths, the collision avoidance feature of the SUT is rarely tested, as illustrated in Fig 2a.



(a) Uniform Path Generator (b) Targeted Path Generator

Fig. 2: Examples of generating random paths of length 100 in 100×100 grid for 10 agents with the uniform- and targeted generators

Targeted Data Generation To address the problem of generating more diverse paths, we guide the path generation by defining and targeting random endpoint N for paths. Namely, we first select a random endpoint that is reachable in n moves from the initial state of the agent (i.e., a point in the circle centered at M and with radius n).Then, we generate a random simple path that reaches from M to N; if the path involves less moves than n moves then random pairs of $\{Left, Right\}$ and/or $\{Up, Down\}$ are added to the path. Finally, the

moves of the planned path are shuffled to add more randomness to the moves. An outcome of this strategy for data generation is illustrated in Fig. 3.

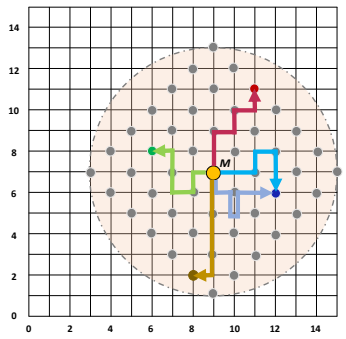


Fig. 3: The possible end points and some random paths with length 6 starting from point M

Our targeted data generator resolves the problem of having compact paths in the uniform data generator, as it can be seen in Fig. 2b. In a simple experiment, we generated paths for 100 agents by these generators in a 1000×1000 grid having displacement length 100. With the first generator, on average, the agents move in squares of 11×10 area. However, this is extended to an average area of 42×43 by using the targeted generator.

4 Filtering DSL: Syntax and Semantics

In this section, we define a domain-specific language to specify filtering constraints on test cases. These constraints are supposed to capture the domain knowledge regarding the fault-detection capability of test cases. This is akin to the criteria used for test-case prioritisation [26].

4.1 Syntax

The syntax of our DSL is presented in Mod. 1.2. According to this syntax, a filtering constraints can be specified as a simple area condition (in line 1 of Mod. 1.2) or a logical combination of constraints `NOT`, and combination of `Constraint`-s with `AND` and `OR` operators (in lines 2–4 of Mod. 1.2). An area constraint first specifies an area, which can be defined as a circle with a specified radius or a square with a specified side as an integer (in lines 6 and 7 of Mod. 1.2). The second and final part involves locality conditions that specify a minimum number of agents at some arbitrary time in a given area (in line 9) and a minimum number "n" of path intersections of degree "d" (in line 10). The intersection

degree for a point is the number of agents that visit that point sometime in their route in a given area. Similar to `Constraint`, locality conditions can also be combined using logical connectives. This syntax can be extended with other domain-specific objects of our target domain to cater for other notions of fault detection capability.

Module 1.2: The DSL for locality-based test selection constraint definition for autonomous agents

```
1  Constraint ->   IN Area Condition |
2                  AND Constraint Constraint |
3                  NOT Constraint |
4                  OR Constraint Constraint
5
6  Area       ->   Circle Integer |
7                  Square Integer
8
9  Condition  ->   Count Integer |
10                 Intersection Integer Integer |
11                 And Condition Condition |
12                 Not Condition |
13                 Or Condition Condition |
```

To illustrate the syntax, a few test selection constraints are defined next for the test input represented in Fig. 4, which includes the suggested movement paths of four agents in a 7×7 grid. All the agents plan to start their moves at the same time $t = 0$, and stop movement after 6 moves at time $t = 6$. The actual running actions in different times will be obviously affected by the decisions of the agents adapting to the traffic.

