Test Suite Reduction for Self-organizing Systems: A Mutation-based Approach

André Reichstaller
University of Augsburg, Germany
reichstaller@isse.de

Benedikt Eberhardinger
University of Augsburg, Germany
eberhardinger@isse.de

Hella Ponsar
University of Augsburg, Germany
ponsar@isse.de

Alexander Knapp
University of Augsburg, Germany
knapp@isse.de

Wolfgang Reif
University of Augsburg, Germany
reif@isse.de

ABSTRACT

We study regression testing and test suite reduction for self-organizing (SO) systems. The complex environments of SO systems typically require large test suites. The physical distribution of their components and their history-dependent behavior, however, make test execution very expensive. Consequently, an efficient test suite reduction mechanism is needed. The fundamental characteristic of SO systems is their ability to reconfigure themselves. We thus investigate a mutation-based approach concentrating on reconfigurations, more specifically the communication between the distributed components in reconfigurations. Due to distribution, we argue for an explicit consideration of higher-order mutants and find a shortcut that makes the number of test cases to execute before reduction feasible. For the reduction task, we evaluate the applicability of two existing clustering techniques, *Affinity Propagation* and *Dissimilarity-based Sparse Subset Selection*. It turns out that these techniques are able to drastically reduce the original test suite while retaining a good mutation score. We discuss the approach by means of a test suite for a self-organizing production cell as a running example.

KEYWORDS

Self-organization, mutation testing, test suite reduction

ACM Reference Format:
André Reichstaller, Benedikt Eberhardinger, Hella Ponsar, Alexander Knapp, and Wolfgang Reif. 2018. Test Suite Reduction for Self-organizing Systems: A Mutation-based Approach. In *AST'18:IEEE/ACM 13th International Workshop on Automation of Software Test , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194733.3194739

1 TESTING SELF-ORGANIZING SYSTEMS

Testing is the effort of executing programs with the intention of revealing failures. That sets three prerequisites: (1) the system under test (SuT) has to be controllable, (2) the output of the SuT has to be observable, and (3) a specification is needed to compare the intended behavior with the observed behavior.

Self-organizing (SO) systems are distributed systems that autonomously adapt their distribution structure and their behavior in order to fulfill their goals under ever-changing environments and conditions [21]. More often than not, their reconfiguration mechanisms are inspired by techniques and concepts of the field of artificial intelligence (AI), like particle swarm optimization or constraint satisfaction, in order to increase resilience, flexibility, or recoverability [4]. When it comes to testing, however, SO systems challenge the aforementioned fundamental prerequisites: (1) there is more an emergent behavior than a concrete, controllable method [11]; (2) there is a change in system structure and behavior accompanying a concrete output value; and (3) there is an underspecified system that takes autonomous responsibility rather then a specification of the concrete expected behavior. On the other hand, these characteristics make testing even more necessary for quality assurance.

For self-adaptation, SO systems typically rely on a kind of feedback loop: they continually monitor their environment; on detecting an adverse situation, they devise a reconfiguration plan; and finally they execute this reconfiguration plan re-affecting their environment. There may be several, concurrent such feedback loops and their implementation may be centralized in one component or, like the target system itself, distributed over a set of components. From a testing perspective the adverse environment situations can be classified as *environmental faults* that have to be handled by the SO system. A first kind of (SO) *system fault* results if such a situation persists without triggering a reconfiguration. A second kind of system fault results from devising an erroneous reconfiguration unsuitable for the given situation. Even when a correct reconfiguration plan has been calculated, a third kind of system fault may be reached when executing the reconfiguration plan, where the resulting configuration does not match the intended one.

In [6], we already considered functional testing of single SO mechanisms in isolation. This approach focuses on revealing failures from the second kind of system faults that concern the computation of reconfigurations suitable for a given situation. In [7], we expanded the approach in order to detect faults of all three kinds by applying a model-based back-to-back testing approach. The resulting framework allows for generating, executing, and evaluating a *test suite* (i.e. a set of test cases) by an automated oracle. Furthermore, in [8], we added a test case generation strategy to the framework that uses heuristics for finding boundary cases where

AST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5743-2/18/05…$15.00
https://doi.org/10.1145/3194733.3194739

the SO mechanism has only left rather little degrees of freedom for devising a reconfiguration. This strategy aims at uncovering faults that may be masked when there are plenty of reconfigurations to choose from.

