International Journal of Cooperative Information Systems
Vol. 26, No. 1 (2017) 1740002 (36 pages)

DOI: 10.1142/S0218843017400020

Automatic Business Process Test Case Selection: Coverage Metrics, Algorithms, and Performance Optimizations

Kristof Böhmer* and Stefanie Rinderle-Ma†

Faculty of Computer Science
University of Vienna
Währinger Strasse 29, 1090 Vienna, Austria
**kristof.boehmer@univie.ac.at*
†*stefanie.rinderle-ma@univie.ac.at*

Accepted 11 November 2016
Published 16 December 2016

Business processes describe and implement the business logic of companies, control human interaction, and invoke heterogeneous services during runtime. Therefore, ensuring the correct execution of processes is crucial. Existing work is addressing this challenge through process verification. However, the highly dynamic aspects of the current processes and the deep integration and frequent invocation of third party services limit the use of static verification approaches. Today, one frequently utilized approach to address this limitation is to apply process tests. However, the complexity of process models is steadily increasing. So, more and more test cases are required to assure process model correctness and stability during design and maintenance. But executing hundreds or even thousands of process model test cases lead to excessive test suite execution times and, therefore, high costs. Hence, this paper presents novel coverage metrics along with a genetic test case selection algorithm. Both enable the incorporation of user-driven test case selection requirements and the integration of different knowledge sources. In addition, techniques for test case selection computation performance optimization are provided and evaluated. The effectiveness of the presented genetic test case selection algorithm is evaluated against five alternative test case selection algorithms.

Keywords: Process modeling and design; process testing; test case selection; genetic algorithm; optimization.

1. Introduction

The application and implementation of process technology provides enterprises with substantial advantages, as it has been recently shown by Reijers *et al.*[1] Over the past years, processes have risen to deeply integrated solutions connecting organizational, functional, and operational aspects.[2] Moreover, an increasing portion of the current processes tend to become fully automated, see Ref. 3. This leads to an increased dependence on the correct execution of current business process models. A critical failure in the execution of an arbitrary automated process can, for example, remain undetected for a long time. This can cause negative effects on the fragile bond of trust between customers and cooperations.[4]

Hence, ensuring the stability and correctness of processes during design and execution time is a crucial challenge.[5] Several approaches for process verification exist, that focus on structural and behavioral correctness of process models.[6] Specifically, when implementing process models, *testing* has proven a valuable complement to capture the process behavior at runtime, for example, with respect to process data.[7] Testing concentrates on creating and executing test cases on the tested process model.[8] At minimum, a test case consists of input data, which is used to initialize a new instance of the process under test, and an expected execution path. The expected path should be followed by the process model instance when executing the test case.[7] A fault can be detected, for example, when an execution path deviates from the expected test case execution path.[7]

Testing plays an important role in process model design, development, and maintenance because it allows to identify faults early during these phases.[7] As process models tend to become more and more complex, manual test case generation becomes time-consuming and, therefore, expensive.[9] Hence, automatic test case generation tools emerged. These can quickly generate hundreds or even thousands of test cases to completely test a single process model.[10]

Each individual test case might be executed quickly. However, executing *all* test cases may still require an excessive amount of time and, therefore, results in high costs.[11] Hence, it becomes necessary to apply test case *selection* and *minimization* techniques. Those techniques select an appropriate[a] subset of the available test cases to be executed. If the subset is small and efficient enough, significant time-savings can be achieved while user defined test requirements are still satisfied.[11,12]

Take, for example, the process model shown in Fig. 1. It is tested by three test cases T1, T2, and T3. Assume that T1 covers the top most path, T2 the middle path, and T3 the path on the bottom. Assume further that a user requirement is to cover 75% of the process model (i.e. 75% node coverage, so 75% of all nodes are tested by test cases).[b] Then possible subsets would be to select T1 and T2 (combined

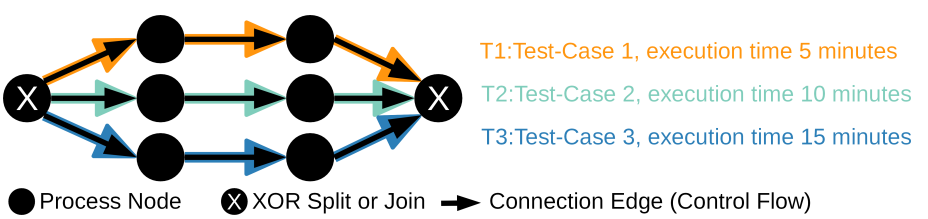


Fig. 1. Test case selection example with coverage illustration.

[a]Appropriate means that user-defined requirements are fulfilled, such as, a minimal coverage objective, for example, that a minimal amount of process nodes is tested.
[b]Multiple coverage metrics exist such as path, branch, or node coverage. However, in this paper we will, for the sake of brevity and simplicity, only consider *node coverage*. However, we are confident that a generalization to other coverage metrics is possible — a discussion is available in Sec. 7.

execution time 15 min),[c] T2 and T3 (25 min), or T1 and T3 (20 min). However, imagine, that the selected test cases should be executed in the minimum possible amount of time, while reaching at minimum 75% node coverage requirement. Then the selection technique must select T1 and T2 as the optimal subset.

Identifying an *optimal* test case subset results in a combinatorial explosion problem (the complexity is exponentially related to the amount of test cases).[11] Hence, it cannot be solved in polynomial time.[13] So, existing approaches utilize heuristics, such as the Greedy Algorithm. Heuristics enable to find solutions where analytical algorithms are infeasible because of the huge search space.[11]

We have analyzed existing process model test case selection and minimization approaches. It was found that those are *inflexible* regarding the supported user-defined coverage requirements. Moreover they only use an *incomprehensive* representation of each node's unique coverage requirements. In addition, they model the coverage effects of each test case in a *limited* fashion. Hence, existing work is not suitable for answering the following research questions:

RQ1 How can node coverage effects for process model test cases be modeled in a more comprehensive way?

RQ2 How can the unique coverage requirements of each process node be determined and utilized during process model test case selection?

RQ3 How to design an algorithm which flexibly supports complex process model test case selection requirements? For example, to optimize the selected test cases based on their execution time.

The above questions have been subject for investigation in a previous conference paper, see Ref. 14. This work raises research questions that aim specifically at the effectiveness and efficiency of the presented test case selection algorithm.

RQ4 How to evaluate the effectiveness of the presented selection algorithm in comparison with random, greedy, hill climbing, sequential backward, and simulated annealing-based selection algorithms? (extended from Ref. 14).

RQ1 Are test execution performance optimizations conceivable, specifically in the context of evolving processes? How can the performance gains be measured? (new).

In order to tackle research questions **RQ1** and **RQ2**, novel and extended coverage metrics are provided. These enable to exploit process model information and characteristics that have not been considered so far, but might be relevant for test case selection. For example, how each node's coverage, in a process model, is affected by its neighborhood nodes (e.g. nodes that are, for example, directly connected) and their type (e.g. if a neighborhood node is a gateway). Moreover, it is assumed that different kinds of nodes in a process model may have a different complexity, for

[c]The expected execution time can, for example, be extracted from recorded execution logs, see Sec. 3 for a detailed discussion.

example, tasks versus gateways. This is exploited to deduce if a node must be tested more or less thoroughly.[15] So this work provides a comprehensive representation of test case coverage effects. In addition, a novel approach is provided to identify the unique coverage requirements of each individual process node.

The investigation of **RQ3** leads to the design of a genetic process model test case selection algorithm. The algorithm enables to flexibily work with different coverage metrics, fitness functions, and process model complexities. The effectiveness of the algorithm has been evaluated against random and greedy selection.[14] Here a comparison with advanced simulated annealing, sequential backward selection, and hill climbing-based business process test case selection techniques is added. This leads to a more comprehensive understanding of the effectiveness of the proposed algorithm (**RQ4**). Finally, **RQ5** demands for optimization concepts to avoid complete recalculations of test case selections in case a process model evolves. The latter is often the case for many reasons, such as, legislation changes.[16,17] The achieved performance gain is evaluated by comparing the performance of the presented algorithms with and without the proposed optimization.

This paper is organized as follows. Prerequisites and coverage metrics are discussed in Sec. 2. Section 4 proposes a performance optimization for the genetic business process model test case selection approach presented in Sec. 3. The evaluation, corresponding results and their discussion are presented in Sec. 5. Section 6 discusses related work. Conclusions and future work is given in Sec. 7.

2. Coverage Metrics

This section introduces coverage metrics for test case selection based on a given process model O. O is defined as directed graph $O := (N, CE, DE)$[d] where N denotes the set of process nodes, CE the set of control flow edges, and DE the set of data flow edges. As auxiliary functions (cf. Ref. 19), we utilize the direct successors of a node n as $n\bullet := \{n' \in N \,|\, (n, n') \in CE\}$ for the control flow and $n\circ := \{n' \in N \,|\, (n, n') \in DE\}$ for the data flow. The direct predecessors of n can be defined accordingly by $\bullet n := \{n' \in N \,|\, (n', n) \in CE\}$ for the control flow and $\circ n := \{n' \in N \,|\, (n', n) \in DE\}$ for the data flow.

In this paper, we are mainly interested in the execution path of each test case. This is because the execution path enables to determine which process model nodes are covered (i.e. tested) by each test case. So a test case is formally defined as:

Definition 2.1 (Test case). A test case v on a process model $O = (N, CE, DE)$ is defined as $v := (N_v, CE_v, enabled)$ with $N_v \subseteq N$, $CE_v \subseteq CE$, and $enabled \in \{0, 1\}$. N_v and CE_v form the expected test case execution path and *enabled* indicates if the test case should be executed (1) or not (0). The ordered set V of all test cases v on a process model O is denoted as test suite. It can be configured to create a

[d]The notion of directed graphs corresponds to the internal representation in order to cover different prevalent process modeling notations such as BPMN.[18]

test suite configuration V_i where all test cases are enabled or not (i.e. $v.enabled$ is set to 1 or 0).

Note, this work does not concentrate on test case generation, test oracles, failure models, or related technical testing aspects. This is because this paper only concentrates on selecting, from a given set of all test cases that verify a given process model, a subset of test cases which fulfills given user requirements. For this it is expected that a predefined set of manually defined or automatically generated test cases is already available. These test cases are expected to be capable of verifying the process model execution behavior. For example, by checking for non-responding services or expected variable states.[4]

Moreover, for this paper an abstracted basic test case definition that only contains the execution path of each test case is sufficient. Hence, a test case is aware of the activities, gateways, etc. that are covered, and hereby verified, during its execution. The test case coverage information is in the following exploited to select test cases that fulfill user-defined process coverage requirements, for example, to select, from a set of all test cases, a subset of test cases that covers 80% of all process model nodes in minimal test case execution time.

Test case selection starts with a test suite V (consisting of all available test cases for the process model O) and a set of requirements R. The requirements R are user-defined and specify that, for example, the minimal node coverage should be 75%. The requirements contained in R must be satisfied to find a test suite configuration which provides an adequate testing of the process. Therefore, it must be decided which test cases should be executed when executing the test suite. Hence, the challenge is to find a minimal subset $V' \subseteq V$ that satisfies all requirements in R.

One typical requirement is that the process must be completely covered (i.e. each node must be tested by the selected test cases). For this, mostly, simple coverage metrics are used. For example, a process node is already marked as completely covered, and therefore, fully tested, when it is checked by at least one test case. However, this approach ignores that each process node has a *unique* complexity and significance (cf. Ref. 15). So we propose that each node should be covered by an individually adjusted number of test cases to achieve an optimal coverage.