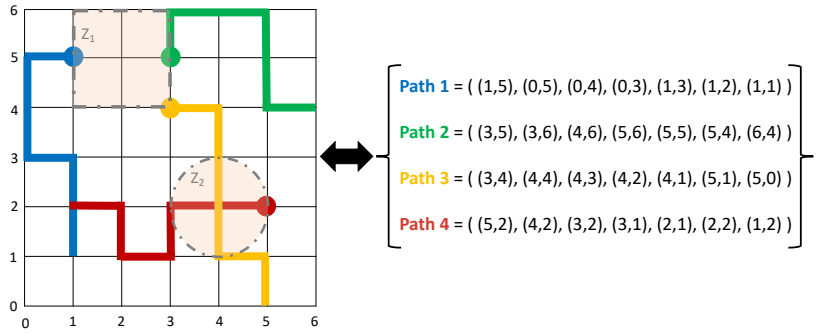


Fig. 4: Test input example including the suggested paths of four autonomous agents

- `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 ($t = 0$) could 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 (3 and 4) crossing a particular point ($(4, 2)$) and that the point is included in a circle with radius 1 (the Z_2 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 that both of the conditions "`Intersection 1 2`" and "`Count 3`" are satisfied in that area.

4.2 Semantics

The formal semantics of our DSL is defined in Mod 1.2 in terms of an *eval* function that maps every constraint and a list of paths to a Boolean. Our semantics assumes a $g \times g$ grid containing m agents with l number of actions (including waiting steps) in total. The only non-trivial case in the definition of *eval*, which uses the auxiliary function *evalCon*; the application of the latter function checks whether there exists an area that satisfies a constraint. The geometric definition of the area is ascertained by function *areaContains*, while the condition to be satisfied in the specified area is checked by function *getCases*; the latter function makes a case distinction based on the type of condition to be satisfied and generates the associated constraint on the involved paths.

$$
\begin{aligned}
&eval :: Constraint \to [Path] \to Boolean \\
&eval\ (IN\ a\ c\ )\ P = \exists\ x, y \in \{0, \ldots, (g-1)\}\ evalCon(AreaInstance\ a\ (x, y)\ ))\ c\ P \\
&eval\ (NOT\ f)\ P = \neg\ eval(f\ P) \\
&eval\ (f_1\ AND\ f_2)\ P = (eval\ f_1\ P) \wedge (eval\ f_1\ P) \\
&eval\ (f_1\ OR\ f_2)\ P = (eval\ f_1\ P) \vee (eval\ f_1\ P)
\end{aligned}
$$

$$
\begin{aligned}
&evalCon :: AreaInstance \to Condition \to [Path] \to Boolean \\
&evalCon\ (AreaInstance\ a\ (x, y))\ c\ P = \exists\ z \in (getCases\ c\ P) \\
&\quad areaContains\ (a\ (x, y))\ z \\
&evalCon\ (Not\ (i\ c\ P)) = \neg\ evalCon(i\ c\ P) \\
&evalCon\ (AreaInstance\ a\ (x, y))\ (c_1\ And\ c_2)\ P = \exists\ z \in getCases\ (c_1\ P) \\
&\quad (areaContains\ (a\ (x, y))\ z)\ \wedge (evalCon\ (AreaInstance\ a\ (x, y))\ c_2\ P) \\
&evalCon\ (AreaInstance\ a\ (x, y))\ (c_1\ Or\ c_2)\ P = \exists\ z \in getCases\ (c_1\ P) \\
&\quad (areaContains\ (a\ (x, y))\ z)\ \vee (evalCon\ (AreaInstance\ a\ (x, y))\ c_2\ P)
\end{aligned}
$$

$$getCases :: Condition \to [Path] \to [checkCase]$$
$$getCases\ (Count\ n)\ P\ =\ (S, n)\ where$$
$$S = \{\ s \mid t \in \{1, \ldots, l\}\ \wedge\ Q = \{P[1][t], \ldots, P[m][t]\}\ \wedge s\ \subseteq\ 2^Q \wedge |s| = n\ \}$$
$$getCases\ (Intersection\ d\ n)\ P\ =\ (S, n)\ where$$
$$S = \{s\}, s = \{(x, y) \mid \exists Q \subseteq P\ \ |Q| = d\ \ \forall q \in Q\ \exists t\ (x, y) = w_i[t]\ \}$$