Our test framework works on a simulation of the SO system and generally results in a large number of test cases. Due to the typically high physical distribution of the components of an SO system under test the sheer number of test cases would make test execution very expensive on actual hardware outside the simulation. In this paper, we thus investigate an approach for test suite reduction for SO systems. As it is a fundamental characteristic of SO systems, we focus on reconfigurations and the resulting component communication for triggering, calculating, and distributing reconfigurations. We follow a mutation-based approach for covering these three kinds of SO system faults for selecting test cases on the basis of their mutation scores. Overall, the paper contains the following contributions:

- We propose a set of mutation operators that are suitable for mutation testing of self-organizing systems.
- Based on these mutation operators, we discuss the need for higher order mutants.
- We elaborate a mutation-based approach on test suite reduction that efficiently takes higher-order mutants into account.
- We show that established classification methods are able to find good approximations for the considered reduction task.

The remainder of the paper is structured as follows: In Section 2, we introduce a running example and evaluation case for our approach. Section 3 describes our mutation-based test suite reduction scheme which is implemented using clustering techniques as described and evaluated in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 A SELF-ORGANIZING PRODUCTION CELL

Future cyber-physical systems will integrate SO mechanisms to resolve the tasks of decentralized decision making, to optimize the system structure, and to autonomously react to component failures at run-time increasing the system's robustness. The self-organizing production cell, as shown in Fig. 1, is an example for future production scenarios. Within the production cell, the production stations are robots (R1, R2, R3, R4) equipped with toolboxes and the ability to change their tools whenever necessary (self-awareness). They are connected via mobile carts (C1, C2, C3) that are able to carry workpieces. The production cell can fulfill any task (Drill, Insert, Tighten, Polish), which corresponds to tools and capabilities available in the cell. This is possible due to the SO mechanisms that reconfigure the carts and robots such that the tools are applied to the workpieces in the correct order. Any violation of the calculated configuration at run-time triggers the SO mechanism calculating and distributing a new system configuration; this forms the responsibility of the SO mechanism in this setting.

Experimental Case. For our experiments and our evaluation we use the setting described by Fig. 1. The SO production cell is implemented in our model-based analysis and simulation tool S# (cf. [7]). This framework also allows for model-based testing of SO mechanisms and offers to plug-in different SO mechanisms. For the evaluation case of this paper we have three different SO mechanisms available—a handcraft central mechanism, a constraint solving based mechanism[1], and a decentralized mechanism.

In the test suites generated by the framework, the test cases are formulated as environmental faults, e.g., a broken driller for a specific robot, that are simulated as a test driver. In the setting of Fig. 1 there are 55 different kinds of environmental faults. The test case generation procedures are described in [7, 8] and rely on techniques that make use of a model checker to generate test inputs systematically based on the model-based description of the environment of the SuT. Neither a model of the SuT and the employed SO mechanism nor their source code are needed, and will, in general, not be available.

The test case generation approach also takes into account the dependency in the execution order of the test cases. The dependency results from the fact that the simulated and manipulated environment is also manipulated by the SO mechanism and further that specific environmental conditions are only reachable by chaining several test case executions (cf. [6]).

3 A MUTATION-BASED APPROACH

With their complex and ever-changing environments, the distributed and interacting components of SO systems are embedded in huge state spaces. The so-called state space explosion also promotes exploding test suites w.r.t. their number of test cases. Even for the rather small setting considered in our experimental case our S# approach returns a test suite comprising as many as 7524 test cases. The enormous effort for executing test suites like this makes regression testing practically unfeasible: test suite reduction techniques are needed. However, many established approaches on test suite reduction require access on source code (cf. Section 5). At least for the SO mechanisms, maybe involving techniques from AI, we cannot assume that their source code or some more abstract behavioral model are actually available. Thus, we have to follow a different alley: Building on the observation that most failures in reconfiguration are caused by communication faults in triggering, calculating, and distributing reconfiguration plans, we suggest a mutation-based approach on test suite reduction for SO systems. After roughly describing the general background of mutation-based testing, this section concentrates on the particular application on the SO scenario.

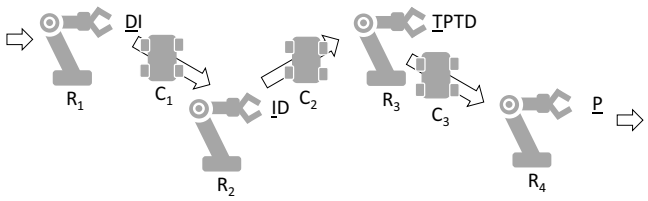
3.1 Mutants as Basis for Test Suite Reduction

Mutation-based testing assesses the quality of test cases by their ability to uncover slightly modified versions of a SuT [5]. With the intention to simulate possible failure scenarios, these modifications, the so called *mutants*, are generated by well-defined *mutation operators* that mimic typical faults. A test case uncovers—or more martially *kills*—a mutant if it is able to distinguish the mutant's behavior from that of the original version of the SuT.