2.1. *Optimal coverage*: *Optimal number of test cases per node*

This approach is called optimal coverage because it determines an optimal coverage value (i.e. how many test cases should be used to test it) for each node. Therefore, it applies, individually for each process node, various complexity metrics. We assume that if a process node (e.g. an activity or gateway) is more complex than another one, it must be tested more thoroughly (i.e. covered by more test cases). Hence, it must get assigned a higher optimal coverage value. We suggest to determine the optimal coverage value $C_o(j)$ of node j as the weighted sum over selected complexity

metrics $comp_i(j)$ for node j, $(i = 1, \ldots, n)$:

$$C_o(j) := 1 + \left\lceil \sum_{i=1}^{n} w_i \cdot comp_i(j) \right\rceil. \tag{2.1}$$

In Eq. (2.1), $w_i \in [0, 1]$ defines the weight for metric $comp_i(j)$. Moreover, a minimal coverage of 1 is assigned to each node, i.e. each node must be covered by at least a single test case. The complexity metrics and the weights reflect the process node coverage requirements. One example for a complexity metric is the Fan-In/Fan-Out metric (cf. Ref. 20): for node j it sums over the number of successors $|j \bullet|$ and predecessors $|\bullet j|$ of node j and divides this sum by the maximum Fan-In/Fan-Out value over all nodes of the process model.

Two types of metrics are considered in this paper. First, *generic metrics* that are based on the process model. These metrics incorporate the *node complexity* (the structural *Fan-In/Fan-Out metric*), the *process structure* (a node is positioned in sequences or more complex loops, error, or concurrent paths), or the *node position* (a fault at an early executed node affects more follow up process nodes than a fault at a late node). Second, metrics that are supported by *historic data* (e.g. log files). Such metrics are *previously identified faults* (it is then more likely to find another fault), *node execution frequency* (an fault does have an higher effect if the faulty node is executed more frequently), *previous coverage* (if a node was not covered during previous tests, then it should be checked during follow up tests). Moreover, historic data enables to determine the *error path probability* (if a node frequently has to fall back to its error path it more likely contains a fault), *frequency of data-modifications* (we assume that a node which modifies multiple variables has likely a higher internal complexity than a node which modifies only one variable), and *known changes* (if a process node is changed then those changes should be checked with tests).

Example: Node j has three incoming edges and one outgoing edge. It is analyzed using the Fan-In/Fan-Out metric[e] with a weight of 1. Further, the node with the maximum Fan-In/Fan-Out metric of the whole process model has four incoming and two outgoing edges. Then $comp_{fan}$ would generate the following result:$(1 + 3)/(2 + 4) = 0.6\overline{6}$. The total optimal coverage can then be calculated by $C_o(j)$ using $1 + \lceil (1 \cdot 0.6\overline{6}) \rceil = 2$, i.e. two test cases are required to thoroughly test j, when considering its complexity. Existing approaches would ignore the node complexity and hence test it with a single test case. This could result in not detecting faults that will be found by the proposed approach. Why? Because, each node has a specific *internal node behavior*[f] which can, for example, contain multiple execution branches. Imagine that the internal node behavior contains a single conditional

[e]This example only utilizes, for the sake of brevity, the Fan-In/Fan-Out metric. The test case selection prototype (cf. Sec. 5), however, uses all the mentioned metrics (see previous paragraph).
[f]Note, each node's functionality is determined by its internal behavior (e.g. realized as a web-service or application) that is executed when executing the process node.

branch which provides two execution paths (e.g. for premium or normal customers). Then a single test case will most likely only test one of the branches so 50% of the node's internal behavior and it will require at least two test cases to thoroughly test the node. Note that we are assuming that the mentioned complexity metrics also allow to assess the nodes internal complexity. For example, a node with many incoming edges most likely has a more complex internal behavior than a node with only one incoming edge.

The presented approach focuses on process test cases which execute and verify whole process execution paths. However, one may argue that it is more efficient to test each process resource independently from the process model itself. This is the case *iff* each resource is fully isolated and deterministic (e.g. if only strictly isolated webservices are used). However, business process models tend to integrate a wide range of interconnected resources, such as, legacy applications or Enterprise-Resource-Planning software. Hence, we found that not all the resources that are utilized by the current processes are isolated. So an access on resource **R1** by an early activity **A1** can affect the behavior and state of another resource **R2** that is called by activity **A2**. In such a case, **R2** will behave differently whether or not **A1** was executed previously. Hence, a fault in **R2** can be triggered or not depending on the execution of **A1**.

Existing work indicates that webservices also partly fail to deal with concurrent access patterns that can stem from parallel process execution paths. Here, related faults can also be identified by process model tests in an efficient manner.[21,22] In addition, we found that the current process execution engines enable to integrate executable code snippets directly in a process model. Those snippets are executed without the need to invoke external resources and can access various process properties (e.g. variables or process execution states). This deep integration makes it difficult to test them independently from the process model but they can be tested by process test cases.

Moreover, the — possibly hidden — interactions of each resource are diverse. So, testing each theoretically possible interaction and mutual impact of each resource results in an enormous testing effort. Hence, we see process execution paths and the related test cases as a way to identify the most important resource access patterns. This enables, in the following, to reduce the overall testing effort.[22] This is also reflected by the calculated optimal coverage heuristic. It represents, for each process model node, an artificial importance to verify its correctness with process test cases. Therefore, the optimal coverage calculation takes multiple aspects into account. These aspects range from data artifacts to historical information (e.g. if faults were identifies for an activity recently). Overall, we conclude that *both* testing approaches are important and must be applied to thoroughly test processes and related resources. That is, the testing of each individual resource based on fine granular test cases and coarse granular process execution-based tests — this work focuses on the later.

2.2. *Test coverage metrics*: *Coverage of all enabled test cases*

The following coverage calculation approaches are applied individually on each process node j. They determine, given a test suite configuration V_i and a process model O, which test coverage is achieved by V_i on j.

Traditional Coverage: The *traditional coverage* cov_{tr} is based on existing coverage calculation approaches. It is calculated for a process node j and a test suite configuration V_i by analyzing the test paths of each enabled test case $v = (N_v, CE_v, 1)$ cf. Eq. (2.2).

$$cov_{tr}(j, V_i) := \sum_{v=(N_v, CE_v, 1) \in V_i} count_{tr}(v, j). \tag{2.2}$$

If j is covered by an enabled test case (i.e. it is contained in $j \in N_v$ of $v = (N_v, CE_v, 1)$), its coverage value is increased by one, cf. Eq. (2.3).

$$count_{tr}(v, j) := \begin{cases} 1 & \text{if } j \in N_v, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

Neighborhood Coverage: Neighborhood coverage reflects that, in a process, each node depends on its predecessors. Hence, if the predecessor of a node j is faulty then j might will never be executed (e.g. the process might terminate because of a fault before reaching j) or has to deal with incorrect data/states. We propose that this should be reflected by increasing the optimal coverage because the complexity increases if multiple predecessors can affect a single node.

Moreover, we assume that each test case that is executed on a predecessor of j also has a slightly positive effect on j itself. Therefore, j's coverage value should be slightly increased if one of its predecessors is tested. Hence, we are proposing to calculate the individual neighborhood coverage of each process node and combine it with its respective traditional coverage. Hereby a comprehensive representation of each test case's positive effects is provided.

This (a) motivates the test case selection algorithm to select test cases which together achieve a broad coverage of functionality supported by the process model under test. In addition, (b) it reflects the positive effects of each test case more comprehensively during test case selection. Why (a)? Because with neighborhood coverage the proposed test case selection algorithm gains less additional total coverage from covering close paths (i.e. paths that all concentrate on one function) than without neighborhood coverage. Hence, it is additionally motivated to cover paths (and therefore functions) which are more diverse and further apart from each other. Both advantages increase the probability that test cases selected by the proposed approach will more likely detect faults than test cases selected by existing approaches.

The neighborhood coverage value for a node j is calculated by analyzing each enabled test case. This is $\forall\, v = (N_v, CE_v, 1)$ over process model $O = (N, CE, DE)$

to identify the *neighborhood path start nodes* $N_{NPS,v}$ by, cf. Algorithm 1:

$$N_{NPS,v} := \{k \in N_v \,|\, \exists\, p \in N \backslash N_v \text{ with } k \in \bullet p\}. \tag{2.4}$$

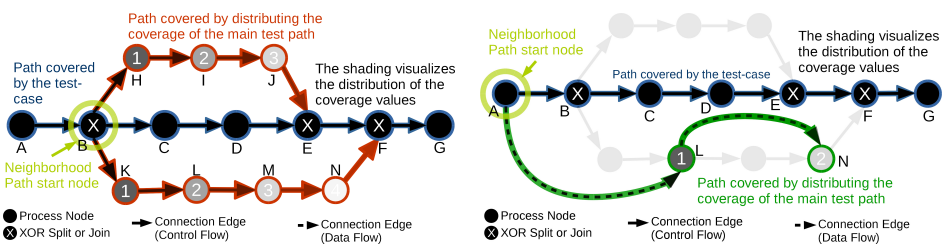
Neighborhood path start nodes are nodes that are covered by a test case v, but also have direct successors that are *not* covered by v. Subsequently, all identified neighborhood path start nodes are analyzed, i.e. $\forall\, s \in N_{NPS,v}$, to determine all direct successors which are not covered by v using

$$N_{FNN,s,v} := \{a \in N \,|\, a \in s \bullet \wedge a \notin v \cdot N_v\}. \tag{2.5}$$

Finally, the successors of all nodes in $N_{FNN,s,v}$ are searched for j to calculate j's neighborhood coverage (cf. Algorithm 2).

Example: Consider Fig. 2(a) with test suite configuration V_i which contains a single[g] enabled test case v with $N_v = \{A, B, \ldots, F, G\}$. Obviously, B is the only neighborhood path start node, i.e. $N_{NPS,v} = \{B\}$. In turn, B results in the set $N_{FNN,B,v} = \{H, K\}$. So H and K are situated in a neighborhood path to the path covered by test case v, but are not covered by v themselves.

Assume that the neighborhood coverage is to be determined for node L. For this, all successors of H and K are searched until L is found or the search reaches a node which is covered by v. During the search, the nodes on the "search paths" are numbered consecutively (using counter c). The number indicates the number of edges or respectively the distance between L and the neighborhood path start node B. The greater the distance, the less the neighborhood coverage. This is expressed by a coverage reduction factor cov_{red}. It indicates how quickly the positive effect of the test case v is reduced when getting further away from nodes covered by v. For $cov_{red} = 0.2$, a node numbered with 1 would be assigned 0.8 of the traditional coverage effect of the neighborhood start node. Here it is assumed that the traditional coverage of v on the neighborhood path start node is always 1. Imagine that j (so $j = L$) is the node marked with a 2, i.e. the node that is two



(a) Utilizing the control flow (b) Utilizing the data flow

Fig. 2. Illustrating the concept of neighborhood coverage.

[g]Note, if a test suite contains multiple enabled test cases then the neighborhood coverage is calculated individually for each test case v on j. Subsequently, each individual neighborhood coverage effect of each test case is added up to calculate V_i's total neighborhood coverage effect on j.

"steps" far from the *neighborhood path start node*, so $c = 2$. Then the control flow neighborhood coverage effect on L can be calculated by $\max((1-(2\cdot 0.2)),0) = 0.6$ when using a cov_{red} of 0.2.

