Using Data Mining Techniques to Generate Test Cases from Graph Transformation Systems Specifications

Maryam Asgari Araghi (ma_asgar@encs.concordia.ca)
Concordia University
Ferhat Khendek
Concordia University
Vahid Rafe
Goldsmiths University of London

Research Article

Keywords: Software testing, Model-based testing, Test case generation, Model checking, Data mining algorithms, Graph transformation systems

Posted Date: August 8th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3226069/v1

Additional Declarations: No competing interests reported.

Using Data Mining Techniques to Generate Test Cases from Graph Transformation Systems Specifications

Maryam Asgari Araghi[1*], Vahid Rafe[2] and Ferhat Khendek[3]

[1,3]Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada.
[2]Department of Computing, Goldsmiths, University of London, London, UK.

*Corresponding author(s). E-mail(s): ma_asgar@encs.concordia.ca;
Contributing authors: v.rafe@gold.ac.uk; ferhat.khendek@concordia.ca;

Abstract

Software testing plays a crucial role in enhancing software quality. A significant portion of the time and cost in software development is dedicated to testing. Automation, particularly in generating test cases, can greatly reduce the cost. Model-based testing aims at generating automatically test cases from models. Several model based approaches use model checking tools to automate test case generation. However, this technique faces challenges such as state space explosion and duplication of test cases. This paper introduces a novel solution based on data mining algorithms for systems specified using graph transformation systems. To overcome the aforementioned challenges, the proposed method wisely explores only a portion of the state space based on test objectives. The proposed method is implemented using the GROOVE tool set for model-checking graph transformation systems specifications. Empirical results on widely used case studies in service-oriented architecture as well as a comparison with related state-of-the-art techniques demonstrate the efficiency and superiority of the proposed approach in terms of coverage and test suite size.

Keywords: Software testing, Model-based testing, Test case generation, Model checking, Data mining algorithms, Graph transformation systems

1 Introduction

Software testing is an important activity in the software development life cycle. It aims at checking the software product against the specified requirements, detect potential bugs and improve the quality before release (Naik and Tripathy, 2011). Software testing can consume a significant amount of time and resources, often accounting for up to half of the total development cost (Beizer, 1990). Automating the software testing activity and its central component of test case generation certainly improves efficiency (Naik and Tripathy, 2011). The use of software models for automating the testing process and reducing its costs has been a common practice for several years (Schieferdecker, 2012; Utting and Legeard, 2006; Utting et al, 2016). With Model-Based Testing (MBT) models representing the behavior/functionality of the system are used to automatically generate test cases (Utting and Legeard, 2006).

Model checking (Enoiu et al, 2014; Mohalik et al, 2014) has been widely used for formal verification. It has been also used in MBT. In this case, the technique requires a model of the system and a test objective defined as a property. The state space of the system is then generated and the property is evaluated. The model checker identifies a counterexample or witness to either refute or validate the property, respectively, as for usual formal verification. The counterexample or witness represents a path from the initial state to a state where the property is either refuted or verified and can be utilized as a test case (Gargantini and Heitmeyer, 1999; Rayadurgam and Heimdahl, 2001). Model checking may be limited by computational constraints in the case of large and complex systems. It may result in the well known problem of state space explosion (Baier and Katoen, 2008). Recently, the use of knowledge discovery has been proposed to address the issue of state space explosion (Pira et al, 2016, 2018). On the other hand, the use of a model checker to generate counterexamples may lead to a significant amount of duplicate tests, as a significant number of the generated counterexamples may be similar in nature (Fraser et al, 2009; Villani et al, 2019).

Graph Transformation Systems (GTS) (Heckel, 2006) is a formalism that is used for describing the dynamics of complex systems. They enable a step-by-step simulation of the system's behavior, starting from an initial state and progressing towards a state that satisfies specified objectives, by means of a well-defined path. This type of analysis is a common application of GTS and provides valuable insights into the functioning of complex systems. In GTS, there are interdependencies between the rules in each path of the sequence in the state space. The order in which the rules are executed is crucial for maintaining system consistency and correctness. Rules may have conditions and dependencies that must be met prior to their execution. The dependencies between the rules can impact the system's behavior and performance. Thus, GTS effectively describes software architectural styles through a formal and systematic approach (Thöne, 2005). Architectural styles specify systems with models of varying sizes sharing a common infrastructure and events (Pira et al, 2016). In the state space of these models, there is a specific sequence of events along the path from the initial state to the goal state. Repeating this sequence leads to the goal state. Frequent patterns, which occur in the data set with a frequency greater than a specific threshold, can be detected using data mining techniques such as Apriori (Agrawal and Srikant, 2000), and FP-Growth (Han et al, 2000).

Testing approaches, especially test case generation techniques, have been developed for systems described with GTS, incorporating data-dependency coverage criteria (Kalaee and Rafe, 2019; Rafe et al, 2022; Khan et al, 2012; Runge et al, 2013). To the best of our knowledge, all existing testing methods for GTS experience a decline in coverage when encountering large models.

The aim of this paper is to address the challenges faced by existing test generation solutions from GTS specifications and which are based on model checking. We propose a novel method where an in-depth examination of the interconnections among the transformation rules in GTS is carried out, focusing specifically on the relationships between the data dependencies. We use data mining techniques to address the issue of state space explosion in GTS. Instead of generating the entire state space, our method generates only a subset of the state space using mining methods and identifies repeating sequences of traversed states. The information gained from the training phase is then utilized to guide the exploration test. While previous work (Pira et al, 2016, 2018) has used data mining techniques to guide the exploration of state space for verification purposes, and searching for violation of safety and liveness properties, we propose a method that builds on previous work (Pira et al, 2016, 2018) and leverages data mining to generate effective test cases from GTS specifications. To avoid duplicated test cases, we incorporate a minimization function that removes duplicates and reduces the length of the test suite (defined as the concatenation of the test cases or just the set of test cases). The proposed solution has been implemented in the GROOVE (GRaph-based Object-Oriented VErification) tool set (Rensink, 2004). To assess the effectiveness of the proposed test generation method and the introduced data-dependency coverage criteria, several experiments have been conducted with three widely used case studies in the field of service-oriented architecture. Our results indicate that the proposed solution outperforms the related state-of-the-art techniques, in terms of coverage and the size of the generated test suite.

The rest of the paper is structured as follows: Section 2 discusses related work. Section 3 provides some background on different concepts used in this paper. Section 4 describes our approach for generating test cases. Our experiments for the evaluation of the proposed approach and its comparison with related work are presented in Section 5. We conclude in Section 6.

2 Related work

To address the issue of redundant test cases, several solutions using data mining techniques have been proposed (Last, 2005). These techniques can generate a set of non-redundant test cases that cover the most significant functionalities in the software. In (Muthyala and Naidu, 2011), a data mining approach was used to streamline the process of test case generation. The authors used the Weka software, which incorporates multiple data mining algorithms, and applied the K-Means clustering method to group test items. The system first produced test cases automatically and then executed the K-Means algorithm on the generated test cases, resulting in the clustering of the test cases. To determine coverage, the system randomly selected a test case from

each cluster and evaluated the coverage level. If the coverage was considered insufficient, the system increased the number of clusters and repeated the process until an acceptable coverage level is achieved. In (Saifan et al, 2016), the authors used two data mining techniques, J48 and Naive-Bayes, to group test cases into clusters. The process involved collecting test cases for a specific system and selecting a suitable coverage and complexity for the data set. The K-clustering data mining method was then used to group several test instances into a single cluster. In (Ilkhani and Abaei, 2010), a novel approach, that combines case-based reasoning (CBR) and data mining to enhance effort estimation and streamline testing, was proposed. The approach was rooted in the idea that software testing can result in the classification of test outcomes into various categories, and that these classified results, along with the implementation of CBR and data mining, can be used to predict future software test cases, thereby decreasing the cost of testing and the development expenses of comparable software. By documenting the test cases and faults encountered during testing, they can be leveraged in future software development projects to minimize testing costs.

(Acharya et al, 2015) introduces a novel approach to prioritize test cases using association rule mining. The system being tested is represented using a Unified Modeling Language (UML) Activity Diagram, which is transformed into an Activity Graph. To maintain a record of the system's history and reveal more problems, a historical data store is established. Whenever a modification is made to the system, the frequently impacted nodes are identified through the detection of recurring patterns. These patterns assist in prioritizing the test cases by identifying the nodes that are most likely to be affected. The prioritization process during regression testing also makes troubleshooting easier. A method was proposed in (Pira et al, 2016) to address the challenge of state space explosion. This method leverages data mining algorithms to analyze the behavior of the system. By employing data mining techniques, the primary objective is to efficiently mitigate the challenges posed by state space explosion. Consequently, for our specific testing requirements, we have chosen to adopt the method proposed in reference (Pira et al, 2016), as elaborated in Section 4.