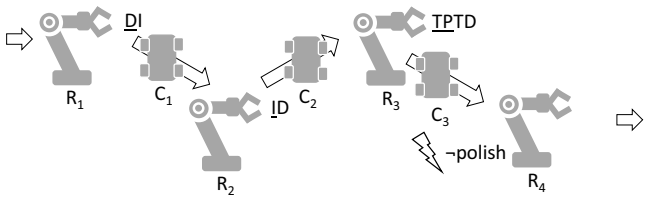
A mutant m can be described by a predicate on test cases:

$$isKilled_m \subseteq TestCases\ . \tag{1}$$

[1]The SO mechanism based on a constraint solver has been implemented in *MiniZinc* (cf. http://www.minizinc.org).



(a) The resource flow starts to the left where R_1 drills a hole into incoming workpieces. Workpieces are successively transported by C_1, C_2, and C_3. The robots apply their tools. Once R_4 is done, the workpieces leave the system.



(b) After R_4 loses its polish tool, the resource flow is reconfigured: R_3 is taking over the previous role of R_4.

Figure 1: A schematic overview of the self-organizing robot cell case study with four robots as well as three carts establishing the resource flow between them. The task is to apply the drill, insert, tighten, and polish capabilities to all incoming workpieces. Each robot's available tools are shown to its right, with D, I, T, and P; the currently allocated ones are underlined. Figure 1a shows an exemplary configuration of the robot cell. As depicted in Fig. 1b, faults result in tool losses that self-organization can cope with by reconfiguring the resource flow.

If $isKilled_m(tc)$ holds, we say that mutant m is killed by test case tc. Given mutants M, the killing set of tc is determined by

$$K_{tc} = \{m \in M \mid isKilled_m(tc)\} .$$

The confidence that test cases are able to reveal faults in real operation is quantified by a *mutation score*: The greater $|K_{tc}|$, the better the score for test case tc; and the greater $|\bigcup_{tc \in TS} K_{tc}|$, the better the score for a whole test suite TS. This holds for any minimized test suite, too.

A reduced test suite $TS' \subset TS$ with $|\bigcup_{tc' \in TS'} K_{tc'}| = |\bigcup_{tc \in TS} K_{tc}|$ maintains the confidence associated with TS at lower execution costs. Consequently, the test suite reduction task can be understood as finding a smaller test suite still killing all those mutants that were killed in the original test suite. In comparison to code-coverage based approaches this has a major advantage: We are able to prioritize parts of the source code based on the set of mutation operators used. This is, however, on the cost of a major disadvantage, namely the high *number of needed executions before reduction*: in order to investigate $isKilled_m(tc)$ for all the mutants $m \in M$ and test cases $tc \in TS$, we have to execute $|M| \cdot |TS|$ test cases in the worst case before we can actually reduce.

For our study, we consider the following exemplary mutation operators for SO systems:

- *Lost Reconfiguration Message (LRM)*: As soon as a robot detects an environmental fault (e.g., a broken driller) it should normally send a *reconfiguration message* to trigger the reconfiguration mechanism. This mutation operator suppresses such messages. In consequence there might be no reconfiguration in spite of an incorrect system configuration.
- *Needless Reconfiguration Message (NRM)*: The inverse of an LRM: A particular robot signals an environmental fault, although there actually is none. Consequently, unnecessary reconfiguration steps might be triggered.
- *False Reconfiguration (FR)*: This operator mimics the loss of a role allocation message that was sent by the reconfiguration mechanism. In consequence, one robot will (maybe erroneously) retain its previous role—this could result again in an incorrect system configuration.

3.2 The Need for Higher Order Mutants

Traditionally, mutation-based testing only considers mutants generated by a one-time application of a single mutation operator. Those mutants are called *first-order mutants* (FOMs). *Higher-order mutants* (HOMs) which are generated by multiple applications of mutation operators are usually neglected. This conforms with the so-called *coupling effect* which states that test cases constructed for explicitly killing FOMs do implicitly also kill combinations of them, as which HOMs can be seen [18]. There are, however, known exceptions that suggest to explicitly consider HOMs. Jia and Harman investigate the role of HOMs and give a classification based on the way they are *coupled* and *subsuming* [15].

For SO systems in general, and our experimental case in particular, we identify two phenomena in the interrelation of system faults which might lead to unwanted effects in test suite reduction, if they would only be considered as FOMs, not explicitly combined in HOMs:

(1) *Failure Masking*

Two or more injected system faults might mask the effect of one another. If so, their combination cannot be killed by any test case of the suite. Let us consider the exemplary FOMs $m1$ and $m2$ with separately injected faults $f1$ and $f2$; and the HOM with both faults combined, denoted with $\{m1, m2\}$. If $f1$ and $f2$ are masking the failure of one another we see that:

$$\forall tc \in TS\,.\, isKilled_{m1}(tc) \wedge isKilled_{m2}(tc) \rightarrow \neg isKilled_{\{m1,m2\}}(tc) .$$

We observed 24 of such cases within the experimental case, which proves the existence of this phenomenon.