Algorithms 1 and 2 focus on the process control flow. Neighborhood coverage can also refer to the process data flow denoted by the data flow edges DE of a

Algorithm 1. Neighborhood coverage calculation (pseudo code).

Algorithm `NeibCov`(V_i, j, cov_{red})
- cov_{nbh}=0
- **Data:** process node j to be analyzed, test suite configuration V_i, and coverage reduction factor cov_{red}
- **Result:** the neighborhood coverage cov_{nbh} of j
- **foreach** *test case* $v \in V_i$ *with* $v.enabled = 1$ **do**
  - **foreach** *neighborhood path start node* s *in* $N_{NPS,v}$ **do**
    - **foreach** $fs \in N_{FNN,s,v}$ **do**
      - cov_{nbh}+=`NeibCovRecur`$(j, s, fs, v, 1, cov_{red})$ `/* adds up the achieved neighborhood coverage of each` v `on` j `*/`
    - **end**
  - **end**
- **end**
- **return** cov_{nbh}

Algorithm 2. Neighborhood coverage calculation subroutine (pseudo code).

Recursive Subroutine `NeibCovRecur`(j, s, n, v, c, cov_{red})
- **Data:** process node to search for j, current neighborhood path start node s, current analyzed node n, analyzed test case v, step counter c, and a coverage reduction factor cov_{red}
- **Result:** cov_{nbh} of j for process model branch starting with s
- **if** $j = n$ (*i.e. the searched node* j *is found*) **then**
  - **return** $\max((1-(c\cdot cov_{red})),0)$`/* calculate the neighborhood coverage */`
- **else if** $n \in N_v$ (*i.e. the currently analyzed node* n *is covered by* v) **then**
  - **return** 0`/* stop the search for this branch */`
- **else**
  - $c = c + 1$`/* increase the step counter by one */`
  - **foreach** $n^* \in n\bullet$ (*i.e.* $n^* \in n\circ$ *for the data flow*) **do**
    - **return** `NeibCovRecur`$(j, s, n^*, v, c, cov_{red})$`/* recursively analyze all successive branches */`
  - **end**
- **end**
- **return** 0

process model $O = (N, CE, DE)$. The data flow based approach uses different sets to determine the neighborhood path starts nodes. Hence, they are based on the data flow edges DE instead of the control flow edges, i.e. $DN_{NPS,v} := \{k \in N_v \,|\, \exists p \in N \backslash N_v \text{ with } k \in \circ p\}$ and $DN_{FNN,s,v} := \{a \in N \,|\, a \in s \circ \wedge a \notin v \cdot N_v\}$. In Fig. 2(b), hence, $DN_{NPS,v} = \{A\}$ and $DN_{FNN,A,v} = \{L\}$ hold. Based on these sets, Algorithms 1 and 2 can be used analogously. In the example depicted in Fig. 2(b), starting from A nodes L and N are successively numbered with 1 and 2, respectively, as they are connected via data edges. Assume a reduction factor of $cov_{red} = 0.2$. Then the neighborhood coverage $cov_{nbh}(L)$ of v turns out as 0.8 and for N as 0.6 respectively based on test case v.

Coverage Degeneration: Imagine that multiple test cases are applied on the same process node. Then partly similar *internal node behavior* (cf. Sec. 2.1) is likely executed and, therefore, tested by multiple test cases. We assume that the individual *positive effect* of an additional test case depends on the amount of test cases which already cover a node. Hereby the positive effect is defined as the likelihood that a test case detects a not yet identified fault. Hence, it is higher when, e.g. a node is currently only covered by two test cases as it would be if the same node is already covered by ten test cases. So, we advocate to slightly decrease the additional coverage gain of each test case if multiple test cases are covering the same process node. We denote this by the term *coverage degeneration* which is captured by a coverage degeneration factor that is determined for each coverage metric.

The degeneration factor for the *traditional coverage* $cov_{tr}(j, V_i)$ of a node j is calculated based on the test suite configuration V_i (cf. Eq. (2.2)). Initially, the degeneration factor (cf. Eq. (2.6)) is identified by weighing the number of enabled test cases with a user-defined factor $w_{deg} \in [0, 1]$. Subsequently, it is put into relation with the maximum possible degeneration factor $w_{degMax} \in [0, 1]$ to limit the maximum degeneration. Assume that 10 enabled test cases cover j, $w_{deg} = 0.05$, and $w_{degMax} = 0.3$. Then the coverage degeneration factor for traditional coverage turns out as $1 - \min(((10 - 1) * 0.05), 0.3) = 0.7$. Hence, the achieved traditional coverage will be multiplied with 0.7 to reduce it by 30% from 10 (+1 for each test case) to 7.

$$cov_{tr}^{deg}(j, V_i) = 1 - \min(((|\{v = (N_v, CE_v, 1) \in V_i \,|\, j \in N_v\}| - 1) \cdot w_{deg}), w_{degMax}). \tag{2.6}$$

The degeneration factor of the *neighborhood coverage* metrics is also based on the number of enabled test cases. More precisely, each enabled test case in V_i is analyzed to count how many test cases generate a positive neighborhood coverage (cf. Algorithm 2) on j (cf. Eqs. (2.8) and (2.9)). Subsequently, the number of test cases is also multiplied with w_{deg}, cf. Eq. (2.7), to calculate the neighborhood coverage

degeneration factor. Again w_{degMax} limits the maximum possible degeneration.

$$cov_n^{deg}(j, V_i, cov_{red}) = 1 - \min(((cov_{nc}^{deg}(j, V_i, cov_{red}) - 1) \cdot w_{deg}), w_{degMax}), \tag{2.7}$$

$$cov_{nc}^{deg}(j, V_i, cov_{red}) = \sum_{v=(N_v, CE, 1) \in V_i} \sum_{s \in N_{NPS,v}} \sum_{fs \in N_{FNN,s,v}} count_n^{deg}(j, s, fs, v, cov_{red}), \tag{2.8}$$

$$count_n^{deg}(j, s, n, v, cov_{red}) = \begin{cases} 1 & \text{if } NeibCovRecur(j, s, fs, v, 1, cov_{red}) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The degeneration factors for the data flow (i.e. $Dcov_n^{deg}$) can be calculated analogously. Due to space restrictions we again abstain from a detailed definition.

By applying the described coverage degeneration technique the proposed test case selection approach gains a more comprehensive view on the coverage effects of each test case, than existing work. It is assumed that this will increase the likelihood of identifying faults that are missed by existing test case selection approaches.

Final Process Node Coverage: The presented metrics, this is, traditional coverage as well as neighborhood coverage for control and data flow, together with the degeneration factors are combined to a comprehensive coverage metric for process nodes (cf. Eq. (2.10)). Hence, a node j, a test suite configuration V_i, and a coverage reduction factor cov_{red} (applied when determining the neighborhood coverage) are taken to determine the coverage which is achieved by V_i on j. Note, $DNeibCov_n(V_i, j, cov_{red})$ calculates the neighborhood coverage based on the process models' data flow.

$$\begin{aligned} C_s(V_i, j, cov_{red}) = {} & cov_{tr}^{deg}(j, V_i) \cdot cov_{tr}(V_i, j) \\ & + cov_n^{deg}(j, V_i, cov_{red}) \cdot NeibCov(V_i, j, cov_{red}) \\ & + Dcov_n^{deg}(j, V_i, cov_{red}) \cdot DNeibCov(V_i, j, cov_{red}). \end{aligned} \tag{2.10}$$

The proposed concepts address the first two identified research questions by providing a more comprehensive view on coverage calculation and coverage requirements than existing process test case selection work. However, we also want to provide a solution to address flexible requirements and questions such as "Which test cases should be selected to get the maximum possible coverage within three hours test suite execution time?". We assume that Genetic Search Algorithms can play a viable role to address such challenges.

3. Genetic Selection Algorithm

A Genetic Algorithm (GA) is a search heuristic that mimics natural selection.[23] The first step is to determine the *individuals* of the problem and their encoding. For test case selection, intuitively, the test cases are *encoded* as binary genes and combined to individuals, i.e. the test suits. Multiple individuals then form the *population*. Each individual is assessed using a *fitness function* that can calculate the individual's quality. Subsequently, the individuals with the highest quality (i.e. fitness) are selected and combined (i.e. by applying *crossover* and *mutation*). The combined individuals form the next generation of the population. Repeatedly applying the last step typically increases the average quality of the whole population over time. Hereby, an adequate solution to the search problem is identified.

Genetic Encoding: Each potential test suite configuration V_i (cf. Definition 2.1) consists of multiple test cases. This is $V_i := \langle v_1, \ldots, v_k \rangle$ which is encoded in a binary way based on the value of the attribute *enabled* in each v:

$$V_i^{enc} := \langle v_1 \cdot enabled, \ldots, v_k \cdot enabled \rangle. \tag{3.11}$$

Generating the First Population: The presented genetic approach supports two ways of generating the first population. The first population is the initial set of all currently evolving test suite configurations, $P := \langle V_1^{enc}, \ldots, V_S^{enc} \rangle$. Note, S holds the user chosen maximum *population_size*.

First, a fully random approach. More precisely, *population_size* test suite configurations V_i^{enc} are generated and filled with randomly generated genes (i.e. test case *enabled* states). A random number $rand \in [0, 1]$ is generated for each test case in V_i. If $rand$ is lower than 0.5 then the test case (i.e. the gene) is disabled (0), else enabled (1). Hereby, the first population starts with a relatively even distribution of enabled/disabled test cases. We see this as a suitable starting point for most search problems (e.g. for the search/test case selection problem that utilizes Eq. (3.13)).

However, secondly, it can sometimes be expected that the final search result (i.e. a test suite configuration which fulfills all user requirements) will have enabled a small or high amount of test cases to become an optimal solution. In such a case, the first population generation approach can be fine-tuned accordingly. For example, assume that a user decides that a large process model should be tested in relatively little time. Then, likely, relatively few test cases will be enabled at the final test suite configuration.

Hence, instead of comparing $rand \in [0, 1]$ with 0.5, it is compared with a configurable value $population\_config \in (0, 1)$. This enables to control how many test cases become, most likely, enabled or disabled. The main advantage of this approach is that the initial starting population is already relatively similar to the final solution. For example, only a few test cases are enabled at the initial population for a search problem were also, likely, only a few test cases would be enabled at the optimal solution.

Therefore, the final search result can be generated quicker. Hence, the genetic search algorithm must not "waste" computation time to disable test cases which could be already disabled in the initial starting population. Additionally, the quality of the identified solution increases. This is because the genetic search algorithm can invest computation time (i.e. generations) into optimizing valid potential solutions instead of searching a solutions that, for example, is not "rejected" by a penalty function (cf. Eq. (3.14)).

Fitness Function: A fitness function allows to assess the quality (i.e. fitness level) of each individual (i.e. of each test suite configuration). For example, here the quality is measured by taking the test suite coverage, which is achieved by a specific test suite configuration, in relation to the required test suite configuration execution time. We assume a test suite configuration with a higher fitness level as better than one with a lower fitness value.