A number of MBT approaches have been proposed for systems specified using the GTS at different levels. These methods vary in terms of their focus and coverage, but they all strive to offer a systematic and efficient way of testing the functionality of systems described using the GTS. The research (Gönczy et al, 2007) presents a methodology for testing service infrastructure components described in a high-level language. They use graph transformation and model checking techniques to generate state spaces and identify test sequences that meet specific requirements. The methodology is demonstrated using a case study of a fault-tolerant service broker and is applicable at the architectural level. The correlation between two common production rules within the GTS formalism has been investigated by (Heckel et al, 2011). They proposed a list of potential causal dependencies and conflicts as model-based coverage criteria. An approach for Model-Based integration testing of component-based software systems was proposed in (Heckel and Mariani, 2004). The systems are specified using GTS rules, which are depicted using UML-based notations. This approach starts by identifying all critical rule pairs, including dependent and conflicting pairs, and then focuses on testing these identified pairs. In (Khan et al, 2012), a technique was

presented for dynamically assessing the coverage of the criteria outlined in (Heckel et al, 2011) during the testing process with the utilization of the AGG tool set. In (Runge et al, 2013), an approach was proposed for generating test cases from GTS specifications using a dependency graph. The graph was obtained through a static analysis of the dependencies between rules. In a study described in (Kalaee and Rafe, 2019), the challenge of generating a test suite was framed as an optimization problem focusing on data-flow coverage criteria. The authors used various meta-heuristic search algorithms to find an optimal test suite. In (Rafe et al, 2022), a technique, that employs a Bayesian optimization algorithm (BOA) in combination with a model checker to create test cases for service-oriented systems, has been introduced. (Kalaee and Rafe, 2019; Rafe et al, 2022), similarly (Runge et al, 2013) employ a dependency graph to identify the test objectives. It has been observed that existing methods tend to exhibit a decrease in coverage in the case of large models.

MBT with model checkers is a popular approach where testing requirements are defined as reachability properties and counterexamples of violated properties serve as test paths. However, using model checkers for generating test cases has some drawbacks such as test suite minimization, test case prioritization, and state-space explosion, as model checkers were primarily developed for verification, and not test generation (Fraser et al, 2009). This paper proposes a new approach using model simulation for GTS and data mining techniques to generate test sequences, and address the aforementioned limitations.

3 Some Background

In this section, we briefly introduce some background concepts related to the proposed approach.

3.1 Graph transformation systems

GTS (Ehrig et al, 2004) is a graphical and formal modeling language that uses graph and graph transformation for describing states and behavior of systems. A GTS is represented as a triple (TG, HG, R): TG is a type graph that represents the total scheme of the system. This graph has several node types (TGN) and edge types (TGE) and two functions src: TGN $\rightarrow$ TGE and trg: TGE $\rightarrow$ TGN. These functions assign to each edge a source and a target node in turn. This graph is known as a meta-model. HG is a host graph that represents the initial configuration of a system. This graph should be an instance of the type graph. R is a set of graph transformation rules in which each rule p over an attributed type graph TG is represented by a triple (LHS, RHS, NAC) as follows: LHS (left-hand side) and RHS (right-hand side) are two graphs that specify precondition and post-condition of rule p, respectively and NAC (negative application condition) is a special configuration. Prerequisites for the execution of rule p are the absence of negative application conditions with the presence of the left-hand side. Various tools, including AGG (Taentzer, 2003), ATOM3 (Lara and Vangheluwe, 2002), VIATRA2 (Varro and Balogh, 2007), and GROOVE (Kastenberg and Rensink, 2006), are available for modeling and analyzing systems specified using GTS. As GROOVE

is the only tool that supports model checking and verification through an integrated graph-based model checker, it was selected to implement the proposed approach.

3.2 Running example

The GROOVE tool set serves as a basis, in this paper, for modeling and analyzing GTS-specified systems. As a running example throughout the paper, we consider a Hotel Management service. This system allows registered guests to reserve rooms and it generates automatically the bill at check-out.

The host graph and the type graph of the service are depicted in Figure 1, while Figure 2 depicts the corresponding graph transformation rules. The GROOVE framework merges LHS, RHS, and NAC graphs into a unified view, utilizing color coding to differentiate the original elements (Rensink et al, 2010). The common nodes and edges between the LHS and RHS graphs are represented by black coloring. Blue coloring denotes nodes and edges that are removed from the LHS graph after applying the transformation rule, while green coloring indicates nodes and edges that are newly created. NAC graphs are depicted with bold red double-bordered nodes and dashed edges. To apply a transformation rule p to a state s (a graph), all instances of LHS p in s (also known as graph matching or images of LHS p) are identified and one of them is subsequently replaced with RHS p. A graph matching is considered valid if it does not include any instance of a graph in NAC p. To fully capture the behavior of a system, its state space must be constructed by repeatedly executing all of the transformation rules on the initial state. A segment of the running example's state space is depicted in Figure 3.

3.3 Data-flow coverage criteria in graph transformation systems

The concept of data-flow testing (DFT) was first introduced by (Herman, 1976). DFT consists of choosing specific program paths to test the relationship between the definition (def for short), and usage (use for short) of data objects (Wan et al, 2018). Definitions 1 and 2 are two fundamental definitions in the context of def-use pairs and data-flow testing.

Definition 1 (Def-Use Pair)**.** *A def-use pair, denoted as* $\mathbf{du}(\ell_d, \ell_u, v)$*, refers to a control-flow path within a program. This path extends from the statement located at* ℓ_d*, where a variable named 'v' is defined, to the use statement of the same variable 'v' at* ℓ_u *Crucially, the def-use pair does not allow for any re-definitions of 'v' along the path. Such a path, connecting the def to the use statement without any intervening re-definitions, is commonly known as a def-clear path.*

Definition 2 (Data-Flow Testing)**.** *For a given def-use pair* $\mathbf{du}(\ell_d, \ell_u, v)$ *in program P, data-flow testing focuses on producing an input t that enables the execution of a specific path p. This path starts at* ℓ_d *where variable v is defined and proceeds to* ℓ_u *without encountering any re-definitions of v. A test case t satisfying these conditions is said to cover the def-use pair du, indicating that the data flow from the definition to the use of variable v has been covered.*

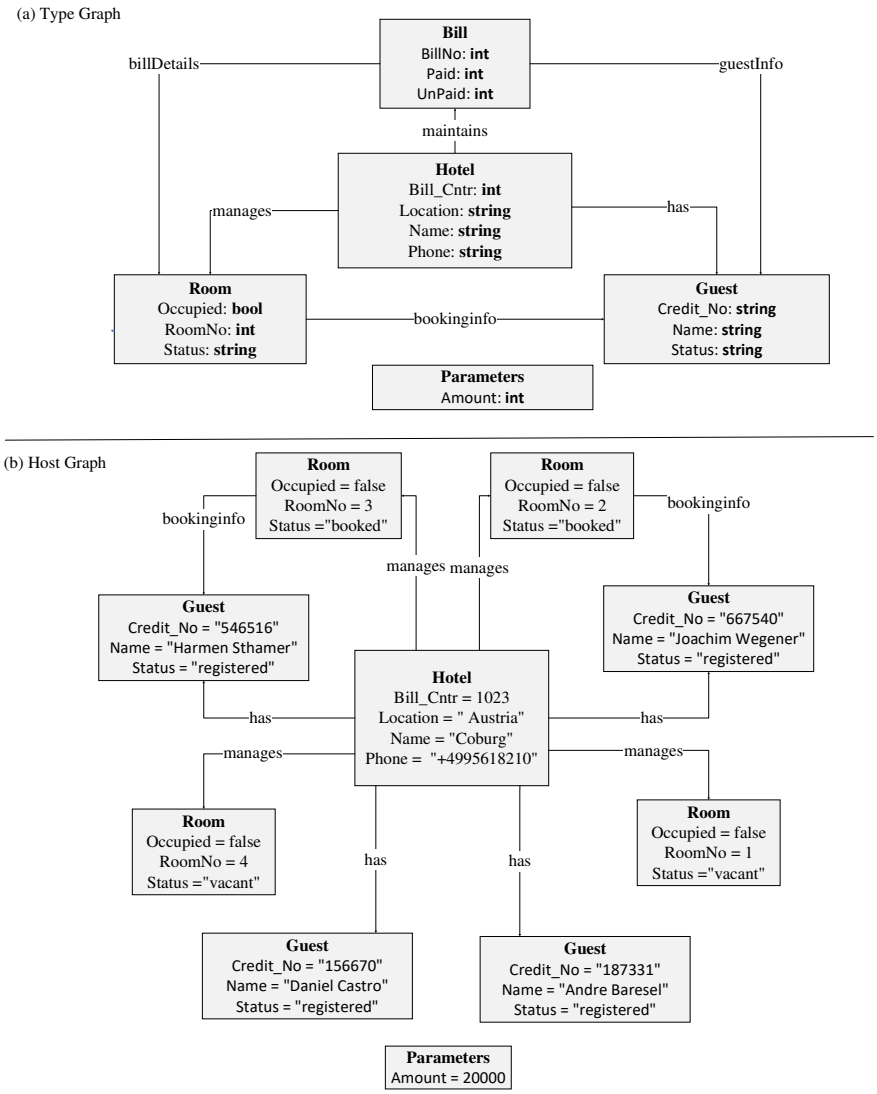

Fig. 1 The type graph(a) and host graph(b) of hotel management system

Multiple criteria for data flow testing were proposed in (Rapps and Weyuker, 1985), among which the all-def-use-path criterion is considered to be the most effective. The objective is to ensure that values generated at one point in the program are created and utilized correctly by focusing on the definition and usage of those values (Ammann and Offutt, 2008). Thus, data-flow coverage is taken into account as a test adequacy criterion in the proposed approach.