Effect: A test case that was originally selected for revealing failures from one of these faults might falsely pass in real operation if the program comprises both.

Instance: One robot erroneously sends a reconfiguration message while another does erroneously not. Considering these faults separately we would either observe that no reconfiguration took place although it should or that a reconfiguration was unnecessarily triggered. In combination, however, a reconfiguration takes place and offers the chance that a correct configuration is chosen.

Algorithm 1 Naive Mutation-based Test Suite Reduction

```
F : system faults which can be injected by mutation operators
TS : the original test suite
TS' ← ∅
Killed ← ∅
for all tc ∈ TS do
    K ← ∅
    for all f1 ∈ F do
        for all f2 ∈ F do
            som ← SOM comprising f1 and f2
            fom1, fom2 ← FOMs comprising f1 and f2, resp.
            for all m ∈ {som, fom1, fom2} \ Killed do
                if isKilled_m(tc) then
                    K ← K ∪ {m}
                    Killed ← Killed ∪ {m}
    if |K| > 0 then
        TS' ← TS' ∪ {tc}
return TS'
```

(2) *Failure in Combination*

Particular failures might only be visible if the program comprises two or more system faults at the same time. If so, the test suite contains test cases not revealing one of the separate faults but doing so for the combination. For the exemplary mutants $m1$, $m2$ and $\{m1, m2\}$ this means that

$$\exists tc \in TS\,.\,\neg(isKilled_{m1}(tc) \vee isKilled_{m2}(tc)) \wedge isKilled_{\{m1,m2\}}(tc)\,.$$

We observed 918 of those cases in our experiment.

Effect: If we exclusively consider FOMs, each of those cases reduces the probability to reveal the HOM in real operation. If this probability goes to zero, i.e., $\nexists tc \in TS\,.\,isKilled_{m1}(tc) \vee isKilled_{m2}(tc)$ even though $\exists tc' \in TS\,.\,isKilled_{\{m1,m2\}}(tc')$ the reduced test suite will not include a test case that covers the failure anymore.

Instance: Two robots identify environmental faults that need to be signaled. If one of the robots erroneously not sends a reconfiguration message, there might be no failure as the other robot triggers a reconfiguration though. However, if both robots do not send a reconfiguration message the failure occurs.

To bypass the mentioned unwanted effects we explicitly consider HOMs in the following. Algorithm 1 shows how this kind of mutation-based test suite reduction could be generally done on the example of second order mutants (SOM). The algorithm successively selects a number of test cases that cover all the SOMs [2]. For this particular example, we have at worst $|TS| \cdot |FOM|^2$ executions in order to investigate $isKilled_m(tc)$. Considering all possible HOMs, this can be generalized to $|TS| \cdot 2^{|FOM|}$.

3.3 Identifying Relevant HOMs

The distributed, decentralized nature of SO systems, however, gives us a short cut: The considered mutation operators *LRM*, *NRM*, and *FR* generate system faults which are *independent* from one another, as they are distributed over different entities that are only connected through message passing. This means that a fault in the source code of robot R1 does not influence the path passed through control flow in robot R2. In other words, for each of the injected system faults within a HOM there is a FOM in which the line of modified code is covered by the same test cases as for the HOM. This special case can be leveraged in order to reduce the needed number of executions before reduction by the following approach:

Algorithm 2 Mut.-based Test Suite Reduction on Relevant Faults

```
M : maps test cases to mutants comprising the relevant faults
TS : the original test suite
TS' ← ∅
Killed ← ∅
for all tc ∈ TS do
    for all m ∈ M(tc) do
        if m ∉ Killed ∧ isKilled_m(tc) then
            Killed ← Killed ∪ {m}
            TS' ← TS' ∪ {tc}
return TS'
```

(1) For each mutation operator we tag the points in code (respectively within the model) at which it could inject the faults. We might therefor introduce another predicate

$$isCovered_m \subseteq TestCases\,. \tag{2}$$

(2) Let F be a list of all the possible faults tagged in code. By execution and logging we can then determine for each test case which faults from F it would generally cover. We need $|TS|$ executions to do so for test suite TS. More concretely, we annotate each test case by a *label vector*, i.e., a binary vector v of length $|F|$, with $v[i] = 1$ if the fault $F[i]$ is covered and $v[i] = 0$ otherwise. By $x[i]$ we access the ith element of a list or a vector x.
(3) The label vectors show us which of the system fault combinations are actually relevant; and only mutants comprising those need to be considered by the reduction algorithm, as only covered faults (and patterns of them) can have an effect on the system behavior. The search for these relevant mutants can be done offline – we need no further execution.
(4) Finally, the test suite reduction algorithm only needs to consider the relevant FOMs and HOMs. We need in worst case $|TS| \cdot |Relevant\ Mutants|$ executions for reduction.