The following fitness function (cf. Eq. (3.13)) utilizes Eq. (3.12), to assess the achieved test coverage of a test suite configuration V_i^{enc}.[h] Therefore, Eq. (3.12) adds up and determines (by using Eq. (2.10)) the coverage of each process node j. We assume, that a node does not gain any advantage from achieving a coverage level which is above its own calculated optimal coverage level (cf. Eqs. (3.12)–(3.14)). Hence, we take the minimum between the achieved final coverage C_s (cf. Eq. (2.10)) of the node j and its optimal coverage C_o (cf. Eq. (2.1)). The added up coverage is then divided by the maximum possible optimal coverage (i.e. the sum of all nodes' optimal coverage) to normalize the generated result.

$$cov_r(V_i, cov_{red}) = \frac{\sum_{j \in N} \min(C_o(j), C_s(V_i, j, cov_{red}))}{\sum_{j \in N} C_o(j)}. \tag{3.12}$$

The first fitness function, cf. Eq. (3.13), utilizes a user-chosen minimum test coverage value $cov_{obj} \in [0, 1]$. It assesses a test suite configuration V_i to check if V_i achieves at least cov_{obj} percent of the total possible optimal coverage within minimal test suite execution time.

$$fit_{minT}(V_i, cov_{obj}, cov_{red}) = \begin{cases} cov_r(V_i, cov_{red})/100 & \text{if } cov_r(V_i, cov_{red}) < cov_{obj}, \\ \dfrac{\sum_{j \in N} \min(C_o(j), C_s(V_i, j, cov_{red}))}{x} & \text{otherwise.} \end{cases} \tag{3.13}$$

Specifically, fit_{minT} starts by determining if the minimum coverage objective cov_{obj} is already fulfilled. Therefore, it compares the average node coverage of V_i (using $cov_r(V_i, cov_{red})$, Eq. (3.12)) with cov_{obj}. If the cov_{obj} is not fulfilled then the achieved coverage is divided by 100 and returned. Hence, the fitness increases when additional test cases are enabled. So the GA is motivated to enable at least enough test cases to achieve a minimum coverage of cov_{obj}.

[h]Note, that V_i^{enc} can always be decoded into a specific V_i by using the known V and setting the respective *enabled* states.

If the cov_{obj} is fulfilled then the achieved coverage is divided by x. x is defined as the sum of the total execution times over all enabled test case in V_i. Calculating the execution time of a single test case v starts by determining the average execution time (e.g. based on timestamps stored in recorded process execution logs[i]) of each node which is part of the execution path N_v. Subsequently, the average execution times of each node in N_v are summed up to calculate v's expected total execution time. Hence, the fitness increases by preferring test cases that are executed quickly while providing a high amount of additional coverage.

The second fitness function $fit_{max}(V_i, cov_{red})$ (cf. Eq. (3.14)) assesses a test suite configuration V_i. It checks if V_i achieves the maximum possible total process model coverage in at most g total test suite execution time. Therefore, it starts by calculating the total test coverage achieved by V_i. Subsequently, it multiplies the coverage with a dynamic penalty factor if the total execution time x of V_i is too high compared to the user chosen maximum execution time objective g (cf. Eq. (3.15)).

$$fit_{max}(V_i, cov_{red}) = \left[\sum_{j \in N} \min(C_o(j), C_s(V_i, j, cov_{red}))\right] \cdot (1 - d(g, x)), \quad (3.14)$$

$$d(g, x) = \begin{cases} 0 & \text{if } x \leq g, \\ 1 & \text{if } x \geq g \cdot 2, \\ \dfrac{(x - g)}{g} & \text{otherwise.} \end{cases} \quad (3.15)$$

Equation (3.15) checks if the total test suite execution time of V_i (i.e. x) is below the user chosen execution time objective g. If x is below g (i.e. the total execution time is below the chosen maximum one) then no penalty is applied. The maximum penalty of 1 is applied if x is twice as high than g. Finally, if x is between g and two times g, then a fraction of the maximum penalty is applied to increase the flexibility of the presented approach. Hence, the algorithm is able to select a test suite configuration which is slightly above the chosen maximum execution time. Therefore the configuration has to, compared to alternative configurations, provide a dramatic coverage improvement for only a slight miss of the execution time objective.

Selection of Parents: Parents must be selected to create offspring that can form the next generation.[23] Therefore, the user chooses an *offspring_rate* that controls how many percent of the old generation will be selected as parents. Subsequently, the parents are replaced with their children to generate the next generation. The selection process itself is based on the Tournament Selection technique.[23] Hence, the algorithm randomly chooses individuals and compares their fitness. The individual with the highest fitness is selected until *offspring_rate* percent of the *population_size*

[i]Note, if no execution logs are available then the expected average node execution times can also be specified manually, for example, by a domain expert.

is chosen. Tournament Selection was chosen because the selection pressure can be controlled by varying the amount of compared individuals. Moreover, it also showed encouraging results when we compared it with other selection techniques during the preliminary tests.

Crossover and Mutation: The proposed GA utilizes Multi Point Crossover.[23] Hence, two parent individuals are selected and a crossover operation is applied to generate two new individuals (children). Therefore, $crossover\_points \in [0, I]$ (i.e. the user chosen amount of crossover points, where I holds the amount of test cases stored in a single individual) points are randomly chosen and ordered. Then the algorithm iterates through all points and the section between the last point and the current one is swapped between the parents.[23] After crossover, each generated child is mutated. Hence, the mutation algorithm iterates through all genes of the child and generates a random value $rand \in [0, 1]$ during each iteration. If $rand < mutation\_rate$, then the current gene is replaced by a randomly generated one.[23] Multi Point Crossover was chosen because it provides the necessary flexibility to adapt it for each problem size using the *crossover_points* variable. Finally, the generated children replace their parents to create the next generation of the population.

Termination: The GA terminates automatically when the termination condition, to repeat the algorithm for $max\_generation \in \mathbb{N}$ number of times, is satisfied. It returns the best individual, i.e. the test suite configuration with the highest fitness value, found until then.

GAs provide flexibility, for example, a custom fitness function can be integrated to address unique coverage selection requirements. Moreover, they can be customized. For example, by exchanging algorithm components, such as, the applied crossover method, or by adapting parameters for various problem sizes. Hence, we propose GAs as an expandable foundation for process model test case selection. In the following, their effectiveness is evaluated in comparison with other test case selection techniques.

4. Performance Optimization

We found that the proposed genetic process model test case selection approach can create high quality test suite configurations in a relatively low amount of time. Even through only a non-optimized proof-of-concept implementation was used. For example, below 5 min computation time were required, on average, to determine a high quality test suite configuration for a large and highly complex process model with 266 process model nodes and 390 test cases. Note that only a standard desktop PC was used during the evaluation. It consists of a 3 gigahertz Intel CPU and 8 gigabyte of RAM running on Debian 8. Moreover, high performance frameworks for GAs (cf. Ref. 24) enable to execute GAs in parallel on multiple machines. This would, likely, speed up the proposed genetic test case selection approach even more.

Nevertheless, it still takes a non-negligible amount of computation time to find a high quality test suite configuration for large and complex process models. In addition, it also must be considered that the current process repositories frequently, each, contain hundreds of models.[25] These business process models are frequently applied in versatile service, partner, and political landscapes that result in the need to change the models in a rapid pace.[16,17] We assume that these changes are typically small compared, for example, to a process model redesign that changes large parts of the model.[26] It is expected that only a single feature or process model node is adapted/added/exchanged/removed.

Hence, the changed process model version shares many of its tests and test paths with its original unchanged version. Therefore, the old test suite configurations (i.e. generated for the original process model) are still partly relevant for the new, changed, process model version. So, this paper proposes a novel approach to speed up the presented genetic test case selection technique for the changed model. Therefore, existing *previously generated test suite configurations* are exploited (i.e. test suite configurations generated for the original, unchanged, process model) as a foundation for the construction of new test suite configurations for the changed process model (cf. Fig. 3). Note, that, after each process model change, a new test suite configuration must be generated. This ensures that the selected/enabled test cases are still optimal and respect all the change operations which were applied on the respective process model.

So, before starting to construct a new test suite configuration, for the new changed process model version, it is checked if an old test suite configuration,

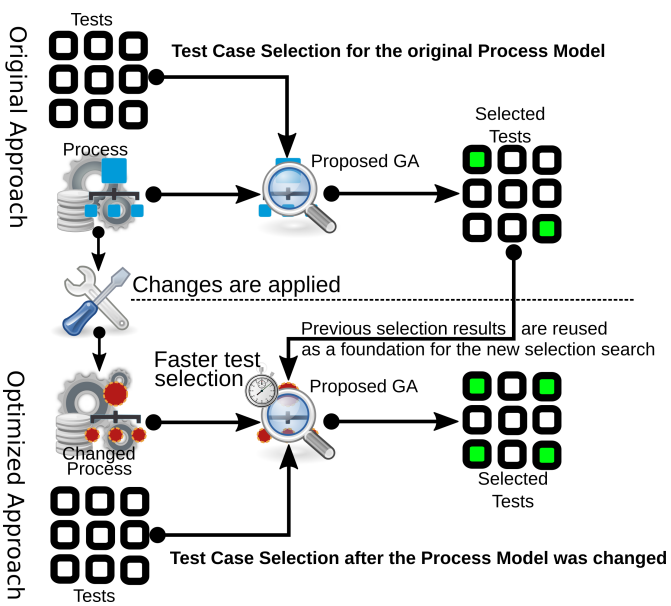


Fig. 3. Increasing the test case selection performance for changed process models.

generated for the original unchanged model, exists. Moreover, the selection query utilized to generate the old configuration must be equal or very similar to the one that is applied on the changed process model. If this is the case, then the old test suite configuration is exploited as a foundation to construct the initial population of the proposed genetic test case selection approach. Hereby, the search for a test suite configuration for the changed process model is accelerated.

Therefore, two steps are executed: First, the old test suite configuration is analyzed to detect if any of the test cases was changed. For example, if a new test case was created or an old one was deleted/replaced because of the applied process model change operations. Subsequently, the old test suite configuration is adapted accordingly by adding new test cases and removing deleted test cases. For example, new test cases are required to cover newly added process nodes. Note, new/changed test cases are always assumed as disabled, i.e. not selected.

Secondly, the adapted old test suite configuration it expanded to generate an initial starting population P. Therefore, the old test suite configuration is converted into a V^{enc}. Hence, the enabled/disabled state of each test case configured in the analyzed/adapted old test suite configuration is converted in a gene. Subsequently, a number $exp \in [0, 1]$ is chosen by the user that controls how many of the genes in V^{enc} are randomly selected and become assigned a new random state. Hence, it is randomly chosen if the gene, and therefore the test case, should be enabled or disabled. This enables to generate slightly different versions of the old test suite configuration. Hence, by applying this step multiple times, once for each yet to be generated entry in P, the initial population P is filled with multiple potential test suite configurations. Those potential test suite configurations are based on a previously determined high quality test suite configuration. In addition, they are also different enough to provide freedom for further optimizations. For example, to identify a test suite configuration which fits all the applied process model changes and conforms to the respective selection criteria (cf. Fig. 3).

Moreover, a new termination approach is proposed. Hence, if an old test suite configuration is reused then the GA will no longer terminate after a fixed amount of generations. Instead, a more *flexible combination* of the *fixed generation termination* approach and a *quality improvement threshold* approach is utilized. This is to terminate when the overall fitness of the best identified test suite configuration stays the same for a specific amount of generations, cf. Ref. 23. Hence, a *min_generation* $\in \mathbb{N}$ value is chosen by the user that indicates that the genetic selection process should be executed for a minimal (i.e. *min_generation*) amount of generations. After the minimal generation amount requirement is fulfilled, then the second termination condition, representing a quality improvement threshold, is used. The quality improvement threshold is controlled by a variable called *min_improvement* $\in (0, 1)$. It indicates that the minimal fitness (i.e. quality) of the current best test suite configuration should be *min_improvement* percent above the best test suite configuration which was found up to *improvement_generation* $\in \mathbb{N}$ generations before. If this requirement cannot be fulfilled then the test case selection ends. This is

because the search has likely identified a high quality test suite configuration which can hardly be further improved.