A dependency graph (DG) (Albanese, 2019) is a visual representation of the relationships between different components of a software system. It shows which

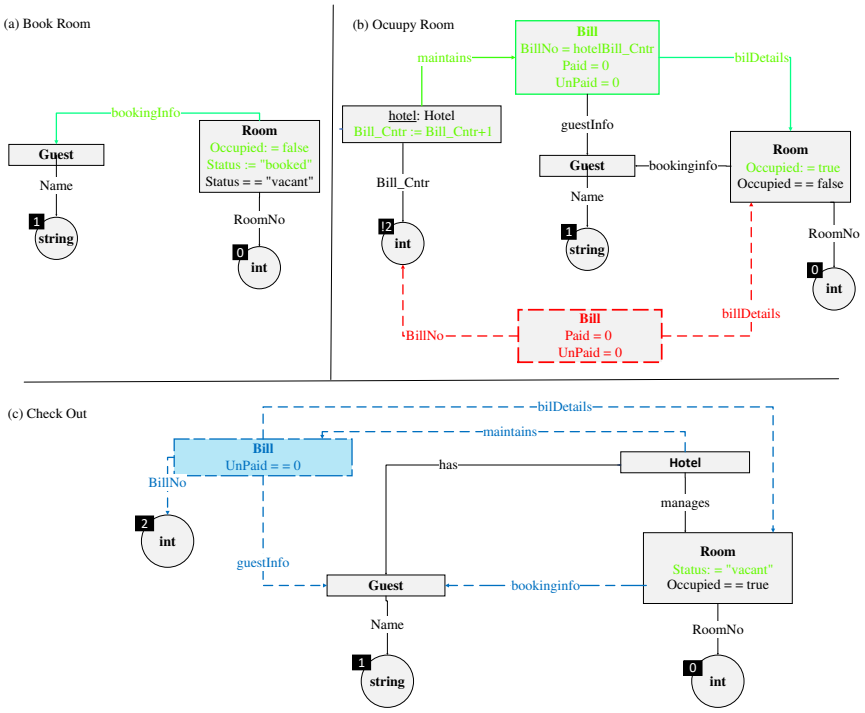

Fig. 2 Portion of the graph transformation rules of hotel management system (a) Book Room, (b) Occupy Room, (c) Checkout

components depend on which other components, and can be used to analyze and understand the structure of a software system, as well as to identify potential issues or opportunities for refactoring.

In a GTS, system operations are defined using rules. There are dependencies between these rules, which means that one rule may rely on the execution or outcome of another rule in order to function correctly. Thus, coverage criteria in GTS are determined by analyzing the dependencies between rules by creating a DG from the SUT. Let R_1 and R_2 be two rules of a graph transformation system. Definitions 3 and 4 show the conditions related to the dependency or interference of each pair of rules based on the DG in (Heckel et al, 2011).

Definition 3 (Dependency)**.** *By establishing any of the following conditions, we say that the rule R_1 is dependent on the rule R_2, and we display this dependence as $R_1 \prec R_2$:*

- *At least one edge or node from the LHS of rule R_1 is added by the RHS of rule R_2.*
- *At least one edge or node from the NAC rule R_1 is removed by the RHS rule R_2.*

Definition 4 (Interference)**.** *By establishing any of the following conditions, we say that the rule R1 interferes with the rule R2, and we display it as $R_2 \nearrow R_1$:*

- *At least one edge or node from the LHS of rule R_1 is removed by the RHS of rule R_2.*

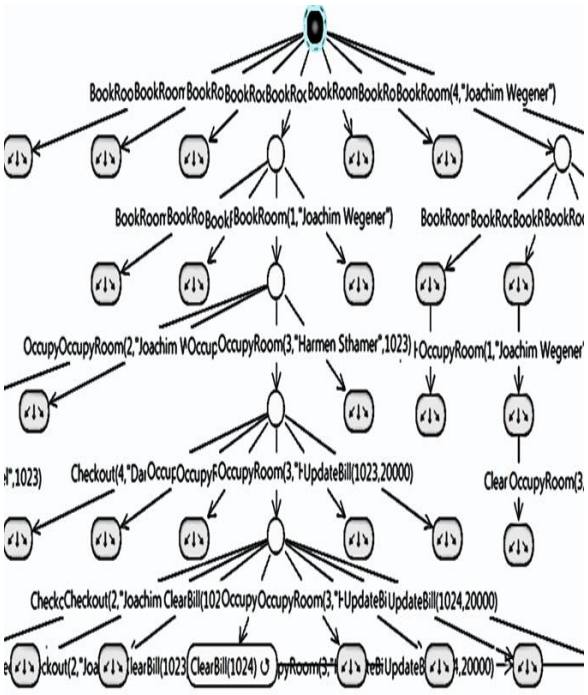

Fig. 3 A segment of the hotel management systems' state space

- *At least one edge or node from NAC rule R_1 is added by RHS rule R_2.*

Definition 5 (Dependency Graph)**.** *A dependency graph, represented as $DG = \langle G, OP, op, lab \rangle$, is a structure that includes:*

- *$G = \langle V, E, src, tar \rangle$, which is a graph*
- *OP: a set of operations*
- *op: $V \rightarrow OP$, a function that maps vertices to operation names*
- *lab: $E \rightarrow \{c, r, d\} \times \{\prec, \nearrow\} \times \{c, u, r, d\}$, a labeling function that differentiates between source and target types such as create, update, read, and delete, as well as dependency types $\prec$, $\nearrow$.*

Accordingly based on the Definition 5, if we demand all edges related to creating and reading, creating and updating, and creating and deleting in the DG, we will be covering all dependencies based on the data being defined and used subsequently (Heckel et al, 2011). These criteria are defined in Table 1 in which our purpose is to produce test cases to cover these coverage criteria.

3.4 Data-mining

Data mining is the process of extracting knowledge and patterns from massive amounts of data. It involves using advanced techniques and algorithms to uncover hidden

Table 1 GTS dependency criteria

Criterion	Coverage Criteria (Dependency)	Description
C1	Create-Read	The first rule adds an element to the host graph, which the second rule needs to run.
C2	Create-Delete	The first rule adds an element to the host graph, which the second rule removes.
C3	Create-Update	The first rule adds an element to the graph, which the second rule updates.
C4	C1 ∪ C2 ∪ C3	Test all def-use pairs (C1 ∪ C2 ∪ C3)

patterns and valuable insights from the data. Data mining encompasses a range of techniques drawn from related disciplines such as databases, statistics, machine learning, neural networks, and pattern recognition (Witten and Frank, 2002).

The key methods for discovering patterns include mining association rules and frequent patterns, classification, and clustering (Han et al, 2012). These techniques form the core of data mining and are used to uncover meaningful information from large amounts of data. Mining association rules is one of the most significant algorithms and was initially introduced in (Agrawal and Srikant, 2000; Agrawal et al, 1993). The most widely used methods for this purpose are the Apriori and FP-Growth algorithms (Han et al, 2012).

Apriori algorithm proposed by (Agrawal and Srikant, 2000) which operates on the principle that any smaller group within a frequent set must also have a high frequency. It begins by identifying single items and gradually combining them to create bigger sets that have high frequency. The Apriori approach is bottom-up, breadth-first and employs a strategy that generates and eliminates possibilities to minimize unneeded calculations.

FP-Growth algorithm, which stands for Frequent Pattern Growth proposed by (Han et al, 2000), uses an FP-tree, a tree-based data structure, to keep track of transaction data. The process starts by determining the frequency of items and constructing an FP-tree from the items with high frequency. The algorithm leverages the tree structure to produce frequent item sets with efficiency and without the need for explicit candidate generation, as seen in the Apriori algorithm. Researchers have proposed the combination of the Apriori algorithm and FP-tree structure, and results demonstrate its potential as a promising solution (Lan et al, 2009; Wu et al, 2008).

In our proposed approach, we use these algorithms to investigate a segment of the state space and extract frequent patterns.

4 The proposed test case generation approach

The proposed approach uses GTS to model the system under consideration, where each rule represents a service as known in service-oriented architecture, a function in object-oriented systems, or an event in interactive or safety-critical systems. Our approach is based on the use of data mining techniques to generate test suites, which can effectively address the state space explosion problem and redundant test cases. The proposed approach gains specific knowledge by exploring a small portion of the

state space of the system under consideration. This knowledge is subsequently used to explore the remaining portion of the state space in an efficient manner until a goal state is reached. The proposed approach consists of five phases, as illustrated in Figure 4. In the subsequent sections, we will elaborate on each of these phases in detail.