Using Algorithm 2 we can thus limit the number of needed executions before test suite reduction to $|TS| + |TS| \cdot |Relevant\ Mutants|$. We save about $|TS| \cdot (2^{|M|} - |Relevant\ Mutants|)$ executions before reduction by this short cut.

4 USING CLUSTERING TECHNIQUES

In the following, we link the mutation-based test suite reduction task with the problem of representative subset selection. Various algorithms were proposed to solve problems of this kind for general classification tasks. We show that using the previously introduced label vectors describing the test cases' ability to cover possible system faults, those algorithms select an adequate (w.r.t. killed mutants) subset of a given test suite at the cost of only $|TS|$ executions before reduction. This can save $|TS| \cdot |Relevant\ Mutants|$ executions before reduction in comparison to Algorithm 2, as test suites are

[2] Note that the resulting test suite is not necessary minimal as the algorithm does not optimize the test case selection per mutant.

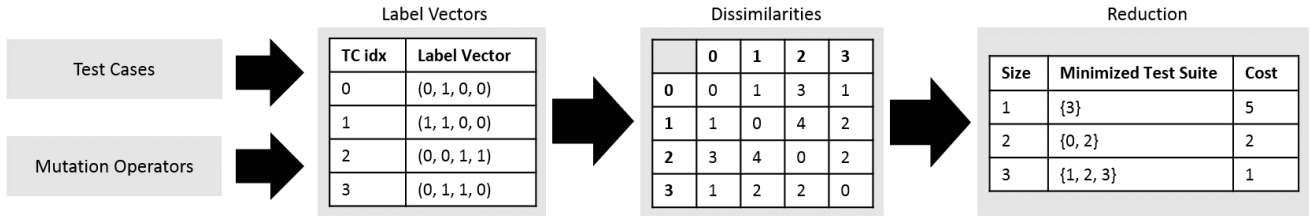


Figure 2: Proposed Process for Mutation-based Test Suite Reduction: Given a set of test cases and a set of mutation operators, first, the label vectors are determined through test case execution; second the dissimilarity matrix is constructed w.r.t. Eq. (3); third, the test suite is reduced using one of the approaches described in Section 4.1.

reduced offline exclusively considering the label vectors derived in one execution per test case. Figure 2 sketches the process that is discussed in the remainder of this section.

4.1 Selecting Representative Subsets

The problem of finding a representative subset from a huge number of models has been extensively studied in the machine learning literature. A lot of algorithms, including [1, 3, 10, 13, 17, 25], were proposed to tackle this problem in different facets. Applications include recommender systems, computer vision or language processing. To the best of our knowledge, there have been no investigations yet applying these algorithms to test suite reduction.

A prominent group of algorithms tackling the subset selection problem directly operates on measurement vectors for data that lies in low-dimensional subspaces [1, 3, 25]. Its members are instantiated only by the measurement vectors describing the data. As we however assume that the label vectors of test cases should not be interpreted as arbitrary measures in vector space and that a particular dissimilarity metric is needed, we will, in the following, concentrate on another group: algorithms selecting the subset based on pairwise similarities or dissimilarities between the data points of the original set. An adequate measure will be discussed in the next section. In concrete, our experiments are based on the *Affinity Propagation* [13] and the *Dissimilarity-Based Sparse Subset Selection* algorithm [10].

Affinity Propagation (AP). This method approaches the representative subset selection task by a message passing algorithm. Each data point is viewed as a node in a network that transmits messages along the edges updated by particular rules. By this process of sending and updating messages an energy function evolves that guides the choice for particular *exemplars*, as the authors call the elements within the representative subset.

The concrete number of output exemplars is determined during the process, which means that we cannot precisely specify the size of the representative subset in advance. By appropriately choosing the diagonal similarities, i.e., the suitability of points for representing themselves as exemplars, we can, however, approximately control the scale. Values between the minimum (few exemplars) and the maximum (many exemplars) possible similarity are common. The algorithm terminates if the cluster boundaries remain unchanged over a number *convIter* of iterations or if a threshold *maxIter* of iterations is reached.