The combination of these two termination factors (a) motivates the genetic test case selection search to leave a local optimum that could be contained in the initial population by forcing the algorithm to explore the search space (cf. the *min_generation* value). Moreover, (b), the quality improvement threshold termination approach motivates the algorithm to terminate as soon as it makes "sense". This is because further improvements are unlikely achievable and, therefore, it likely reduces the overall computation time invested in each test case selection search.

The proposed novel initial population generation approach enables to reuse the computational effort which was invested in already executed test case selection searches. Hereby, test suite configurations can be generated substantially faster compared to a non-optimized approach. This is because a non-optimized approach always has to start its test case selection search from scratch. The impact of the presented optimization is evaluated in the following. Therefore, this work compares the amount of generations that are required to identify a suitable test suite configuration for the genetic selection approach with and without the discussed optimization.

5. Evaluation

This work assesses the feasibility of the proposed process model test case selection approach. In addition, it evaluates the impact of the presented optimization. Therefore, an evaluation is conducted which uses three different process models with increasing size and complexity. Moreover, the genetic selection approach was compared with alternative selection techniques. Those alternatives consist of *random*, *adaptive greedy*, *simulated annealing*, *sequential backward selection*, and *hill climbing*-based test case selection techniques.

Designing Test Problems: The test data which was used for the evaluation consists out of three artifacts, namely, (a) three process models (with low, medium, and high complexity); (b) test cases (one test case was generated for each possible execution path for each model); and (c) historic data (e.g. recorded execution logs, to determine the execution frequency of a node or its average execution time). All test data were artificially generated. Moreover, each evaluated test case selection technique was executed on each of the three models and their related data (i.e. test cases and historic data). Each test case was sextupled. This simulates that the internal behavior of process nodes is typically very complex and, therefore, multiple test cases with various test data are required to thoroughly test it.

The process model generation starts with an initial model with a low complexity (20 nodes, 42 test cases, 7 unique execution paths). Subsequently, the initial model was then extended by adding additional paths and XOR splits to generate a model with medium complexity (80 nodes, 120 test cases, 20 paths) and high complexity (266 nodes, 390 test cases, 65 paths). Finally, artificial historic data (i.e. execution

log data) were generated in a deterministic way. For example, the node execution time was determined from the node position and a default execution timespan. Hence, the test data is "stable" and can be reproduced for future evaluations. The same test data (i.e. process models, execution logs, and so on) are utilized for the evaluation of the presented genetic process model test case selection approach and the associated performance optimization.

Metrics and Evaluation: The evaluated test case selection techniques were compared as follows. First, the proposed genetic search algorithm-based approach tried to answer one of the two questions: (a) "Which test cases should be executed to achieve a X percent process node coverage within a minimal test suite execution time?"; or (b) "Which test cases should be executed to achieve the maximum possible coverage in Y minutes test suite execution time?".

Subsequently, the timespan which is required to execute the identified test suite configuration, for questions (a) or (b), was calculated. Finally, the determined timespan was used by the other evaluated selection techniques. These are, for example, random selection, adaptive greedy selection, and simulated annealing based selection. The random and adaptive greedy test case selection approaches select one test case after another. They stop when selecting another test case would create a test suite configuration which requires more time to execute than the one identified by the proposed GA-based technique. The simulated annealing-based approach, in comparison, generates and randomly alters whole test suite configurations. It strives to identify a test suite configuration which takes equal or less time to execute than the one identified by the proposed genetic selection approach. In addition, the solution should provide a maximum possible process node coverage.

The *random selection* (cf. Ref. 27) technique randomly selects each test case from a list of not yet selected test cases. *Adaptive greedy selection*, however, analyzes and orders each available not yet selected test case based on its additional coverage/required execution time balance. Finally, the test case which provides the most additional coverage for the least additional execution time is chosen. The *genetic selecting* technique utilizes the approach described in Sec. 3.

The *simulated annealing*-based approach (cf. Ref. 28) always starts with the a randomly generated active test suite configuration. Hence, each test case's enabled/disabled state was defined randomly. Subsequently, a configurable random amount of test cases (controlled by the variable alter $\in (0, 1)$), of the currently active test suite configuration, is randomly selected and its enabled/disabled state is flipped. Hence, a disabled test case becomes enabled and vice versa. Finally, the generated new test suite configuration is analyzed to determine if it should be accepted as the new active solution. Therefore, an *acceptance_probability* $\in (0, 1)$ factor is calculated, cf. Eq. (5.16). This enables the simulated annealing-based approach to deal with local optima. So, it can temporarily accept and improve on a solution which is worse than the previously identified ones.

Hence, if the new solution is better than the previously identified best one then it is always accepted (i.e. *acceptance_probability* is set to 1). Here, better means that the new solution has to reach a higher coverage level than previously identified solutions. In addition, the total test case execution time must be below or equal to the permitted maximum execution time. Otherwise, the acceptance probability is calculated, by Eq. (5.16). Equation (5.16) utilizes the coverage reached by the old best solution $cov_{best} \in \mathbb{N}$, the new solution $cov_{new} \in \mathbb{N}$, and a temperature factor $tmp \in \mathbb{N}$. Finally, the calculated *acceptance_probability* factor is compared with a random value $randAccept \in (0, 1)$. The new solution is accepted if *randAccept* is smaller than the calculated *acceptance_probability* factor. Moreover, the temperature *tmp* (which influences the probability that a solution which is worse then the previously accepted one is accepted) is reduced. Therefore, *tmp* is multiplied with a $cool\_down \in (0, 1)$ factor to simulate a stepwise annealing, which also gave this algorithm its name. The total test suite configuration execution time controls if a new solution is accepted or not. Hence, it must be below or equal to the execution time which was predetermined by the genetic selection approach. The algorithm will repeatedly execute above steps until the temperature is less or equal than one.

$$ap(cov_{new}, cov_{best}, tmp) = \begin{cases} 1 & \text{if } cov_{new} > cov_{best}, \\ e^{(cov_{new} - cov_{best})/tmp} & \text{otherwise.} \end{cases} \tag{5.16}$$

The *sequential backward selection* (cf. Ref. 29) technique starts with a test suite configuration where each available test case is selected (i.e. enabled). Subsequently, it iteratively disables one test case after another until a stopping criterion is fulfilled, for example, when the test suite configuration meets a user defined maximum test suite execution time requirement. Hence, the selection approach is required to repeatedly identify the next test case that should be disabled. Therefore, it individually disables each still enabled test case and subsequently analyzes the coverage/execution time relation of the hereby generated test suite configuration. This enables to identify the test cases that adds the least value (i.e. least additional coverage for its required execution time) to the test suite configuration. This test is then disabled and the discussed iterative approach is repeated.

Hill climbing (cf. Ref. 30) is a local search technique that starts with a random valid test suite configuration (i.e. a valid but likely non-optimal solution). Subsequently, it iteratively generates slightly changed candidate solutions (i.e. slightly changed test suite configurations) and compares these to the initial starting solution. If a candidate solution reaches a higher fitness then it is accepted and utilized to spawn additional candidate solutions. Note, to reach a higher fitness, the coverage must be higher than the previous solutions while the total test suite execution time is still below a user chosen maximum execution time requirement. The iterative search stops if none of the generated candidate solutions improves the solution that was utilized to spawn them. Alternatively, it also stops after a maximum amount of iterations was computed.

The test suite configurations (i.e. the configuration identified by each of the four selection algorithms) were evaluated by determining the achieved final average node coverage, cf. Eq. (3.12). Hence, the optimal coverage, C_o (cf. Eq. (2.1)), of each node $j \in N$ of a process model $O = (N, CE, DE)$ is added up and then compared with the added up achieved final node coverage, C_s (cf. Eq. (2.10)). C_s is calculated based on the analyzed test suite configuration. In addition, fault coverage was determined by assigning artificial faults to each process node. We assume that a test suite configuration V_i would find more faults for the process node j if the achieved final coverage gets closer to the optimal coverage of j (cf. Eq. (5.17)).

$$fault(V_i, j, cov_{red}) = \begin{cases} 1 & \text{if } C_s(V_i, j, cov_{red}) > 0 \wedge C_s(V_i, j, cov_{red}) \leq C_o(j) \cdot 0.25, \\ 3 & \text{if } C_s(V_i, j, cov_{red}) > C_o(j) \cdot 0.25 \wedge C_s(V_i, j, cov_{red}) \leq C_o(j) \cdot 0.50, \\ 5 & \text{if } C_s(V_i, j, cov_{red}) > C_o(j) \cdot 0.50 \wedge C_s(V_i, j, cov_{red}) \leq C_o(j) \cdot 0.75, \\ 7 & \text{if } C_s(V_i, j, cov_{red}) > C_o(j) \cdot 0.75, \\ 0 & \text{otherwise,} \end{cases} \tag{5.17}$$

$$faultCoverage(V_i, cov_{red}) = \frac{\sum_{j \in N} fault(V_i, j, cov_{red})}{|N| \cdot 7}. \tag{5.18}$$

Note that cov_{red} represents the user chosen *coverage reduction factor* which is utilized at $C_s(V_i, j, cov_{red})$ (i.e. the coverage calculation, cf. Eq. (2.10)) during the incorporation of the neighborhood coverage of j. For example, imagine that the achieved coverage (for the process node j) would be between 25% and 50% of j's optimal coverage. Then it was assumed that three out of seven faults would be found by the test suite for the node j. Finally, the detected faults for each node were added up and divided through the maximum possible detectable amount of faults to normalize the result of Eq. (5.18).

The performance optimization evaluation utilizes the same test case selection queries/criteria as the four compared selection approaches. However, it does not compare the reached coverage and fault detection rate. Instead, the performance of the genetic selection approach *with* and *without* the proposed optimization is compared. So, the amount of generations needed by both approaches to identify an equally good result is determined and compared.

Therefore, a four-step approach is applied. (1) The proposed "classic" genetic test case selection algorithm is applied on a selection question/test problem combination. This enables to find an appropriate test suite configuration for the respective selection question. (2) Subsequently, the test problem (i.e. a process model and associated execution logs) are modified. Hereby the variable $proc\_alt \in (0, 1)$ controls how many percent of the process's nodes are modified. This enables to mimic typical process model changes such as replacing, editing, or deleting specific process model nodes.

(3) The classic (i.e. non-optimized) approach is again applied on the modified test problem (i.e. the process model and execution logs). Hereby a suitable test suite configuration for the changed process model is identified. (4) Finally, the optimized genetic selection approach is applied on the changed process model. Hence, the initially generated result (from (1)) is exploited. Note that it was generated for the old unchanged process model version. Specifically, it is utilized as a foundation to construct the initial population of the optimized genetic selection approach. The optimized approach stops as soon as it has found a test suite configuration which has a fitness value (e.g. process model test coverage) that is higher or equal than the fitness value of the test suite configuration which was identified by the classic genetic selection approach (i.e. without the proposed optimization), see step (3). Finally, after a test suite configuration with a higher or equal fitness was found, the *generations* need by the genetic approach from step (3) and step (4) are compared. This enables to assess if the proposed optimization speeds up the test case selection. This would be the case if the optimized approach requires less generations than the non-optimized one.