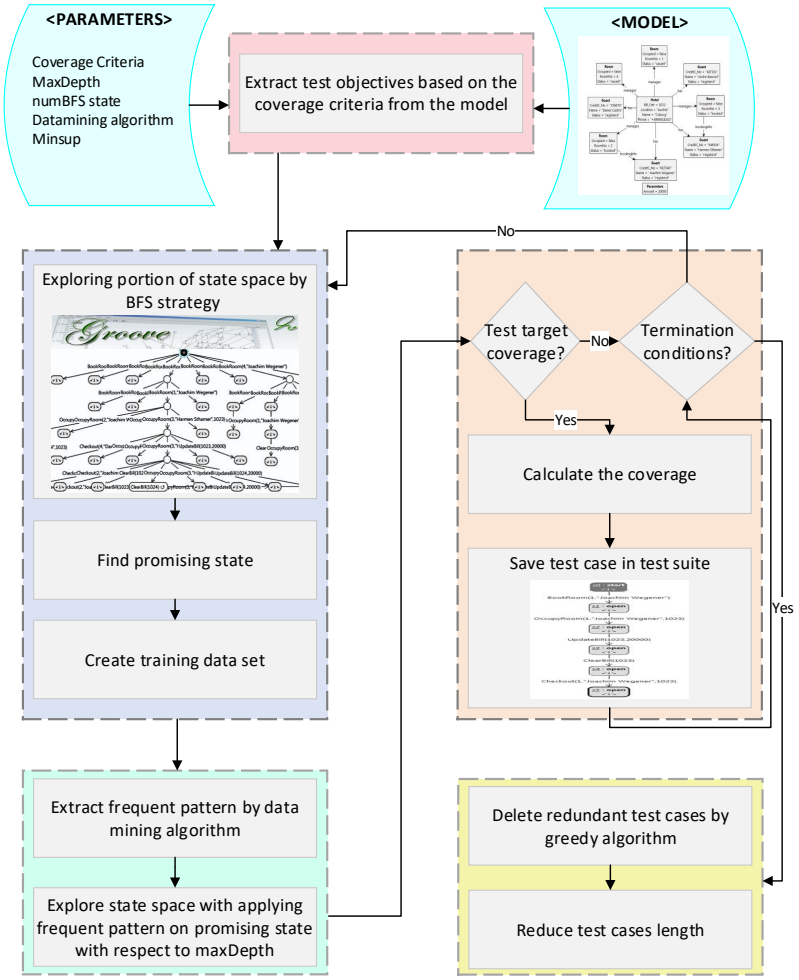

Fig. 4 The proposed approach for test suite generation using model checking and data mining algorithms

4.1 Test criterion formulation phase

Drawing on the definitions of DFT presented earlier, a test case (TC) is deemed to cover a def-use pair if it meets a specific condition, as outlined in Definitions 1 and 2. Specifically, the test case must contain a definition-clear path linking the point in the code where the variable is defined (Def) to the point where it is used (Use). This condition ensures that the variable remains in the intended state during program execution, thereby minimizing the possibility of unexpected errors or behavior. In summary, a test case is considered to cover a def-use pair if it satisfies the requirement of having a definition-clear path between the Def and Use variables. Algorithm 1 contains the pseudo-code for generating the test criterion, which is responsible for extracting the def-use pair statically.

Algorithm 1 Test objective extraction

Require: R $(GTS - rules)$
Ensure: $C1, C2, C3$ $(Test\ objectives)$
1: Define three sets: $C1\_Pairs, C2\_Pairs, C3\_Pairs$ to store du_Pairs
2: $r\_counter = 0$
3: **while** $r\_counter < R.length()$ **do**
4: Get the rule r from the set R
5: $consumed\_Set$ = set of edges that erase entities created by r
6: $produced\_Set$ = set of edges that create entities consumed by r
7: $preserved\_Set$ = set of edges that preserve entities created by r
8: $updated\_Set$ = set of edges that update entities created by r
9: $r\_counter++$
10: **end while**
11: $r\_counter = 0, dep\_r\_counter = 0$
12: **while** $r\_counter < R.length()$ **do**
13: Get the rule r from the set R
14: **while** $dep\_r\_counter < R.length()$ **do**
15: Get the rule dep_r from the set R
16: **if** $r.preserved\_Set \cap r.produced\_Set \neq 0$ **then**
17: $C1\_Pairs = du\_Pair(dep\_r, r)$ *and* $r.preserved\_Set \cap r.produced\_Set$
18: **end if**
19: **if** $r.consumed\_Set \cap r.produced\_Set \neq 0$ **then**
20: $C2\_Pairs = du\_Pair(dep\_r, r)$ *and* $r.consumed\_Set \cap r.produced\_Set$
21: **end if**
22: **if** $r.updated\_Set \cap r.produced\_Set \neq 0$ **then**
23: $C3\_Pairs = du\_Pair(dep\_r, r)$ *and* $r.updated\_Set \cap r.produced\_Set$
24: **end if**
25: $dep\_r\_counter++$
26: **end while**
27: $r\_counter++$
28: **end while**

4.2 Coverage tracking phase

The proposed approach aims at optimizing the data-flow coverage criterion by evolving test suites through the use of a data mining algorithm (introduced in Section 4.4). In order to achieve this, we must establish a representation of the desired solutions and a fitness function. This section details the problem and fitness formulation.

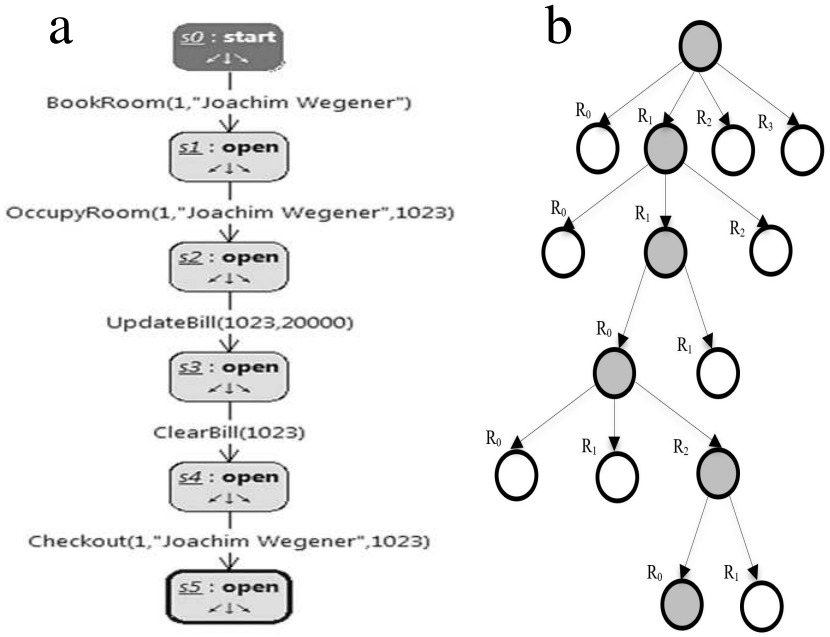

Fig. 5 The structure of the test case in the proposed method a) Subset of state space associated with the running example b) Corresponding test case to the path depicted in part a

4.2.1 Problem statement

The proposed method involves generating a test case set T that includes multiple test cases t_i, represented as $T = < t_1, t_2, ..., t_n >$. Each test case comprises a series of graph transformation rules that begin from the initial state. As shown in Figure 5, part a, the test case in the state space resembles a path with the sequence of rule execution $< R_1, R_1, R_0, R_2, R_0 >$ on the initial state, which is equivalent to Figure 5, part b. The parameter values are obtained from either the initial graph or the implementation of other rules. To create a test suite, an exhaustive exploration of the state space of a system is performed, and a data set is generated. The state space has an initial state, and candidate solutions for generating a test suite consist of a set of finite sequences of transitions within the state space starting from this initial state.

4.2.2 Fitness formulation

We evaluate each test case, denoted as t, using the F function presented in Equation 1, which counts the number of uncovered objectives by the test case, and the lower this value, the higher the test case's competence. The method used to identify the number of objectives uncovered by a test suite is presented in Algorithm 2. By analyzing each test case, the covered objectives are eliminated from the set of test objectives, and the test case is added to the final test set. The coverage of the test suite is also demonstrated by Equation 2.

$$F(t) = Test\ Objectives - number\ of\ covered\ objective\ by\ the\ t. \tag{1}$$

$$Coverage(TS) = \left(\frac{\sum_{i=0}^{TCi} \text{Covered Test Objectives}}{\text{Test Objectives}} \right) \times 100. \tag{2}$$

Algorithm 2 Fitness function

Require: Test suite T, Coverage Criteria C, du_Pairs (with respect to C) D
Ensure: Test suite coverage TS_C

```
foreach test case t in the T do
    foreach consecutive pair p of rule transition (d, u) in path of t do
        while p ∈ D && p is not covered by test targets do
            if C = C1 ||| C = C4 then
                if entities created by d ∩ entities read by u ≠ 0 then
                    TS_C.add(p)
                end if
            end if
            if C = C2 ||| C = C4 then
                if entities created by d ∩ entities deleted by u ≠ 0 then
                    TS_C.add(p)
                end if
            end if
            if C = C3 ||| C = C4 then
                if entities created by d ∩ entities updated by u ≠ 0 then
                    TS_C.add(p)
                end if
            end if
        end while
    end for
end for
return D − TS_C
```

4.3 State space exploration by Breadth-First Search strategy and creation of the training data set

The purpose is to select a test objective from a set of objectives derived from Algorithm 1. The goal is to find a path that covers the chosen objective. Each test objective consists of a rule pair that defines a relationship between a definition and its corresponding use. The solution starts by searching for the definition section. If it successfully locates the definition part, it sets the starting point of the path to the position where the definition is found and then proceeds to search for the use part. In case the definition section is not found, the starting graph remains the same as the initial state.