Dissimilarity-Based Sparse Subset Selection (DS3). Elhamifar et al. formulate the representative subset selection as a *Row-Sparsity Regularized Trace Minimization Problem* [10]. Considering a convex relaxation of this formulation they propose the DS3 algorithm for solution. In [9] an implementation is presented that builds on the *Alternating Direction Method of Multipliers (ADMM).* This implementation was used for our experiments.

Like AP, also DS3 is not directly parametrized by the concrete number of representatives. Instead, a regularization parameter ρ can be set which controls the scale. A second parameter *maxIter* determines the maximum number of iterations. If an internally computed error does not fall below a threshold ϵ in $k < maxIter$ steps the algorithm terminates.

4.2 Dissimilarity Metric

Let us consider two test cases tc_1 and tc_2 with label vectors v_1 and v_2; and let us assume we would represent tc_2 by tc_1 which means that we exclusively execute tc_1. The number of mutants we could possibly miss by not executing tc_2 would depend on $|\{l \in \{1, \ldots, n\} \mid v_2[l] = 1 \wedge v_1[l] = 0\}|$. Note that in addition to the missed FOM, we could possibly miss some HOM that would only be killable in combination with a non-covered FOM (cf. pattern (1) in Section 3.2). Contrary, each mutant exclusively covered by tc_1, but not by tc_2, could cause failure masking issues (cf. pattern (2) in Section 3.2) and thus increase the chance of missing HOMs again. This can be estimated by $|\{l \in \{1, \ldots, n\} \mid v_1[l] = 1 \wedge v_2[l] = 0\}|$.

Combining these considerations we get the following dissimilarity measure for test cases based on their label vectors, which can be interpreted as comparing two bit patterns by the *Hamming distance* [14]:

$$Dist(v_1, v_2) = |\{l \in \{1, \ldots, n\} \mid v_1[l] \neq v_2[l]\}| \,. \tag{3}$$

This can be transformed to a similarity measure by:

$$Sim(v_1, v_2) = 1 - Dist(v_1, v_2)/n \tag{4}$$

Using these metrics and an enumerable test suite TS we can generally construct a $|TS| \times |TS|$ similarity matrix S as well as a dissimilarity matrix D with the same dimensions by $D[i][j] = Dist(TS[i].v, TS[j].v)$ and $S[i][j] = Sim(TS[i].v, TS[j].v)$ for all $0 \leq i, j \leq |TS| - 1$, where $tc.v$ denotes the label vector v of test case tc. These matrices serve as inputs for the subset selection algorithms.

4.3 Application on Case Study

Up to here we assumed for the sake of understandability that each test case can be executed at any time without any dependencies. This assumption, however, does not generally hold. In particular, the test inputs derived for our case study depend on one another. In order to cope with these dependencies we did some additional pre-processing: First, we constructed a directed *dependency graph* associating test cases with matching pre- and post conditions (i.e. particular system configurations). For the experimental case described in Section 2 this led to a graph with 7524 nodes and 8 884 634 edges. Second, we sampled a manageable set of *test sequences*: A test sequence is formed by a path through the dependency graph. Because of the huge number of possible paths and test sequences, which in fact is unfeasible for the subset selection algorithms (at least for our computing capacity), we based our experiments on 500 sequences generated by random walk. The dissimilarities between test sequences π_1 and π_2 were determined by:

$$Rep(\pi_1, \pi_2) = \sum_{tc_2 \in \pi_2} \min_{tc_1 \in \pi_1} Dist(tc_1.v, tc_2.v) \tag{5}$$

Note that Eq. (5) is not symmetric and hence violates the properties of a metric. It quantifies, how well π_2 is represented by π_1. Both considered algorithms for representative subset selection can cope with this viewpoint, however.

Using Eq. (5) and mutation operators *LRM*, *NRM* and *FR* (cf. Section 3.1) the matrices D and S were constructed. For assessing the degree of optimization w.r.t. the given similarity/dissimilarity notion we used the following empirically derived parametrization:

- *AP*:
  - *pref*: equally drawn from $[m-2, m+2]$ where m is the median of similarities in S.
  - *convIter*: 15
  - *maxIter*: 3000
- *DS3*:
  - ρ: equally drawn from $[0.001, 0.005]$.
  - *maxIter*: 3000.
  - ϵ: 10^{-7}.

We additionally applied a uniform sampling of k test sequences for each k proposed by the AP or DS3 serving as a baseline. Building on the previous considerations we evaluated the reduced test suites $TS' \subset TS$ by a *representation cost measure* :

$$\sum_{tc \in TS} \min_{tc' \in TS'} Dist(tc.v, tc'.v). \tag{6}$$

The left plot in Figure 3 presents the results for the AP, the DS3 and the uniform samples on our case study. As we can see, both classifiers are more or less on the same level; they outperform the uniform samples regarding the representation cost. The right graphic, which depicts the actual number of killed mutants (FOMs and SOMs), suggests that optimizing the representation cost also optimizes the number of actually killed mutants. We see that test suites reduced by the classifiers kill more mutants than the uniform samples for the same k. Even if not all of the killable mutants (6245 in our experiment) are actually killed, the DS3 algorithm got 91.3 % of them with 62 paths including 247 test cases.