The generations were chosen as a comparison metric for multiple reasons. For example, they are independent from the computation hardware performance, implementation details, and software stack. Moreover, they can easily be measures/ compared. Other factors, such as, execution time, CPU time, or CPU cycles could easily be influenced by the listed factors. Imagine if one genetic selection algorithm implementation uses a distributed multi-machine genetic framework vs. a single threaded proof of concept implementation.

Results: The evaluation results were generated by applying all evaluated test case selection techniques on the described three test problems. Hence, the evaluated techniques include the proposed genetic selection algorithm, random test case selection, adaptive greedy selection, simulated annealing based selection, a sequential backward selection-based approach, and a hill climbing-based technique. For each test problem, two questions were analyzed (a) "Which test cases should be executed to achieve a X percent coverage within a minimal test suite execution time?" whereby X is 20, 40, 60, or 80% of the maximal possible optimal total process coverage. In addition, a second question was evaluated. This is (b) "Which test cases should be executed to achieve the maximum possible coverage in Y minutes test suite execution time?" whereby Y is 20%, 40%, 60%, or 80% of twice the time which would be necessary to execute each process node once.

Primary tests were executed to identify appropriate configuration values for the designed genetic test case selection technique. *Mutation_rate* (0.5%), w_{deg} (i.e. coverage degeneration, 10%), w_{degMax} (i.e. maximal coverage degeneration, 50%), cf. Eq. (2.6) and (2.7), cov_{red} (i.e. neighborhood coverage reduction per step, 30%), cf. Algorithms 1 and 2, all weights w (e.g. for coverage metrics, 1) and *offspring_rate* (50%) were fixed for all three test problem complexity levels. The value of *max_generation* (low complexity test problem:300, medium:500, high:800),

population_size (200, 400, 800) and *crossover_points* (4, 8, 15), chosen individuals for tournament selection (3, 5, 10), however, were chosen individually to reflect the increasing test problem complexity.

For example, it was found that a low number of *crossover_points* would, naturally, exchange very large chunks of test suite configurations during child generation. This hardens the fine tuning of the identified results during the final stage of the search. This challenge increases from small to large test problem complexity/sizes, so the number of *crossover_points* was increased whenever the test problem complexity increases. The fully random initial population generation approach was always used for question (a) while for question (b) *population_config* was set to 0.8. This represents that for test case selection question (b) a majority of the test cases must, likely, be disabled. This is because the permitted test suite execution time is strictly limited.

Finding appropriate parameters for GA based approaches can be hard. However, there are parameterless GAs available (see Ref. 31) that configure their parameters fully automatically. In addition the presented parameters can most likely be reused. This is because they were defined for three generic complexity levels that, as we assume, should fit for most existing process model and test suite sizes.

The configuration values of the evaluated non-genetic selection approaches also reflect the test problem complexity. Hence, *alter* (low complexity test problem:0.05, medium:0.01, high:0.009), *tmp* (10,000, 100,000, 1,000,000), and *cool_down* (0.003, 0.001, 0.0005) were chosen individually for each test problem complexity level. It is expected that this ensures a proper comparison with the proposed genetic approach. Hence, if the test problem size increases then the algorithm is provided more freedom to analyze the search space. This is achieved, for example, by a higher starting temperature and a smaller cool down factor. The hill climbing based approach utilizes *cand* percent (low complexity test problem:0.6, medium:0.8, high:1.0) of the test suite size (i.e. based on the amount of test cases that can be enabled/disabled) as the number of candidates generated during each iteration. Moreover it terminates after at most *iter* (10,000, 100,000, 1,000,000) iterations.

It was found that the default simulated annealing and hill climbing approach struggle to identify a *valid* solution on certain selection problems. Hereby, valid means that the solution is not rejected by the maximum test case execution time based penalty function. This was observed on test problems were only a low amount of test cases must be enabled at a potential solution to become valid, i.e. for question (b). For such selection problems the initial random starting solution was tuned accordingly. For example, only a low amount of randomly chosen test cases were enabled.

The results show that the GA outperforms the random, adaptive greedy, and simulated annealing selection techniques (cf. Figs. 4–7). It is also noticeable that the GA benefits from increasing the test problem complexity. The GA achieved a 3.6%/3.5% higher coverage/fault detection rate, for questions (a), compared to the adaptive greedy selection technique, for the low complexity test problem. For

the test problem with high complexity the genetic selection technique provides a 9%/10.3% higher coverage/fault detection than the adaptive greedy selection (cf. Tables 1 and 3). The other test case selection approaches (random, hill climbing, sequential backward selection, and simulated annealing based selection) are even more significantly outperformed by the GA. Note that Tables 1 and 3 display evaluation results which were already presented in Ref. 14. Tables 2 and 4 show novel results which compare the proposed approach with additional test case selection techniques.

Additionally, we found, that the simulated annealing based test case selection approach frequently was not able to identify a valid solution for certain questions. This was even the case after applying the described tuning of the initial starting solution. This was observable, for example, for question (b). For this question the

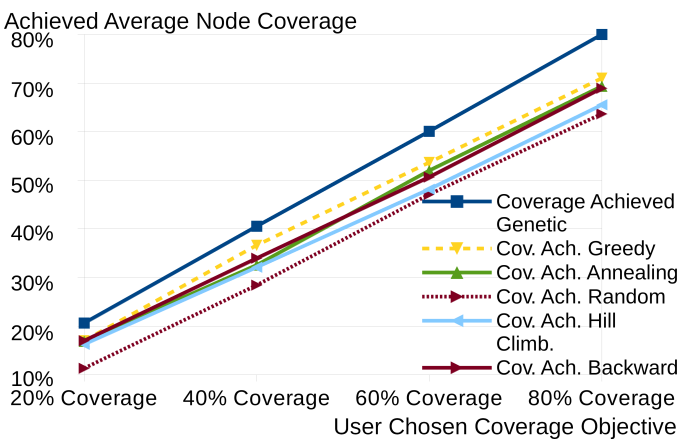


Fig. 4. Average node coverage across all three process model complexities for question (a) (User chosen coverage objective).

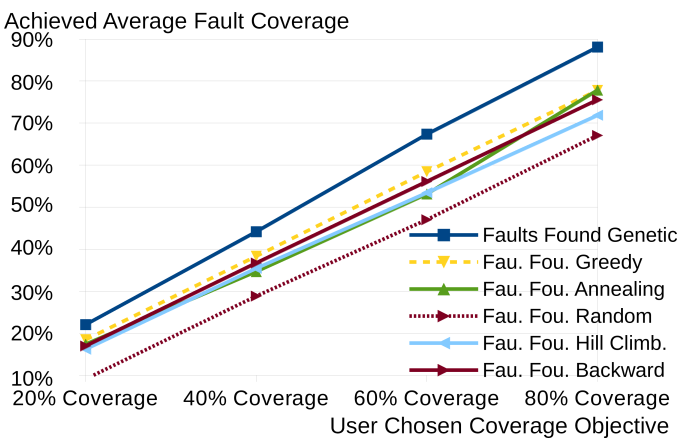


Fig. 5. Average *fault* coverage across all three process model complexities for question (a) (User chosen coverage objective).

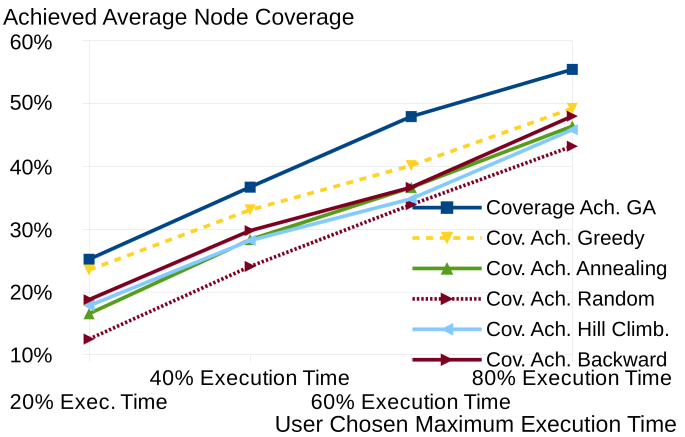


Fig. 6. Average node coverage across all three process model complexities for question (b) (User chosen maximum test suite execution time).

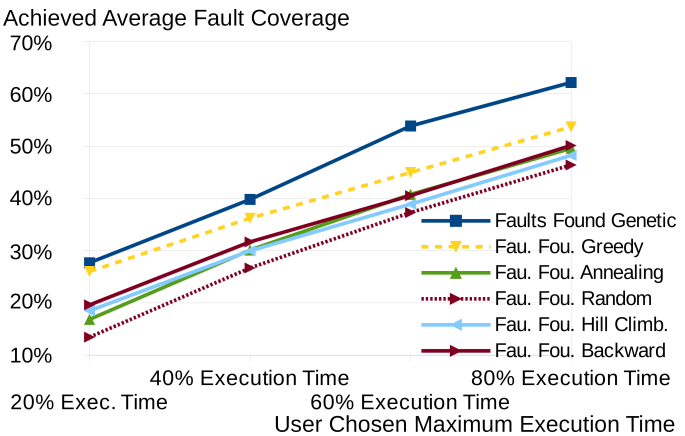


Fig. 7. Average *fault* coverage across all three process model complexities for question (b) (User chosen maximum test suite execution time).

selection algorithm has to ensure that the total test suite execution time is *not* above the permitted one. It was found that it struggles to find solutions for test problems that utilize a relatively small permitted total test suite execution time, e.g. if only 20% of twice the total execution time of each process node was permitted. Hence, the listed average evaluation results for the simulated annealing based approach only contains results where at least a single valid non-rejected test suite configuration was identified. The results are depicted in Figs. 4–7.

Overall it became obvious that the GA is able to make better use of the additional flexibility provided by more complex test problems. Hereby, flexibility means that the selection algorithm can be chosen from more test cases or more test suite execution time is permitted. Hence, it is assumed that the GA is better suited for complex process models and selection requirements than the compared techniques.

Table 1. Raw evaluation results for question (a) (User chosen coverage objective).

Process complexity, node coverage objective	Proposed genetic selection		Adaptive greedy selection		Random selection	
	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)
Low, 20% coverage	**21.2**	**23.5**	19	21.4	8.1	3.2
Medium, 20% coverage	**20.5**	**21.5**	15	16.4	10.2	10.8
High, 20% coverage	**20.2**	**21**	17	17.5	15.7	13.2
Low, 40% coverage	**41.6**	**46.4**	38.7	40	25.3	26.7
Medium, 40% coverage	**40**	**42.7**	35.1	37.1	31	30.2
High, 40% coverage	**40**	**43.3**	36.1	37.8	29	29.5
Low, 60% coverage	**60.2**	**70.7**	53.4	62.2	50.3	48.7
Medium, 60% coverage	**60**	**65.9**	53	56.6	45.7	46.6
High, 60% coverage	**60**	**65.4**	54.6	56.4	45.3	45.7
Low, 80% Coverage	**80**	**94.2**	69	81.4	65	69.6
Medium, 80% coverage	**80**	**86.9**	73	78.7	66	68.5
High, 80% coverage	**80**	**83.1**	71	73.2	60	63.1

Table 2. Raw evaluation results for question (a) (User chosen coverage objective).