To achieve the desired outcome, which could involve reaching either the definition or the use part, the breadth-first search (BFS) algorithm is employed. BFS explores the states in a systematic manner, examining each level starting from 0 and progressing up to *maxl*. The value of *maxl* depends on factors such as the model's size and the number of states that have been explored. By employing BFS, the solution can systematically traverse the states and work towards achieving the objective.

Since the states at level *maxl* are closer to a goal state compared to those at lower levels, they are prioritized for further exploration (Pira et al, 2018). Among these states, there are promising states that are more likely to reach a goal state, and these are selected based on similarity to our test objectives. Once these promising states are identified, a training data set is generated from the explored states to extract the necessary knowledge.

In order to extract the necessary knowledge, we generate a training data set from the explored states by considering all paths that meet the following criteria: each path must begin at an initial state and lead to a promising state. Our approach modifies the methodology employed by (Pira et al, 2018) for creating the training data set. In this research paper, we make the assumption that the system under study can be effectively represented by a GTS. As a result, both the properties of the system and the individual states within the model's state space are graph structures. These graphs consist of nodes and edges, each of which possesses specific labels that provide additional information and context.

In the GROOVE tool set, when it comes to labeling nodes, a label (*lbl*) for a particular node (n) is denoted by a self-loop edge on that node, which itself is labeled with the corresponding (*lbl*). This means that the similarity between nodes is determined by examining only the outgoing edges of each node, disregarding any incoming edges. Let's consider a scenario where the goal state, denoted as t can either be a part of the definition or the usage. At the *maxl*, a state s is considered promising if its similarity to the goal state t is higher compared to other states at the same level. Algorithm 3 focuses on calculating the similarity between the states in *maxl* and the goal state. Once the similarity of the states at *maxl* is computed, a subset of states with the highest similarity is selected as the promising states. By choosing these promising states, the training data set is composed of all paths leading to these selected states.

Algorithm 3 Similarity function

Require: s (a state in $maxl$), t (a goal state (def/use))
Ensure: Similarity score

Let G_s and G_t the corresponding graph of s and t
foreach node n_s in G_s **do**
    foreach node n_t in G_t **do**
        $pair.n_s = n_s, pair.n_t = n_t$
        $allPairs.append(pair)$
    end for
end for
foreach $pair$ in $allPairs$ **do**
    $pairs.similarityCount+ =$ number of pair edges with equal labels
end for
Sort $allPairs$ based on $similarityCount$ in descending order
foreach $pair$ in $allPairs$ **do**
    $similarityCounter+ =$ unique pairs with the highest $similarityCount$
end for
if G_t has NAC **then**
    foreach pair of node in between $G_t's$ NAC and G_s **do**
        $NACCounter = numberofpairedgeswithequallabels$
    end for
end if
return $similarityCounter - NACCounter$

4.4 Data mining phase

We adopt a data mining-based approach to effectively identify frequent patterns within the training data set. To achieve this, we have enhanced the existing FP-growth and Apriori algorithms. The primary focus of our modified algorithms is to maintain the order of rules during the exploration of the state space. Unlike traditional methods that generate multiple frequent patterns, our approach aims to extract a single frequent pattern with the highest occurrence frequency. This selected pattern is then leveraged wisely to explore the remaining state space until a specific termination criterion is met. Termination criteria include achieving full coverage or reaching a predefined time limit.

Let us assume that the frequent pattern is denoted as $r_0, r_1, ..., r_k$, where each element (r_i) belongs to the set R ($0 \leq i \leq k$). Furthermore, we identify promising states using the set S. Additionally, we introduce a parameter called $maxDepth$, which represents the maximum depth allowed for intelligent exploration.

For each state s present in S, we execute the following procedure: we sequentially apply each rule from the frequent pattern ($r0, r1, ..., rk$) to the state s. This process continues until the termination criterion is satisfied or the maximum depth is reached. In situations where the frequent pattern is not applicable to a specific state, we employ a strategy of randomly selecting successive states and proceeding with the exploration. This approach allows us to continue the search for an applicable rule within the state

space. By opening states in a random manner, we increase the chances of finding a state where the frequent pattern can be effectively applied. This dynamic approach enhances the overall efficiency and effectiveness of the algorithm. Whenever a response is obtained, we consider the path from the initial state to the goal state. In our case, the goal state may refer to the definition/use part of the test objective. To provide a clearer understanding, we present the pseudo-code in Algorithm 4.

Algorithm 4 Intelligent State Space Exploration with Data Mining

Require: FP (generated frequent pattern with Apriori/FP-Growth Algorithm), g (goal state), S (set of promising states), $maxDepth$ (maximum depth of exploration specified by the user

Ensure: The path from an initial state to the goal state

```
1:  foreach state s in S do
2:      currentState = s, depth = 0
3:      while depth ≤ maxDepth do
4:          foerach rule r in FP do
5:              apply rule r to currentState and update it
6:              if rule r is not applicable to current state then
7:                  Select successive state randomly and update currentState
8:                  Break to next iteration
9:              end if
10:             if currentState = g then
11:                 return path from initial state leading to currentState
12:             end if
13:         end for
14:         depth+ = FP.length
15:     end while
16: end for
17: return Null
```

4.5 Test suite minimization phase

We introduce a two-step approach for minimizing test suites to address the issue of redundant test cases. This approach focuses on enhancing the effectiveness of the test suite while optimizing resources and saving time.

The first step involves eliminating redundant test cases by selecting test cases from the test suite using a greedy strategy. We examine the test suite and compare the covered objectives of each test case. If we find that the covered objectives of one test case are completely contained within the covered objectives of another test case we consider the second test case redundant and remove it from the test suite.

The second step revolves around reducing the length of test cases. Our aim is to shorten the test case path by focusing only on the portion of the path starting from the initial state and ending at the state where all Def-Use pairs have been visited.

We disregard the remaining portion of the path since it does not contribute to the coverage of the objectives.

By implementing these two steps, we effectively minimize the test suite by eliminating redundant test cases and reducing the length of the remaining test cases. This approach enhances the efficiency and effectiveness of the testing process, optimizing resource utilization and save valuable time.

5 Evaluation

In this section, we first examine the proposed approach, and the data mining algorithms it uses, to fine tune it and carefully analyze the results obtained from experiments with three case studies. In the second part of the section, to assess the effectiveness of our approach, we compare it with related test case generation methods, such as the search-based (Kalaee and Rafe, 2019), machine learning-based (Rafe et al, 2022), as well as model verification based test generation techniques (Wan et al, 2018).

Given the approximate nature of the algorithms proposed in this paper, each experiment is repeated 10 times to ensure reliable outcomes, and the average result is recorded for further analysis. To evaluate the efficiency of the algorithm, we use Mann-Whitney statistical tests and U tests. These tests use a standard method known as effective size $\hat{A}_{12}$, introduced by Vargha and Delaney (Arcuri and Briand, 2011). The $\hat{A}_{12}$ value represents a non-parametric measure of the effective size, and we use two methods to compare the probabilities of higher (or lower) values for coverage (or test suite set size).

If $\hat{A}_{12}$ is less than 0.5, it indicates that the occurrence of results obtained from the first method is less likely compared to the second method. A value of $\hat{A}_{12}$ equal to 0.5 suggests equal probability, while $\hat{A}_{12}$ greater than 0.5 indicates a higher probability for the first method. Furthermore, we assessed the significance level, denoted as the $P-value$, which, if less than 0.05, indicates a significant difference between the results of the two methods. If the $P-value$ is greater than 0.05, additional tests are required to reach a conclusion.

All experiments were conducted on a system equipped with an Intel(R) Xeon(R) CPU E5-2620 V4 operating at 2.10 GHz, with 16 GB of RAM.

5.1 Case studies

We selected three well known case studies, the Online Shopping System (Engels et al, 2007), Bug Tracker System (Runge et al, 2013), and Travel Agency System (Rafe, 2013). Each of these case studies is known for its large state space. Using these case studies, we aim at evaluating the effectiveness of our approach in handling such challenging scenarios. The Online Shopping System (OSS) allows customers to purchase products online and pay with a credit card. The Bug Tracker System (BTS) manages software bugs during development projects. The Travel Agency System (TAS) books flights and hotels for clients based on their preferences and budget. The specifications of the chosen case studies, including the number of rules and test objectives for each criterion, are presented in Table 2. These models are based on real-world scenarios

with large state spaces. The criteria are labeled as C1, C2, C3, and C4, which correspond to Create-Read, Create-Delete, Create-Update, and the combination of C1, C2, and C3, respectively.