Still, we have to note that the choice between the exhaustive Algorithm 2 and one of the classifiers keeps a trade-off between minimizing the number of executions before reduction (classifiers) and actual mutation coverage (exhaustive algorithm).

5 RELATED WORK

The challenge of reducing a test suite is finding a subset of the comprised test cases that maintains given test requirements. Those requirements are mostly defined by means of coverage criteria. Existing research on test suite reduction can be classified by (1) the type of coverage criterion and (2) the quality measure they use for reduced test suites [24]. While it seems quite common to use fault-based techniques for (2) ([2, 12, 22, 23, 26]), for (1) the huge majority of approaches rather applies code coverage criteria, such as statement-, branch- or decision coverage ([2, 16, 22, 26]).

We found two exceptions that also based (1) on mutants, similar as we do: Offut et al. [19] propose a procedure for mutation-based test suite reduction called *ping-pong*. This applies elaborated heuristics in order to efficiently find a subset of test cases that kills all the mutants that were killed by the original set, too. They show, that they can reduce mutation-based test sets by over 30%. In contrast to us they exclusively consider FOMs. As they show, the introduced heuristics reduce the number of test cases to execute before reduction. Still, during the search for the reduced set, each chosen test case has to be executed one time per mutant to kill. If they could use the label vectors we proposed, they could save executions in a similar way as we do. However, they actually cannot, as the faults they consider are not independent from one another.

Shi et al. [24] evaluate classical test suite reduction algorithms on requirements called *Mutant Adequate Reduction* (MAR) and *Statement-Mutant Adequate Reduction* (SMAR). While the first exclusively considers killed mutants the second combines this with code coverage. They use the mutation tool PIT for evaluation on open-source projects. Again, exclusively FOMs are considered and again, label vectors cannot be used for reducing the number of test cases to execute before reduction.

To the best of our knowledge no study considers test suite reduction for the particular system class of self-organizing systems. In [8], Eberhardinger et al. addressed the related task of test case selection. They present a couple of strategies which leverage particular characteristics of self-organizing systems and test oracles in order to speed up a search based approach on test input generation. In contrast to that, the present paper is concerned with reducing an already existing set of test cases whose elements are being executed at least once.

6 CONCLUSION

We discussed a mutation-based approach on test suite reduction for self-organizing systems. It takes first- as well as higher-order mutants into account that simulate the effect of communication faults during the reconfiguration process. The use of established clustering techniques for representative subset selection showed promising results w.r.t. our experimental case. Possible directions of future work include the analysis of the impact of outliers on the overall representation cost by a reduced test suite. A more detailed investigation of adequate mutation operators for self-organizing systems has to be done. According to our previous ideas about risk-based interoperability testing, mutants might be weighted by