Process complexity, node coverage objective	Annealing selection		Backward selection		Hill climbing selection	
	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)
Low, 20% coverage	19.8	21.3	19	20.2	18.1	19.1
Medium, 20% coverage	14.7	15.5	15	15.1	14.2	14.4
High, 20% coverage	16.1	14.9	17	15.5	16.5	15
Low, 40% coverage	33.4	36.9	35.8	39.1	34.1	38.2
Medium, 40% coverage	32.9	34	33.2	36	31.5	35
High, 40% coverage	31.4	33	32.9	35	30.7	33
Low, 60% coverage	53.3	56.2	52	60.2	49	57
Medium, 60% coverage	52	54.8	51	55.6	48.5	53.2
High, 60% coverage	50.7	48.3	49	52.5	47.6	50
Low, 80% coverage	75.2	92.8	68.1	80	65.6	76
Medium, 80% coverage	68.6	70.9	7.9	75.6	66.1	72
High, 80% coverage	64.2	69.8	68.8	71	64.8	67.5

Table 3. Raw evaluation results for question (b) (User chosen maximum test suite execution time).

Process complexity, max execution time objective	Proposed genetic selection		Adaptive greedy selection		Random selection	
	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)
Low, 20% execution time	**32.1**	**37.1**	**32.1**	**37.1**	11	13
Medium, 20% execution time	**24.6**	**24.6**	20.3	20.3	14.7	14.2
High, 20% execution time	**18.9**	**21.3**	18.1	20.62	11.8	13.1
Low, 40% execution time	**43**	**47.1**	41.6	45.2	29.3	30
Medium, 40% execution time	**36.6**	**37.6**	31.5	33.3	21.2	27.8
High, 40% execution time	**30.5**	**34.7**	26.1	30.1	21.7	22.3
Low, 60% execution time	**53.6**	**64.3**	48.1	55.7	38.5	44.1
Medium, 60% execution time	**48.1**	**50.5**	39.2	41.2	35	35.8
High, 60% execution time	**42.1**	**46.8**	33.1	38	28.2	32
Low, 80% execution time	**61.1**	**70.7**	55	60.5	47.9	56.5
Medium, 80% execution time	**56.4**	**60.7**	50.1	52.6	43.1	45.6
High, 80% execution time	**48.8**	**55.2**	42.6	48	38.6	37.1

Table 4. Raw evaluation results for question (b) (User chosen maximum test suite execution time).

Process complexity, max execution time objective	Annealing selection		Backward selection		Hill climbing selection	
	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)	Achieved coverage (%)	Detected faults (%)
Low, 20% execution time	21.2	18.6	23.2	22.4	22	21.2
Medium, 20% execution time	16.2	17.3	18.5	19.6	17.5	18.2
High, 20% execution time	12.2	14.6	14.6	16.8	13.9	16
Low, 40% execution time	32.8	36.2	34.6	39.6	32.9	37.3
Medium, 40% execution time	30.2	29.5	30	30	28	28.5
High, 40% execution time	22	24.6	24.7	25.4	23.7	24.1
Low, 60% execution time	41.2	48.5	42.1	50.4	40	48.7
Medium, 60% execution time	37.9	39.1	38.1	40.1	36.1	38
High, 60% execution time	30.8	34.7	29.8	31.1	28.4	30
Low, 80% execution time	55.6	61.8	57.3	60.1	54.3	57
Medium, 80% execution time	44.2	47.7	46.5	48.2	44.2	45.9
High, 80% execution time	39.4	39.2	40.2	42.1	39	41.8

In addition it was found to provide at least equally good results for all other problems. The improvement is even higher if the results are compared with random selection. The evaluation also shows comparable results for question (b) (cf. Figs. 6 and 7).

Note, that the evaluation results were generated by averaging the outcome of 100 runs of the random, simulated annealing, and genetic test case selection approach on each test problem and question. This ensures that the randomized behavior of those approaches does not falsify the results. This could else be the case, for example, because of a single "randomly" generated outstanding good or bad result.

The performance optimization evaluation results were generated by applying the presented genetic business process test case selection approach on the already described three test problems and two selection questions. To assess the impact of the proposed performance optimization the genetic approach was applied with and without the presented optimization. Subsequently, the amount of generations required for both approaches (with/without the presented optimization), to identify a comparable high quality test suite configuration, were compared. This enables to assess if the proposed optimization was able to reduce the amount of generations required by the genetic approach and, therefore, improved its performance. If this is the case, then, the presented optimization enables to generate an equally good result faster. Alternatively it would, probably, enable to identify better results if the same amount of generations is permitted (and therefore computation time) than required by the non-optimized genetic selection approach.

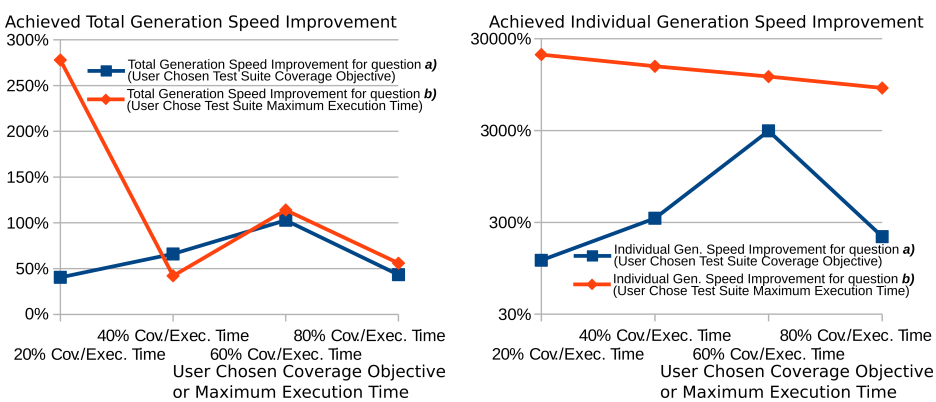
The evaluation of the performance optimization reuses the already presented configuration values for the genetic selection approach. Therefore, for the sake of brevity, only variables which are only used by the proposed optimization technique are listed in the following. Hence, *exp* was always set to 0.25 for each test problem complexity level while $proc\_alt \in (0, 1)$ was defined as 0.3 for the low complexity, 0.2 for the medium complexity, and 0.1 for the test problem with high complexity. The variable *proc_alt* defines, in percent, how many nodes of each test problems' process model are altered to simulate typical process model changes. Hence, the *proc_alt* variables' value decreases while the test problems' process model complexity increases (and therefore the amount of nodes specified in each process model). This is because this evaluation focuses on assessing the impact of the proposed optimization when small process model changes are applied. It is assumed that those changes only affect a low amount of process model nodes.

Three adaption techniques were applied to simulate process model changes: (a) a node is completely removed from the process model; (b) a new node is added to the model and placed right between a randomly chosen existing node and one of its successors or predecessors; and (c) a node is altered (i.e. it gets assigned new execution times and execution logs). Note, that the simulated process model changes were applied on randomly chosen process model nodes. Moreover, for each changed node a randomly chosen adaption technique was used.

To assess the impact of the presented optimization technique several values were collected and analyzed. First, the *total generation speed improvement* which sums up all the generations from each run. This performance indicator was collected separately for the genetic approach with and without the proposed optimization. Subsequently, it was determined if and how much less generations were required by the genetic approach with the optimization compared to the genetic approach without the optimization (in percent) to identify an equally good test suite configuration. Secondly, the *individual generation speed improvement* metric determines for each individual run the difference in the generation count between the optimized and the non-optimized approach (in percent). Note that the evaluation of the proposed performance optimization was executed 100 times. This enables to even out the random aspects of the analyzed approach. The results show the average improvement for all 100 evaluation runs.

The results show that the proposed optimization substantially increases the performance of the presented genetic selection approach (cf. Fig. 8). Depending on the analyzed improvement assessment value a performance improvement between 21.8% and 58513.5%, compared to the genetic approach without the proposed performance optimization, was found. The raw results of the performance optimization evaluation are shown in Table 5.

An enormous performance improvement was found for the second, process model execution time based, selection question (b). It originates from a large amount of evaluation runs where the optimized genetic approach was able to determine a valid solution, with higher or equal quality than the solution determined by the non-optimized approach, in below 10 generations. In comparison, the classic non-optimized genetic approach typically required between 100 and 500 generations. However, even when excluding these exceptional good results the average speed



(a) Average total generation speed improvement

(b) Average individual generation speed improvement

Fig. 8. Average generation speed improvement for all three process model complexities for question (a) and (b) when applying the presented optimization approach.

Table 5. Raw evaluation results of the performance optimization evaluation for question (a) and (b).

Process complexity, node coverage/ execution time objective	Total generation speed improvement		Individual generation speed improvement	
	Coverage objective (%)	Execution time objective (%)	Coverage objective (%)	Execution time objective (%)
Low, 20% coverage/execution time	59.4	693.2	219.6	58513.5
Medium, 20% coverage/execution time	21.8	41.6	42.6	1924.5
High, 20% coverage/execution time	40.1	98.6	85.7	124.1
Low, 40% coverage/execution time	125.7	63.6	660	44292.2
Medium, 40% coverage/execution time	32.9	28.4	271.3	733.1
High, 40% coverage/execution time	39.5	34.4	65.9	41.7
Low, 60% coverage/execution time	65.5	202.1	2955.9	11900.8
Medium, 60% coverage/execution time	48.6	93.1	178.6	18473.8
High, 60% coverage/execution time	194.2	46.8	5806.7	4490.4
Low, 80% coverage/execution time	63.1	67.2	472.9	15160.3
Medium, 80% coverage/execution time	23.8	53.5	65.6	602.9
High, 80% coverage/execution time	43.1	47.0	91.6	10391.1

improvement still reaches up to 150%. Of course, the exact improvements depend on the selection question and test data complexity.

Moreover, we found that the proposed optimization is especially beneficial when the permitted amount of generations (cf. the *max_generation* variable) is relatively small. Hereby, small must be seen in comparison to the amount of generations required to identify a Pareto optimal solution. A Pareto optimal solution is an optimal solution where applying any change would reduce its quality. Hence, we conclude that the proposed optimization is most useful when the permitted/available test case selection algorithm computation time is limited. We expect that in such a case a quick and high quality result generation is particularly important. Hence, ongoing rapid process model changes will, likely, benefit from the proposed optimization approach. It is assumed that in such a rapidly changing scenario the timespan that can be invested in testing and designed each change is, likely, heavily limited.

6. Related Work

Related work can be classified into two categories: test case selection and minimization. Minimization is only partly relevant for this paper because it concentrates less on selection, but more on test suite redundancy prevention. Hence, minimization approaches remove test cases that are only covering process parts that are already sufficiently tested by other test cases in the test suite. However, the research areas are connected and the proposed approach can be used to generate results which are comparable to existing minimization approaches, for example, by defining a 100% coverage objective which should be reached in minimal test suite execution time. Hence, this section also discusses minimization approaches.

Two strategies are currently applied to achieve minimization. One option is to analyze and minimize an already existing set of test cases (also called test suite). Farooq and Lam describe their minimization objective as a Traveling Sales Man (cf. Ref. 32) or an Equality Knapsack problem (cf. Ref. 11). Subsequently, they apply Evolutionary Computation heuristics to search for a minimal set of test cases that still provides full structural coverage. However, the authors only used their approach to minimize test cases which were generated through model-based software testing using UML activity diagrams.

Alternatively, the test case generation algorithms can try to generate a duplicate/redundancy free test suite. Reference 33 utilizes Orthogonal Array Testing to ensure a redundancy free test suite. Orthogonal Array Testing is a statistical method that calculates which parameter values should be tested in which combination. In addition, the authors apply semantic constraints to reduce the amount of generated process model test cases. Reference 34 instead searches for an optimal amount of test points where sensors can be added to a process model to detect faulty behavior. Hence, the work identifies the minimal amount of test points which are necessary to achieve a user chosen coverage level.