Table 2 Case studies details

Case Study	#Rule	#Test Objective			
		Create-Read C1	Create-Delete C2	Create-Update C3	All Dependencies C4
OSS	19	28	7	12	47
BTS	32	73	13	5	91
TAS	43	66	10	10	86

5.2 Experiment settings

In order to optimize the efficiency of the proposed solution, we conducted extensive experiments to determine the optimal parameter values. Table 3 presents the results obtained from these experiments, including empirical values for the maximum test case length, and the number of states traversed by the BFS algorithm for various case studies. The experimental values for the parameters were determined based on the outcomes of these experiments.

To determine the suitable value for the maximum length of the test case parameter, different values were tested. For values of the maximum length set to less than 50, the coverage significantly decreased. On the other hand, setting the maximum length to more than 50 did not result in any further increase in coverage. However, it did increase the time required to generate the test case set. Consequently, a value of 50 was determined to be optimal for this parameter.

The number of states traversed by the BFS algorithm plays a crucial role in the learning process's efficiency, and its value depends on the size of the model. Larger models have a larger state space, which in turn necessitates a higher number of states to be traversed by the BFS algorithm for constructing the training data set.

The efficiency of data mining algorithms is significantly influenced by the minimum support percentage (*minsup*) parameter. To identify the optimal value for this parameter, we conducted a series of tests to measure the coverage achieved for different (*minsup*) values. Subsequently, the value yielding the highest coverage was selected for further analysis.

Table 3 Fine-tuning the parameters for test case generation

Case Study	Max TC length	# State BFS	Apriori's minsup	FP-Growth's minsup
OSS	50	5000	0.2	0.2
BTS	50	25000	0.5	0.3
TAS	50	10000	0.5	0.7

5.3 Experiment results

This section presents an overview of the outcomes obtained from our experiments.

5.3.1 Achieved coverage

When evaluating the effectiveness of a test generation method, it is common to compare its achieved coverage with that of a baseline technique such as Random Testing (RT) (Arcuri and Briand, 2011). Random Testing is a straightforward technique that utilizes random search in state space without any specific guidance to select test cases. This comparison allows us to assess the performance of the proposed approach in comparison to the baseline method

Table 4 presents the results obtained from applying the data mining-based test generation approach to the OSS, BTS, and TAS case studies. The table comprises various columns that contain significant data related to the experiments. It displays the data mining technique used, the selected evaluation criterion, and the coverage results represented by the mean, median, and variance values. The mean value represents the average coverage achieved during the experiments. The median value shows the middle coverage value, separating the higher and lower results. The variance value indicates the extent of variation or spread in the coverage results obtained.

From table 4, among the three algorithms, the FP-Growth algorithm consistently achieved the highest coverage, demonstrating perfect coverage (100%) across all cases and criteria. It outperformed both the RT and Apriori algorithms in terms of coverage. The second-best algorithm in terms of coverage is the Apriori algorithm. The RT algorithm exhibited moderate coverage results, with coverage percentages varying across different cases and criteria. It did not surpass the coverage performance of either the FP-Growth or Apriori algorithms.

Figure 6 presents the average coverage achieved per coverage criterion for each case study.

In Table 5, the performance of the Fp-Growth algorithm is evaluated by comparing its average coverage with that of the other algorithms. This analysis sheds light on how effectively Fp-Growth performs in terms of coverage in comparison to its counterparts. Specifically, the value $\hat{A}_{12}$ represents the estimated probability of Fp-Growth achieving better coverage compared to the other algorithms.

Referring to Table 5, we observe that in 8 cases, Fp-Growth and the Apriori and RT algorithm yield equal results (i.e., $\hat{A}_{12} = 0.5$). Furthermore, it becomes evident that in 16 cases Fp-Growth has higher coverage. To visually represent the effect size of the average coverage on the selected case studies, we utilize box plots, as depicted in Figure 7.

5.3.2 Test suite size

In this section, we will compare the algorithms according to the size of the test suites they generate. Table 6 shows the results of our experiments conducted with the selected case studies. The tables present relevant information, including the data mining algorithm used, the coverage criterion used, the number of test cases generated,

Table 4 Analyzing the effectiveness of the data mining algorithms through the achieved coverage

		Case I: OSS			Case II: BTS			Case III: TAS		
Algorithm	Criterion	Mean	Median	Variance	Mean	Median	Variance	Mean	Median	Variance
RT	C1	81.07	82.14	88.33	75.89	75.89	10.30	92.12	92.12	45.41
	C2	74.28	74.28	141	70.76	76.92	65.55	83	80	110.75
	C3	92.5	91.66	30.99	**100**	**100**	**0**	89	90	44.62
	C4	87.02	87.23	34.94	75.27	76.92	65.55	92.09	92.09	45.92
Apriori	C1	**100**	**100**	**0**	88.35	87.67	19.57	88.03	87.87	8.53
	C2	**100**	**100**	**0**	78.46	76.92	30.32	**100**	**100**	**0**
	C3	**100**	**100**	**0**	**100**	**100**	**0**	**100**	**100**	**0**
	C4	**100**	**100**	**0**	86.37	86.37	1.58	98.13	97.67	1.29
FP-Growth	C1	**100**	**100**	**0**	**100**	**100**	**0**	**100**	**100**	**0**
	C2	**100**	**100**	**0**	**100**	**100**	**0**	**100**	**100**	**0**
	C3	**100**	**100**	**0**	**100**	**100**	**0**	**100**	**100**	**0**
	C4	**100**	**100**	**0**	**100**	**100**	**0**	**100**	**100**	**0**

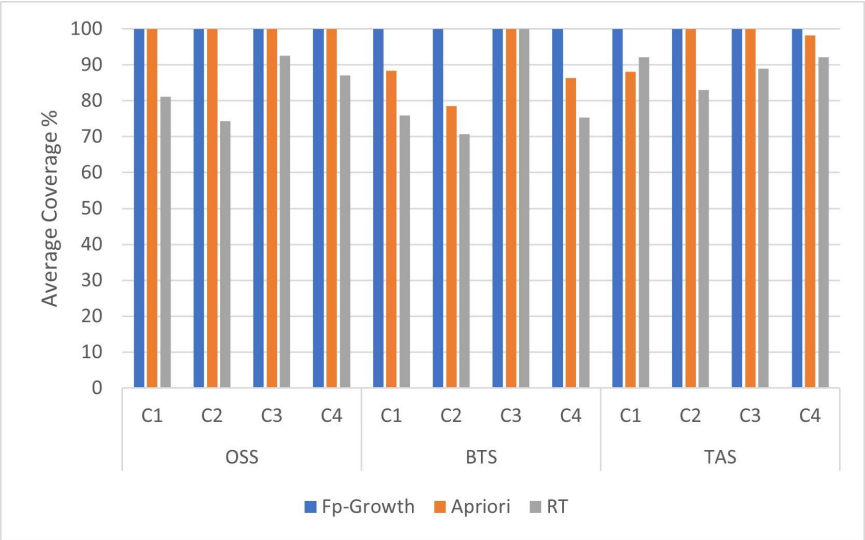

Fig. 6 The average coverage achieved by the different algorithms

Table 5 Effect size comparison of average coverage by FP-Growth and the other algorithms per case-study (FP

		Case I: OSS		Case II: BTS		Case III: TAS	
Algorithm	Criterion	$P-Value$	$\hat{A}_{12}$	$P-Value$	$\hat{A}_{12}$	$P-Value$	$\hat{A}_{12}$
RT	C1	6.2944×10^{-5}	**1**	6.2944×10^{-5}	**1**	7.4679×10^{-4}	**0.9**
	C2	5.3075×10^{-5}	**1**	4.8800×10^{-5}	**1**	7.1229×10^{-4}	**0.9**
	C3	1.9753×10^{-3}	**0.85**	-	0.5	6.5058×10^{-4}	**0.9**
	C4	6.3403×10^{-3}	**1**	2.2381×10^{-4}	**0.95**	2.2676×10^{-4}	**0.95**
Apriori	C1	-	0.5	6.2944×10^{-5}	**1**	6.1133×10^{-5}	**1**
	C2	-	0.5	5.4699×10^{-5}	**1**	-	0.5
	C3	-	0.5	-	0.5	-	0.5
	C4	-	0.5	5.9802×10^{-5}	**1**	6.9964×10^{-4}	**0.9**

Note: The notation $\hat{A}_{12} < 0.5$ indicates that FP-Growth resulted in lower coverage, $\hat{A}_{12} = 0.5$ denotes equal coverage, and $\hat{A}_{12} > 0.5$ indicates higher coverage than the other algorithms. Effect sizes with statistically significant differences ($p - Value < 0.05$) are highlighted in bold.

and the length of those test cases. Additionally, the total length of the test suite, which represents the sum of the lengths of all test cases, is displayed in the last column.

It is important to note that when the coverage achieved by different algorithms varies significantly, comparing the size of their test suites becomes less relevant. Our approach primarily focuses on coverage-based test generation, aiming to achieve higher coverage. Therefore, a higher coverage implies a better performance of the algorithm. As a result, we only examine the test suite size for cases in which we did not find any statistically significant difference in coverage, denoted by $\hat{A}_{12} = 0.5$ in Table 5.