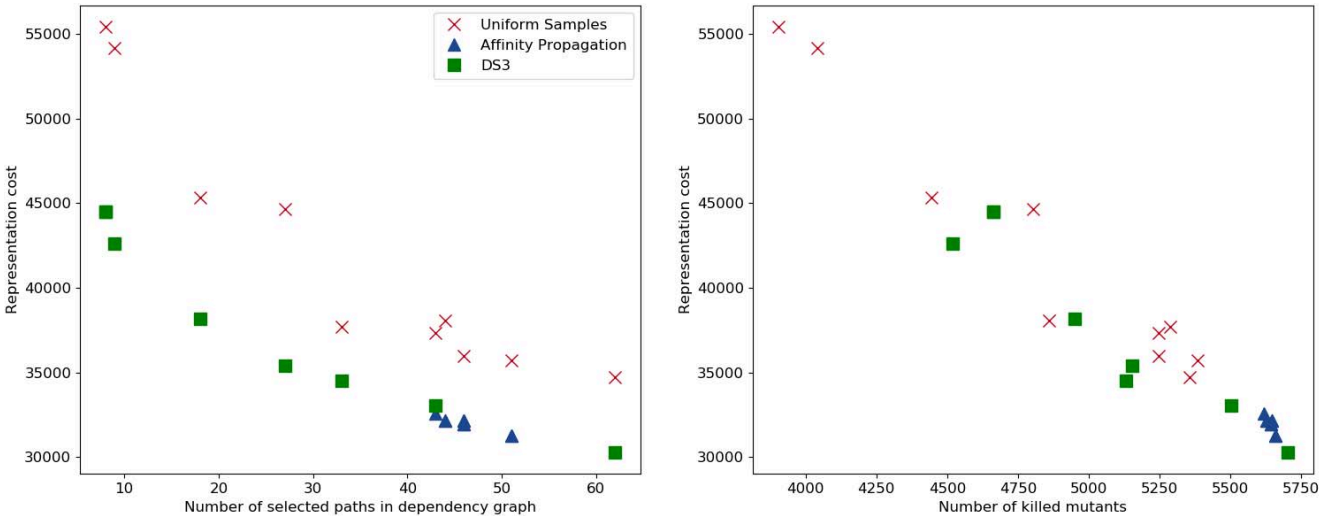


Figure 3: Results of the Mutation-based Test Suite Reduction using the classifiers.

a relevance score for prioritization [20]. In general, we notice that techniques from the area of artificial intelligence (AI) might have useful, still unexplored applications in testing. The other way round also AI approaches need to be tested. Thus, we will go on to investigate both of these sides by testing AI with AI.

ACKNOWLEDGMENTS

This research is sponsored by the research project *Testing Self-Organizing, Adaptive Systems (TeSOS)* of the German Research Foundation.

REFERENCES

[1] Laura Balzano, Robert Nowak, and Waheed Bajwa. 2010. Column subset selection with missing data. In *NIPS workshop on low-rank methods for large-scale machine learning*, Vol. 1. Citeseer.
[2] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 106–115.
[3] Christos Boutsidis, Michael W Mahoney, and Petros Drineas. 2009. An improved approximation algorithm for the column subset selection problem. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 968–977.
[4] Rogério de Lemos et al. 2013. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.). LNCS, Vol. 7475. Springer, 1–32.
[5] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.
[6] Benedikt Eberhardinger, Gerrit Anders, Hella Seebach, Florian Siefert, Alexander Knapp, and Wolfgang Reif. 2017. An approach for isolated testing of self-organization algorithms. In *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 188–222.
[7] Benedikt Eberhardinger, Axel Habermaier, Hella Seebach, and Wolfgang Reif. 2016. Back-to-back testing of self-organization mechanisms. In *IFIP International Conference on Testing Software and Systems*. Springer, 18–35.
[8] Benedikt Eberhardinger, Hella Seebach, Dominik Klumpp, and Wolfgang Reif. 2017. Test Case Selection Strategy for Self-Organization Mechanisms. In *Test, Analyse und Verifikation von Software – gestern, heute, morgen*. dpunkt, 139–157.
[9] Ehsan Elhamifar, Guillermo Sapiro, and S Shankar Sastry. 2016. Dissimilarity-based sparse subset selection. *IEEE transactions on pattern analysis and machine intelligence* 38, 11 (2016), 2182–2197.
[10] Ehsan Elhamifar, Guillermo Sapiro, and Rene Vidal. 2012. Finding exemplars from pairwise dissimilarities via simultaneous sparse recovery. In *Advances in Neural Information Processing Systems*. 19–27.
[11] Stephanie Forrest. 1990. Emergent computation: Self-organizing, collective, and cooperative phenomena in natural and artificial computing networks. *Physica D: Nonlinear Phenomena* 42, 1 (1990), 1–11.
[12] Gordon Fraser and Franz Wotawa. 2007. Redundancy based test-suite reduction. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 291–305.
[13] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.
[14] Richard W Hamming. 1950. Error detecting and error correcting codes. *Bell Labs Technical Journal* 29, 2 (1950), 147–160.
[15] Yue Jia and Mark Harman. 2008. Constructing subtle faults using higher order mutation testing. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. IEEE, 249–258.
[16] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering* 29, 3 (2003), 195–209.
[17] Leonard Kaufman and Peter Rousseeuw. 1987. *Clustering by means of medoids*. North-Holland.
[18] A. Jefferson Offutt. 1989. The coupling effect: fact or fiction. In *ACM SIGSOFT Software Engineering Notes*, Vol. 14. ACM, 131–140.
[19] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. 1995. Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software*.
[20] André Reichstaller, Benedikt Eberhardinger, Alexander Knapp, Wolfgang Reif, and Marcel Gehlen. 2016. Risk-based interoperability testing using reinforcement learning. In *IFIP International Conference on Testing Software and Systems*. Springer, 52–69.
[21] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. 2006. Towards a generic observer/controller architecture for Organic Computing. *GI Jahrestagung (1)* 93 (2006), 112–119.
[22] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.
[23] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.
[24] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 246–256.
[25] Joel A Tropp. 2009. Column subset selection, matrix factorization, and eigenvalue optimization. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 978–986.
[26] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An empirical study of JUnit test-suite reduction. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 170–179.