Selection analyzes all test cases and selects those which provide the most value. Reference 12 selects all test cases which cover process model areas that were changed since the last test runs. Ruth instead concentrates on external partners and selects only test cases which cover a process partner that was adapted.[35] Ruth's approach requires that each partner process definition is publicly available which is rather unlikely in real world scenarios.

Overall, existing work is frequently utilizing simple and relatively inflexible selection requirements such as "Which test cases should be selected to achieve a 100% coverage?". Hence, existing work is frequently not optimizing test suite execution times to their full potential. Additionally, existing work is treating each process node equally. For example, current work is assuming a node as completely tested if at least a single test case tests this node once, independently from the nodes' complexity. Hereby the unique coverage requirements of each process and node are not respected. This reduces the likelihood to identify a fault because important nodes are not tested as thoroughly as necessary. Finally, we found that existing work does not utilize a comprehensive approach (such as the presented Neighborhood Coverage) to describe test case coverage effects.

7. Conclusions and Discussion

This paper provides coverage metrics and a GA for test case selection specifically geared towards process model testing ($\mapsto$ **RQ1** and **RQ2**). The evaluation results support their feasibility even for complex process models. It is also shown that historic information such as log files can positively influence the generated results. They enable the incorporation of test case execution times, hence enabling the selection of those test cases that fulfill user-chosen requirements in minimal time.

The presented GA basically enables the creation of more flexible test case selection approaches for process models (addressing **RQ3** and **RQ4**). It can also be adapted to meet unique user requirements, such as "node X must always be tested", "only the partner processes should be tested", or "only modified process parts should be tested". Therefore, only adequate optimal coverage calculation metrics must be defined. As shown, the flexibility can be further increased based on different fitness functions. Overall, this work provides the most comprehensive and flexible process model test case selection solution so far. This is because it takes the *characteristics* of process models, e.g. based on neighborhood coverage metrics and execution log files, into account.

We see this as an advantage of the GA over more targeted approaches that, as we found, require more effort to customize them for new test case selection criteria. Moreover, the evaluation shows that more targeted algorithms, such as, the greedy algorithm, are outperformed by the presented approach regarding the test selection result quality. By its nature, the GA-based approach is more computational intense than, for example, the greedy algorithm. However, we do not see that as a major issue because we found, during the evaluation, that even overly large and complex process models could be analyzed in a reasonable amount of time. This gave us a positive impression of the overall performance, especially, because this paper only utilizes a prototypical implementation of the presented approach. Moreover, high speed GAs are available that can distribute their work over multiple machines.[36]

Moreover, this paper presents and evaluates novel performance optimization approaches for genetic test case selection techniques ($\mapsto$ **RQ5**). The presented optimizations enable to significantly reduce the computation time required to identify a high quality test suite configuration. The results indicate that the presented approaches can be successfully applied in various environments. This also includes environments where the process models are large and complex, frequently adapted, and the permitted computation time to identify a test suite configuration is, therefore, limited.

This paper, for the sake of brevity and simplicity, concentrates only on a single coverage calculation approach, namely, node coverage. This is because node coverage enables to illustrate the proposed approach and the addressed limitations of existing work in an easily comprehensible way. This comes from that fact that its fine granularity represents the coverage of each node — compared to competing coverage approaches, such as, the more cause grained branch coverage. For example, how the proposed neighborhood coverage affects the calculated coverage of each process node that "surrounds" a tested process execution path can more easily be illustrated based on node coverage than, e.g. on branch coverage. Node coverage also enabled to visualize how the coverage is reduced with increasing distance from the tested path in a easily comprehensible way.

Moreover, we are confident that the presented ideas and concepts can be generalized to apply them on competing coverage approaches, such as, branch coverage or path coverage. This also includes the presented genetic process model test case

selection approach along with the concept of respecting the unique characteristics of process models during test case selection. For example, to apply a transition from node to branch coverage the presented optimal coverage calculation approach, cf. Eq. (2.1), must be adapted. For this, it is proposed to modify it so that it no longer measures how many test cases are covering (i.e. testing) a single node, but how many test cases are covering a single branch. A similar transition is, as we assume, likely applicable on all the other presented concepts and will, in detail, be presented in future work.

Future work will also incorporate process model test case prioritization and minimization. Hereby the applicability and feasibility of flexible GAs will be analyzed for these domains. Moreover, we plan to conduct a case study to analyze the impact of the proposed coverage metrics on the test selection quality in real-world scenarios. This will enable to assess the feasibility of, for example, optimal coverage or neighborhood coverage in additional large, complex, and diverse environments.

In addition, we found that the proposed approach has a strong focus on the process model. For example, it determines how each test case covers the model or which process model nodes should be tested with the most effort. However, there are challenges which cannot be addressed with such an approach. For example, imagine a scenario where two test cases provide equal coverage and execution time but only one of them would actually identify a fault. In such a scenario, the proposed approach would be indecisive between both test cases because it uses an abstracted coverage/execution time-based test rating criteria for selection purposes. Hence, future work will focus on representing test cases in a more diverse way. Hereby we plan to improve test case selection by raking test cases based on an artificial fault detection likelihood. The planned ranking approach will incorporate, among other aspects, a mixture of test data diversity and control flow coverage.

References

1. H. A. Reijers, I. T. P. Vanderfeesten and W. M. P. van der Aalst, The effectiveness of workflow management systems: A longitudinal study, *Int. J. Inform. Manage.* **36** (2016) 126–141.
2. W. M. P. van der Aalst, Business process management: A comprehensive survey, *Softw. Eng.* **2013** (2013) Article ID 507984, 37pp.
3. W. M. P. van der Aalst, A. H. M. ter Hofstede and M. Weske, *Business Process Management*: *A Survey* (Springer, Berlin, 2003), pp. 1–12.
4. K. Böhmer and S. Rinderle-Ma, A systematic literature review on process model testing: Approaches, challenges, and research directions, preprint (2015), arXiv:1509.04076.
5. F. Leymann and D. Roller, *Production Workflow Concepts and Techniques* (Prentice Hall, 2000).
6. J. Mendling, *Metrics for Process Models*: *Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, Vol. 6 (Springer, New York, 2008).
7. Z. Zakaria, R. Atan, A. A. A. Ghani and N. F. M. Sani, Unit testing approaches for BPEL: A systematic review, in *Asia-Pacific Software Engineering* (IEEE, 2009), pp. 316–322.

8. Z. J. Li, W. Sun, Z. B. Jiang and X. Zhang, BPEL4WS unit testing: Framework and implementation, in *Web Services* (IEEE, 2005), pp. 103–110.
9. V. Stoyanova, D. Petrova-Antonova and S. Ilieva, Automation of test case generation and execution for testing web service orchestrations, in *Service-Oriented Systems Engineering* (IEEE, 2013), pp. 274–279.
10. K. Kaschner and N. Lohmann, Automatic test case generation for interacting services, in *Service-Oriented Computing* (Springer, New York, 2009), pp. 66–78.
11. U. Farooq and C. P. Lam, Evolving the quality of a model based test suite, in *Software Testing, Verification and Validation* (IEEE, 2009), pp. 141–149.
12. B. Li, D. Qiu, S. Ji and D. Wang, Automatic test case selection and generation for regression testing of composite service based on extensible bpel flow graph, in *Software Maintenance* (IEEE, 2010), pp. 1–10.
13. M. Harman and B. F. Jones, Search-based software engineering, *Inf. Softw. Technol.* **43** (14) (2001) 833–839.
14. K. Böhmer and S. Rinderle-Ma, A genetic algorithm for automatic business process test case selection, in *On the Move to Meaningful Internet Systems*: *CoopIS* (Springer, New York, 2015), pp. 166–184.
15. J. Cardoso, Process control-flow complexity metric: An empirical validation, in *Services Computing* (IEEE, 2006), pp. 167–173.
16. W. Fdhila, S. Rinderle-Ma and C. Indiono, Change propagation analysis and prediction in process choreographies, *Coop. Inf. Syst.* **24** (2015) 47–62.
17. S. Rinderle, M. Reichert and P. Dadam, Correctness criteria for dynamic changes in workflow systems — A survey, *Data Knowl.* **50** (2004) 9–34.
18. S. Kriglstein, G. Wallner and S. Rinderle-Ma, A visualization approach for difference analysis of process models and instance traffic, in *Business Process Management* (Springer, 2013), pp. 219–226.
19. S. Rinderle, M. Reichert and P. Dadam, Flexible support of team processes by adaptive workflow systems, *Distrib. Parallel Databases* **16** (2004) 91–116.
20. V. Gruhn and R. Laue, Complexity metrics for business process models, in *Business Information Systems* (Springer, 2006), pp. 1–12.
21. M. Alrifai, P. Dolog and W. Nejdl, Transactions concurrency control in web service environment, in *Web Services* (IEEE, 2006), pp. 109–118.
22. K. Böhmer and S. Rinderle-Ma, A testing approach for hidden concurrencies based on process execution logs, in: *Service-Oriented Computing* (Springer, New York, 2016).
23. A. Eiben and J. Smith, *Introduction to Evolutionary Computing* (*Natural Computing Series*) (Springer, New York, 2008).
24. D. Lim, Y. S. Ong, Y. Jin, B. Sendhoff and B. S. Lee, Efficient hierarchical parallel genetic algorithms using grid computing, *Future Gener. Comput. Syst.* **23**(4) (2007) 658–670.
25. R. M. Dijkman, M. La Rosa and H. A. Reijers, Managing large collections of business process models-current techniques and challenges, *Comput. Ind.* **63** (2012) 91–97.
26. M. Weske, *Business Process Management*: *Concepts*, *Languages*, *Architectures* (Springer, New York, 2012).
27. M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. dAmorim and R. B. Prudêncio, Randomized constraint solvers: A comparative study, *Innov. Syst. Softw. Eng.* **6**(3) (2010) 243–253.
28. K. A. Dowsland and J. M. Thompson, Simulated annealing, in: *Handbook of Natural Computing* (Springer, New York, 2012), pp. 1623–1655.
29. A. R. Webb, *Statistical Pattern Recognition* (John Wiley & Sons, New York, 2003).

30. S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik and D. D. Edwards, *Artificial Intelligence*: *A Modern Approach*, Vol. 2 (Prentice Hall, 2003).
31. G. R. Harik and F. G. Lobo, A parameter-less genetic algorithm, in *Genetic and Evolutionary Computation* (Morgan Kaufmann Publishers, 1999), pp. 258–265.
32. U. Farooq and C. P. Lam, A max-min multiobjective technique to optimize model based test suite, in *Software Engineering*, *Artificial Intelligences*, *Networking and Parallel/Distributed Computing* (IEEE, 2009), pp. 569–574.
33. A. Askaruinisa and A. Abirami, Test case reduction technique for semantic based web services, *Comput. Sci. Eng.* **2**(3) (2010) 566–576.
34. D. Borrego, M. T. Gómez-López and R. M. Gasca, Minimizing test-point allocation to improve diagnosability in business process models, *Syst. Softw.* **86**(11) (2013) 2725–2741.
35. M. E. Ruth, Concurrency in a decentralized automatic regression test selection framework for web services, in *Mardi Gras Conf.* (ACM, 2008), pp. 7:1–7:8.
36. J. Porta, J. Parapar, R. Doallo, F. F. Rivera, I. Santé and R. Crecente, High performance genetic algorithm for land use planning, *Comput. Environ. Urban Syst.* **37** (2013) 45–58.