$$areaContains :: AreaInstance \to CheckCase \to Boolean$$
$$areaContains\ (AreaInstance\ (Circle\ r)\ c)\ (S,\ n) = \exists Q \in S$$
$$\forall q \in Q\ (q_x - c_x)^2 + (q_y - c_y)^2 \leq r^2$$
$$areaContains\ (AreaInstance\ (Square\ d)\ c)\ (S,\ n) = \exists Q \subseteq S[1]\ |Q| = n$$
$$\forall q \in Q\ |q_x - c_x| \leq \frac{d}{2}\ \wedge\ |q_y - c_y| \leq \frac{d}{2}$$

Here are the used types:

$$type\ Point\ =\ (Integer, Integer)$$
$$type\ Path\ =\ [Point]$$
$$type\ CheckCase\ =\ ([Path], Integer)$$
$$newtype\ AreaInstance\ =\ AreaInstance\ Area\ Point$$

5 Experiments

In this section, we design and conduct an experiment to answer the research questions set forth in the introduction, which we recall below:

Q1 Can filtering test inputs make fault detection more efficient?
Q2 Can filtering test inputs lead to a more efficient process for finding the most concise failing test input?

QuickCheck is used for tool support in this experiment where the test inputs are generated randomly with and without filters defined with our DSL, and the results are compared with each other at the end. The chosen experiment size is a 100×100 grid including 5 agents where each agent has 5 displacement steps and 5 waiting steps in total.

The SUT instance of this experiment is a set of autonomous agents called SafeTurtles [8], which are implemented in Erlang. The choice of language is mainly dictated by the ease of interfacing to QuickCheck, but otherwise any other programming language could be used for the SUT, as QuickCheck is very flexible to make any kind of a connection to the SUT. In SafeTurtles, there are a few agents, called turtles, that can move on the grid. Each turtle has a goal and a planned path for reaching its goal. The control algorithm of each turtle

is supposed to observe the environment and autonomously avoid collisions with other turtles. However, due to the intentional weakness of the collision avoidance mechanism, the turtles do not take move prediction into account. Therefore, if more than one turtle try to step onto the same position at the very same time, they will collide.

In this experiment, the following filtering constraints F1, F2 and F3 are used along with the targeted data generator (explained in Sec. 3.2), and for a better evaluation of the results, the tests are repeated 100 times in each case to get good statistics.

- **F1**: In Circle 5 Count 2
- **F2**: In Circle 3 Count 2
- **F3**: In Circle 1 Intersection 1 2

Among these filters, F2 is defined to be stricter than F1, i.e., for all sets of test cases, F2 accepts a subset of those test cases accepted by F1. However, rejecting many test inputs does not necessarily mean that the filter makes fault detection more efficient. Among the mentioned filters, the concern captured by F3 is different from both F1 and F2. Choosing these different types of filters is expected to give us more insight on the effect of filtering the test cases.

5.1 Fault detection time

The total testing time comprises two major parts: test input generation time and test execution time. Test execution requires execution or simulation of the SUT and hence, the test execution time is expected to be significantly larger than test input generation time. While, for the sake of completeness, we also consider test input generation time, we expect test execution time to be much more significant and hence, will be the focus of our experiment results.

Test execution time Figure 5a represents the number of (passed) test cases up to detecting the first fault by using different filters. To make a rigorous analysis of the obtained results, statistical hypothesis testing is used. Here, the considered statistical question concerns if the mean of one population is significantly smaller than the other one. To start with, we applied the *Kruskal-Wallis* test [19], to check if there is any significant difference in the mean of these four populations; we got the p-value less than $2.2e^{-16}$; meaning with confidence level 99% the test detects that there is some significant difference among the groups. To zoom into the differences, we next performed pairwise tests between all pairs of populations (due to space restrictions, we report only some of them below and in Table 1).