Table 6 Comparison of the different algorithms with respect to the size of the generated test suite

		Case I: OSS			Case II: BTS			Case III: TAS		
Algorithm	Criterion	#TC	TC Length	TS Length	#TC	TC Length	TS Length	#TC	TC Length	TS Length
RT	C1	4.5	25	112.5	23.8	23.8	566.4	11.2	23.8	266.6
	C2	4	10	40	5.8	22	127.6	2	34.5	69
	C3	3	17.4	52.2	2.2	17.4	38.28	2.6	22.2	57.72
	C4	5.1	25.8	131.6	11.4	25.4	289.6	15.4	20.3	312.6
Apriori	C1	6	10	60	3	33	99	3	29	87
	C2	9	4	36	5	57	35	74	7	28
	C3	7.6	5	38	3	5 15	4	10		40
	C4	5	12	60	3	38.4	115.2	‘3	36	108
FP-Growth	C1	6	10	60	3	33.3	99.9	3.2	31.3	100.2
	C2	4	9.1	36.4	5	7	35	4	7	28
	C3	5	7.3	36.5	3	5	15	4.5	9.7	46.35
	C4	5	12	60	3.7	40.9	151.33	3.1	36.7	113.77

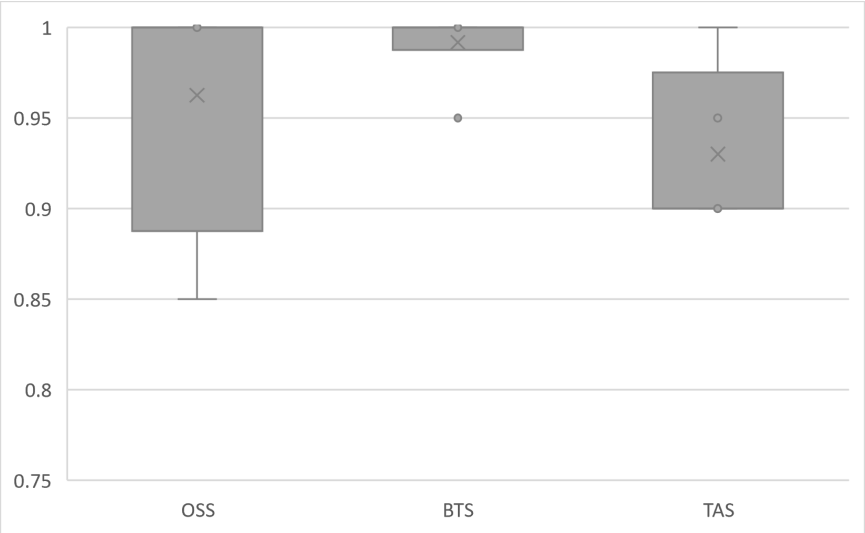

Fig. 7 Illustration of the variation in coverage across different case studies using box plots: The Fp-Growth algorithm demonstrates a higher likelihood of achieving superior coverage compared to the counterpart algorithms.

Table 7 illustrates the effect size of the FP-Growth algorithm in terms of test suite size compared to other algorithms. The table includes p-values that indicate whether the differences observed between these algorithms are statistically significant or simply random. It is evident that in 1 out of 8 cases, FP-Growth exhibits a lower average test suite size ($\hat{A}_{12} < 0.5$) in comparison with Random testing. However, it is worth noting that in the remaining 7 cases all $p - Values$ in the table are greater than 0.05. This implies that there is not enough statistical evidence to confidently assert that these algorithms differ significantly in terms of test suite size.

5.4 Comparison with other test case generation techniques

In this section, we evaluate the effectiveness of our proposed data mining-based test generation approach by comparing it with model checking-based, search-based, and Bayesian-based test generation approaches. Our evaluation focuses on two key aspects: achieved coverage and test suite size.

5.4.1 Model checking-based test (MCT) generation approach

To assess the scalability of our approach, we conduct a comparative analysis with the conventional technique of test case generation assisted by model checking. While model checking has been previously employed for data flow testing (Agrawal and Srikant, 2000), these methods directly operate at the program source code level to generate test objectives based on data-flow criteria. In contrast, our approach focuses on the abstract model level.

Table 7 Effect size comparison of test suite for FP-Growth and the other algorithms where the same coverage is achieved

Case Study	Criterion	Algorithm	$P-Value$	$\hat{A}_{12}$
Case I: OSS	C1	Apriori	0.8206	0.47
	C2	Apriori	0.6997	0.44
	C3	Apriori	0.9691	0.49
	C4	Apriori	0.9253	0.49
Case II: BTS	C3	RT	0.0022	**0**
	C3	Apriori	0.5656	0.42
Case III: TAS	C2	Apriori	0.9256	0.48
	C3	Apriori	0.9086	0.52

Note: The notation $\hat{A}_{12} < 0.5$ indicates that FP-Growth resulted in lower , $\hat{A}_{12} = 0.5$ denotes equal , and $\hat{A}_{12} > 0.5$ indicates higher test suite size than the other algorithms. Effect sizes with statistically significant differences ($p - Value < 0.05$) are highlighted in bold.

In the context of Model Checking-assisted Test case generation, the test objectives are defined as a collection of trap properties formulated using temporal logic formulas like Linear Temporal Logic (LTL). A trap property serves as a clever trick to compel the model checker to search for a counterexample that demonstrates the achievement of the test objective. This counterexample is subsequently interpreted as a test case.

To conduct a comparison between our approach and the traditional test generation method based on model checking, we used the Algorithm in (Kalaee and Rafe, 2019) implemented in GROOVE. The search strategy employed by GROOVE to explore counterexample paths is known as best-first search (BFS). The results of the comparison between the FP-Growth algorithm and MCT are presented in Table 8. In this evaluation, the coverage criterion used is C4. The test suite generation time is limited to 30 minutes, allowing for a fair comparison of the coverage achieved within a specific time frame.

According to Table 8, it is clear that MCT encounters a memory issue and fails to generate the test suite. This limitation arises as the model state space expands. Our proposed method, FP-Growth, exhibits better scalability.

5.4.2 Search-based and Bayesian-based test case generation

Recent studies related to our proposed approach are search-based (Kalaee and Rafe, 2019) and Bayesian-based testing (Rafe et al, 2022) test generation.

In (Kalaee and Rafe, 2019), search algorithms are used to tackle the challenges associated with MCT Test case generation and to maximize coverage. These algorithms, such as Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Bat Algorithm (BA), Gravitational Search Algorithm (GSA), and a hybrid algorithm combining GA and PSO (HGAPSO), are integrated into the GROOVE tool set. The focus is on covering data flow in software models specified using the GTS.

On the other hand, (Rafe et al, 2022) proposes a Bayesian-based testing approach that leverages Bayesian optimization techniques to effectively handle the state space.

Table 8 Comparison of FP-Growth algorithm with the related state-of-the-art test generation techniques with respect to the achieved coverage for 30 minutes of execution

		Case I: OSS			Case II: BTS			Case III: TAS		
Reference	Algorithm	Mean	Median	Variance	Mean	Median	Variance	Mean	Median	Variance
(Agrawal and Srikant, 2000)	MCT	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory	Out of memory
(Kalaee and Rafe, 2019)	HGAPSO	**100**	**100**	**0**	87.68	85.71	33.75	96.50	98.25	21.65
	GA	**100**	**100**	**0**	85.15	84.61	30.89	98.25	100	10
	PSO	91.90	91.48	9.85	82.08	81.31	17.70	84.99	84.3	20.85
	BA	93.39	93.61	6.49	77.87	77.47	23.22	85.11	85.46	23.38
	GSA	92.33	91.48	11.28	78.56	77.47	16.97	87.2	87.2	9.01
(Rafe et al, 2022)	c-BOA	90.4	90.42	33.95	86.48	86.48	1.32	**100**	**100**	**0**
	tp-BOA	91.70	93.61	29.48	90	89.01	10.28	**100**	**100**	**0**
	n-BOA	66.59	66.59	29.67	89.01	89.01	8.12	75.46	69.76	68.45
	FP-Growth	**100**	**100**	**0**	86.92	86.81	1.19	97.67	100	24.78

This method aims to optimize the selection of test cases and achieve better coverage. They utilize three distinct structures for the Bayesian Network: c-BOA, tp-BOA, and n-BOA.

To evaluate the achieved coverage, Table 8 presents a comparison between our proposed method and the search-based testing approach by (Kalaee and Rafe, 2019), as well as the Bayesian-based testing approach by (Kalaee and Rafe, 2019), for selected case studies. The test suite generation time is limited to 30 minutes, ensuring a fair comparison of the coverage achieved within the specified time frame. The coverage criterion used for this comparison is C4. The objective is to determine which method achieves higher coverage within the given time constraint.

In Table 9, the performance of the Fp-Growth algorithm is evaluated by comparing its average coverage with that of the other algorithms. This analysis sheds light on how effectively Fp-Growth performs in terms of coverage in comparison to its counterparts. Specifically, the value $\hat{A}_{12}$ represents the estimated probability of Fp-Growth achieving better coverage compared to the state-of-the-art testing techniques.

Based on the findings presented in Table 9, it is clear that Fp-Growth demonstrates significant improvement. The effect sizes range from 0.76 to 1, indicating a strong impact. To visually represent the effect size of the average coverage on the selected case studies, we utilize box plots, as depicted in Figure 8.