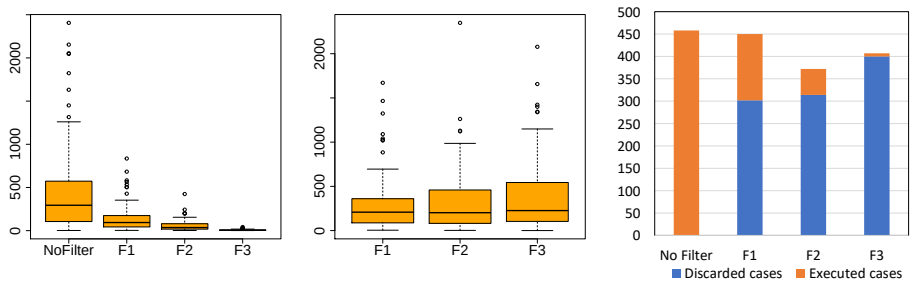
We first checked the normality of the data distributions. For this purpose, *Shapiro-Wilk* test [27] is applied on the data, and since the calculated p-values are below 0.05 (and even below 0.01) for the number of executed tests for each of the filtering cases, the data are supposed not normal[6]. As a result, based on

[6] The experimental data and the code of statistical tests are available in "`exp`" sub-directory of [8].

the required statistical question, one tailed *Mann-Whitney-Wilcoxon u-test* [30] is selected for doing statistical analysis on the number of executed tests.

As shown in Table 1, having the alternative hypothesis "the number of required test executions till reaching the failed test by having the filter F1 is significantly smaller than the case of having no filter", the p-value of the t-test is less than 0.01 for our data. It means with confidence level 99% the supposed alternative hypothesis is valid. Applying other u-tests for similar hypotheses also show that with confidence level 99% the number of tests by having F2 is significantly smaller that F1, and the number of tests by having F3 is significantly smaller than F2. These results indicate that having each of these filters lead to detecting the intended fault with smaller number of test cases than having no filter. However, for the particular fault of the system, the filter F3 has better results than F2 and F1.

Figures 5b and 5c show the number of discarded test cases and the relative portions of accepted and discarded cases for each case, respectively. In Fig. 5c, it can be seen that from a total of over 400 generated test inputs, more than 300 were discarded by any filtering strategy.



(a) Executed cases (b) Discarded cases (c) Average result

Fig. 5: The number of executed (accepted) and discarded test cases up to detecting the failure

Valid test input generation time Test input generation time depends on the complexity of the data generator. When applying filtering, the complexity filtering can increase the test input generation time. However, there is a delicate interaction between data generation and filtering: "smarter" data generation schemes may take more time, but at the same time may lead to fewer ineffective test cases; the latter will lead to fewer discarded test cases and hence, save some time in the filtering phase. For instance, for the proposed generators of Sec. 3.2 and the SUT parameters of Fig. 2, to generate a single valid test input having filter F3 with uniform data generator, about 87 test inputs are discarded on average (calculated from 100 attempts). However, the targeted data generator

reduces this number to only 7 discarded cases for a single valid test input. This shows that, as expected, the test data generator is much less costly than the uniform data generator in generating valid test inputs when the test selection constraint is a filter like F3.

	P-value		
	Executed test cases (u-test)	**Successful shrink steps (t-test)**	**Failed shrink steps (u-test)**
F1	$2.149e^{-08}$	0.3443	0.001143
F2	$<2.2e^{-16}$	0.06088	$4.426e^{-06}$
F3	$<2.2e^{-16}$	0.0329	$3.214e^{-09}$

Table 1: The p-values of applying statistical tests on our experiment results with the alternative hypothesis that "applying no test input filter results in a higher number of executed test-cases before reaching the first failure (resp. successful and failed shrinking steps) than applying filters F1, F2, or F3".

5.2 Shrinking time

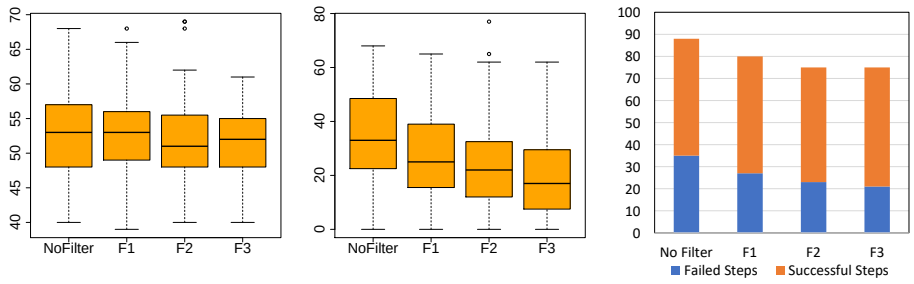
In QuickCheck, shrinking is a mechanism to reduce a failing test case in order to help the tester identify the root cause of failure. A successful shrinking step indicates that the failed test cases involved steps that did not effectively contribute to failure and hence, the test case could be shrunk by removing them. Figures 6a and 6b show the number of successful and failed shrink steps in reaching for the most shrunk failing test inputs in our experiment. As the p-values of applying Shapiro-Wilk test on successful shrink step results are greater than 0.05 (meaning normal data) and on failed shrink steps are less than 0.05 (meaning not normal data), *t-test* [28] is used for the comparison of successful shrink step results and u-test for the failed ones.

We first applied the *Anova* test [9] for checking whether the four categories show a significant difference in the means of *successful* shrinking steps. (In this case, *Bartlett* test with p-value 0.43 indicated that the variances of the groups are not significantly different and we can apply Anova test on them). It turns out that the successful shrinking steps are not significantly different according to the Anova test. Namely, the Anova test shows p-value 0.169 (greater than 0.01), meaning that at least with confidence level 95% the groups are not significantly different than each other. As shown in Table 1, we also considered pairwise differences, defining the alternative hypothesis to "having no filters leads to a smaller number of *successful* shrink steps"; in this case, the p-values amount to about 0.3, 0.06, and 0.03 for filters F1, F2 and F3, respectively. It means at least with confidence level 95%, only the filter F3 significantly reduce the number of successful shrink steps in this experiment.

On the other hand, the data indicates that the case for *failed* shrinking steps is clearer: the Kruskal-Wallis test on the number of failed shrinking tests leads to

the p-value $1.345e-08$ (less than 0.01); hence, with confidence level 99%, at least one of the groups is significantly different than one other. Our pair-wise u-tests in the paper goes deeper into that and confirms this result as follows. As shown in Table 1, for the target alternative hypothesis of "having significantly smaller number of *failed* shrink steps by having filters", it results in p-values less than 0.01 for each the filters F1, F2 and F3. It means that with confidence level 99% there is a significant improvement in decreasing the number of failed shrink steps by having these filters. This happens because the filtering constraints directly eliminate the (modified) inputs from test execution that cannot result in failure. In addition, doing u-test for a similar hypothesis shows that with confidence level 95% the number of failed shrink steps by F3 is significantly less than F1 and F2 as well. Figure 6c shows the average number of successful and failed shrink steps in this experiment. The average number of successful shrinking steps is very close in all of the cases. Nevertheless, due to discarding some of the idle test inputs by filtering in the shrinking process, smaller number of test execution is required by having the filters on average in the shrinking process.

In order to analyse further the mutual effect of filtering and shrinking it is useful to apply different strategies and constraints in initial data generation and shrinking. However, QuickCheck does not support this feature yet.



(a) Successful steps (b) Failed steps (c) Average result

Fig. 6: The number of shrinking steps

Threats to Validity The subject system used in our experiments is an abstraction of real-world autonomous systems. To address this threat, we plan to extend our SUT to accommodate more domain concepts in our project and in tandem extend our DSL to reflect the extended domain knowledge. Although our experiments show promising results with our specific targeted data generator and filtering constraints, the results cannot be generalised to other data generators or filtering constraints. We would like to consider a wider variety of data generators and incorporate filtering constraints in them, and also study the relative effect of different data generators, filtering constraints and their complexity, and test execution platforms in our future work.

6 Related Work

Random testing has been used as a lightweight method for testing systems, particularly at their early stages of development and deployment [5,21,14]. To mitigate the prohibitive cost of test execution, one could augment random testing with either more intelligent test generation algorithms or filtering and test selection criteria. From the former category, using constraint solvers and search-based algorithms are prominent examples. The latter approach, i.e., filtering, has the advantage of being compositional, i.e., different filters can be composed to cover different aspects of test design. Furthermore, constraint solving would typically lead to the same test values for a particular constraint, while random data generation with filtering provides certain degree of test data variability each time.

Considering code level constraints, TestEra [17] and Korat [3] are examples of having pure filtering style; ASTGen [6] has a pure generating style, and UDITA [13] can be used for both filtering and generating of test cases. For defining test harnesses, TSTL [15] also provides a DSL for test data generation. Our approach puts much more emphasis on embedding domain knowledge in filtering rather than test generation. However, the principles of our approach can also be applied to design intelligent generators and a thorough empirical comparison of the two alternative approaches remains as future work, especially that we witnessed a clear dependency of results on the link between the data generator and the filter.

Scenario-based testing is a well-studied area in testing autonomous systems. Concrete scenarios for testing can be designed by either analysing the crash data [23,4] or naturalistic driving data (NDD) [18,25]. For analysis and simulation of particular scenarios in cyber-physical systems (CPS), several DSLs are designed [11,2,24,10]. Fremount et al. [12] used SCENIC [11] for defining parametric scenarios for testing autonomous vehicles and used VERIFAI toolkit [7] for the analysis of the scenarios and generating concrete test case and used SVL [20] simulator for executing test cases. Our main departure point from much of the informal scenario-definition languages [2,24,10] is the rigorous geometric and logical basis for our DSL. Compared to other languages that do have a formal basis [7,12], our focus on locality of grid-based agents is a distinctive feature of our DSL. We do expect that our DSL can be extended with other features in the aforementioned languages and our concrete filters can be composed with theirs to cover different aspects of the domain.

7 Conclusions and Future work

In this paper, we proposed a methodology for filtering randomly generated test cases in order to make fault detection more efficient. We have implemented our methodology in QuickCheck and used a case study of autonomous agents to empirically evaluate the proposed methodology. Our empirical results indicate that filters reflecting domain knowledge can significantly reduce the time to reach

failures. Also our results indicate that in the process of shrinking a failing test case into a minimal one, using filters can lead to fewer failed shrinking attempts.

As a natural next step, we would like to incorporate the definition of filters into the data generators. In other words, instead of generating and then filtering, we would like to generate test data (also with possible randomness) from the DSL specification. This would involve extending QuickCheck with new DSL-based data generators that would also allow for better results in the test shrinking process.

Finally, we would like to scale up our case study towards our demonstrator within the SafeSmart project. The next step is using the existing Robot Operating System (ROS) version of our case study, which features more elaborate decision making by the agents as well as continuous dynamics of agents. Further, we shall apply our method in the context of SUMO/Veins simulations of communicating vehicles (V2X). Our objectives will go beyond collision-freedom and consider other dangerous or undesired configurations of the system, e.g., excessive braking of the vehicles [29]. The semantics of our DSL should also be extended to not only consider possible failures, but also consider severity and likelihood of undesired situations. This will lead to a model-based framework for evaluating both safety and comfort of the autonomous system under test.

Acknowledgements. We thank Jan Tretmans, Verónica Gaspes, and the anonymous reviewers of ICTSS for their valuable comments on this work. 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/V026801/1.

References

1. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with quviq quickcheck. In: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang. pp. 2–10 (2006)
2. ASAM: ASAM openSCENARIO. https://www.asam.net/standards/detail/openscenario/ (June 2021)
3. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates. ACM SIGSOFT Software Engineering Notes **27**(4), 123–133 (2002)
4. Carsten, O., Merat, N., Janssen, W., Johansson, E., Fowkes, M., Brookhuis, K.: Human machine interaction and safety of traffic in europe. HASTE final Report **3** (2005)
5. Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.: Adaptive random testing: The art of test case diversity. Journal of Systems and Software **83**(1), 60–66 (2010)
6. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 185–194 (2007)

7. Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In: International Conference on Computer Aided Verification. pp. 432–442. Springer (2019)
8. Entekhabi, S., Arts, T.: Safesmartturtle. https://github.com/ThomasArts/SafeSmartTurtle (June 2021)
9. Fisher, R.A.: Xv.—the correlation between relatives on the supposition of mendelian inheritance. Transactions of the Royal Society of Edinburgh **52**(2), 399–433 (1919). https://doi.org/10.1017/S0080456800012163
10. Foretellix Inc.: M-SDL. https://www.foretellix.com/open-language/ (June 2021)
11. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 63–78 (2019)
12. Fremont, D.J., Kim, E., Pant, Y.V., Seshia, S.A., Acharya, A., Bruso, X., Wells, P., Lemke, S., Lu, Q., Mehta, S.: Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC). pp. 1–8. IEEE (2020)
13. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in udita. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. pp. 225–234 (2010)
14. Hamlet, D.: When only random testing will do. In: Proceedings of the 1st international workshop on Random testing. pp. 1–9 (2006)
15. Holmes, J., Groce, A., Pinto, J., Mittal, P., Azimi, P., Kellar, K., O'Brien, J.: Tstl: the template scripting testing language. International Journal on Software Tools for Technology Transfer **20**(1), 57–78 (2018)
16. Jonsson, B., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems: Advanced Lectures. LNCS, Springer (2005). https://doi.org/10.1007/b137241
17. Khurshid, S., Marinov, D.: Testera: Specification-based testing of java programs using sat. Automated Software Engineering **11**(4), 403–434 (2004)
18. Kruber, F., Wurst, J., Botsch, M.: An unsupervised random forest clustering technique for automatic traffic scenario categorization. In: 2018 21st International Conference on Intelligent Transportation Systems (ITSC). pp. 2811–2818. IEEE (2018)
19. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. Journal of the American statistical Association **47**(260), 583–621 (1952)
20. LG Electronics Inc.: SVL Simulator. https://www.svlsimulator.com/ (June 2021)
21. Liu, H., Xie, X., Yang, J., Lu, Y., Chen, T.Y.: Adaptive random testing through test profiles. Software: Practice and Experience **41**(10), 1131–1154 (2011)
22. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
23. Najm, W.G., Toma, S., Brewer, J., et al.: Depiction of priority light-vehicle precrash scenarios for safety applications based on vehicle-to-vehicle communications. Tech. rep., United States. National Highway Traffic Safety Administration (2013)
24. Queiroz, R., Berger, T., Czarnecki, K.: Geoscenario: An open dsl for autonomous driving scenario representation. In: 2019 IEEE Intelligent Vehicles Symposium (IV). pp. 287–294. IEEE (2019)
25. Roesener, C., Fahrenkrog, F., Uhlig, A., Eckstein, L.: 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). pp. 1360–1365. IEEE (2016)

26. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Test case prioritization: An empirical study. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). pp. 179–188. IEEE (1999)
27. SHAPIRO, S.S., WILK, M.B.: An analysis of variance test for normality (complete samples). Biometrika **52**(3-4), 591–611 (dec 1965). https://doi.org/10.1093/biomet/52.3-4.591, https://doi.org/10.1093/biomet/52.3-4.591
28. Student: The probable error of a mean. Biometrika pp. 1–25 (1908)
29. Thunberg, J., Sidorenko, G., Sjöberg, K., Vinel, A.: Efficiently bounding the probabilities of vehicle collision at intelligent intersections. IEEE Open Journal of Intelligent Transportation Systems **2**, 47–59 (2021). https://doi.org/10.1109/OJITS.2021.3058449
30. Wilcoxon, F.: Individual comparisons by ranking methods. biom. bull., 1, 80–83 (1945)