However, we observe that in 2 cases, Fp-Growth, HGAPSO, and GA algorithm yield equal results (i.e., $\hat{A}_{12} = 0.5$). Furthermore, in 11 out of 24 cases, the $p - Values$ surpass the threshold of 0.05. In these specific instances, there is insufficient statistical evidence to confidently assert a difference between Fp-Growth and the other methods in terms of coverage. It is important to acknowledge that such occurrences are more likely to transpire when the subject pool is limited. Therefore, under these circumstances, it is not possible to establish a fair and conclusive comparison. However, it may be worth considering the comparison of test suite size as an alternative criterion to determine the superiority of algorithms. For example, we can consider the test suite size as an alternative criterion to compare algorithms.

Table 10 displays the results obtained from our experimental analysis concerning test suite size in the selected case studies. The table provides essential information, including the algorithm utilized, the number of test cases generated, and the length of each test case. Furthermore, the last column presents the total length of the test suite, which represents the summation of all test cases lengths.

The table serves as a valuable reference for evaluating and comparing the different algorithms in terms of test suite size. By examining the number of test cases and their respective lengths, we can assess the efficiency and comprehensiveness of each test case generation technique.

Table 11 presents an analysis of the effect size of the FP-Growth algorithm concerning test suite size in comparison to other algorithms. It is indeed crucial to consider the varying coverage achieved by different algorithms when evaluating the relevance of comparing test suite sizes. In our approach, which prioritizes coverage-based test generation with the objective of achieving higher coverage, a higher coverage indicates superior algorithm performance. Therefore, the examination of test suite size becomes particularly significant when there is no statistically significant difference in coverage

Table 9 Effect size comparison of average coverage for FP-Growth and the related state-of-the-art test generation techniques per case study

		Case I: OSS		Case II: BTS		Case III: TAS	
Reference	Algorithm	$P-Value$	$\hat{A}_{12}$	$P-Value$	$\hat{A}_{12}$	$P-Value$	$\hat{A}_{12}$
(Agrawal and Srikant, 2000)	MCT	NA	NA	NA	NA	NA	NA
(Kalaee and Rafe, 2019)	HGAPSO	-	0.5	0.7244	0.45	0.4421	0.59
	GA	-	0.5	0.1320	0.70	0.8243	0.53
	PSO	6.2944×10^{-5}	**1**	1.9753×10^{-3}	**0.76**	6.2944×10^{-5}	**0.97**
	BA	6.2944×10^{-5}	**1**	7.4679×10^{-4}	**0.9**	6.3403×10^{-5}	**1**
	GSA	2.2676×10^{-4}	**0.95**	2.2381×10^{-4}	**0.92**	5.9802×10^{-5}	**0.98**
(Rafe et al, 2022)	c-BOA	1.0131×10^{-2}	**0.84**	0.7237	0.55	0.0779	0.35
	tp-BOA	4.6744×10^{-3}	**0.86**	0.0831	0.27	0.0779	0.35
	n-BOA	1.6025×10^{-4}	**1**	0.2000	0.33	1.8528×10^{-4}	**0.98**

Note: The notation $\hat{A}_{12} < 0.5$ indicates that FP-Growth resulted in lower coverage, $\hat{A}_{12} = 0.5$ denotes equal coverage, and $\hat{A}_{12} > 0.5$ indicates higher coverage than the other algorithms. Effect sizes with statistically significant differences ($p - Value < 0.05$) are highlighted in bold.

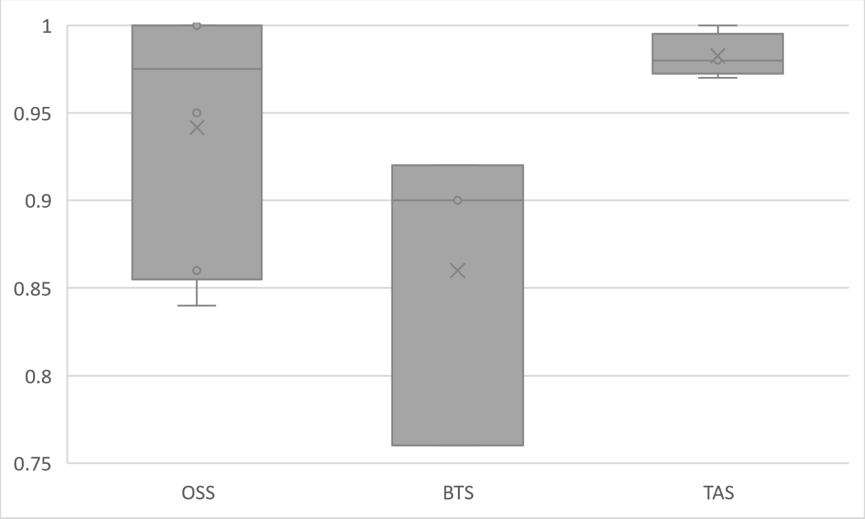

Fig. 8 Illustration of the variation in coverage across different case studies using box plots: The Fp-Growth algorithm demonstrates a higher likelihood of achieving superior coverage compared to the counterpart algorithms.

among the compared algorithms or where the selected algorithms achieve the same coverage. These scenarios are denoted by $p - Value > 0.5$ and $\hat{A}_{12} = 0.5$ in Table 9, respectively.

From Table 11, it becomes apparent that in 9 out of 11 cases, FP-Growth demonstrates a lower average test suite size ($\hat{A}_{12} < 0.5$) in comparison to state-of-the-art testing techniques. However, it is important to acknowledge that in the remaining 2

Table 10 Comparison of the different algorithms with respect to the size of the generated test suite

Reference	Algorithm	Case I: OSS			Case II: BTS			Case III: TAS		
		#TC	TC Length	TS Length	#TC	TC Length	TS Length	#TC	TC Length	TS Length
(Agrawal and Srikant, 2000)	MCT	NA	NA	NA	NA	NA	NA	NA	NA	NA
(Kalaee and Rafe, 2019)	HGAPSO	2.3	24.9	57.3	6.1	39.58	221.4	3.8	39.4	149.6
	GA	2.5	24.86	62.1	6.1	39.86	243.1	4	39.6	158.4
	PSO	2.1	24.48	51.1	5.1	39.08	199.4	3.6	40	144
	BA	2.2	23.76	52.4	4.7	38.08	181.8	3.8	39.15	148.6
	GSA	2.5	24.44	60.9	4.7	38.81	182.6	3.8	38.17	144.7
(Rafe et al, 2022)	c-BOA	3	25	75	5.8	40	232	5.9	40	236
	tp-BOA	31	25	77.5	6.4	40	256	5.9	40	236
	n-BOA	2.1	25	52.5	5.7	40	228	2.9	40	116
	FP-Growth	5	12	60	3.7	40.9	151.33	3.1	36.7	113.77

cases, all the $p-Values$ displayed in the table exceed the threshold of 0.05. This indicates that there is insufficient statistical evidence to confidently conclude that these algorithms significantly differ in terms of test suite size.

Table 11 Effect size comparison of test suite between FP-Growth and the state-of-the-art testing technique where the selected algorithms achieve the same coverage or there is no significant differences between their coverage

Case Study	Algorithm	$P-Value$	$\hat{A}_{12}$
Case I: OSS	HGAPSO	0.9072	0.52
	GA	0.9255	0.48
Case II: BTS	HGAPSO	1.5475×10^{-4}	**0**
	GA	1.6022×10^{-4}	**0**
	c-BOA	1.6025×10^{-4}	**0**
	tp-BOA	1.7155×10^{-4}	**0**
	n-BOA	1.5748×10^{-4}	**0**
Case III: TAS	HGAPSO	1.6118×10^{-4}	**0**
	GA	1.6684×10^{-4}	**0**
	c-BOA	1.6399×10^{-4}	**0**
	tp-BOA	1.7265×10^{-4}	**0**

Note: The notation $\hat{A}_{12} < 0.5$ indicates that FP-Growth resulted in lower , $\hat{A}_{12} = 0.5$ denotes equal , and $\hat{A}_{12} > 0.5$ indicates higher test suite size than the other algorithms. Effect sizes with statistically significant differences ($p - Value < 0.05$) are highlighted in bold.

6 Conclusion

Software testing is an essential activity in the software development life cycle and plays an important role in improving software quality. Software testing can be costly and time-consuming. MBT has emerged as a new paradigm. This enables automation for both test case generation and execution. Model checking is a widely used technique within the field of MBT to automate test case generation. Model checkers are used to explore the entire state space in search of the desired targets. However, a major obstacle faced by such tools is the state space explosion for large and complex systems. Furthermore, these test generation techniques often result in repetitive test cases, another drawback.

In this paper, we proposed an approach to improve test case generation, from graph transformation systems specifications, using model checkers. The proposed solution uses data mining algorithms to avoid exhaustive exploration of the state space. Knowledge discovery is used to gain valuable insights into the application of graph transformation rules for a smart navigation of the state space. Graph transformation systems exhibit interdependencies between rules within ordered sequences (of rules)

used in the state space. Some of these ordered sequences of rules are recurrently present in different parts of the state space. Consequently, the proposed solution scans a portion of the desired model state space and uses data mining techniques to identify repeated patterns. This acquired knowledge is then leveraged to navigate wisely the remaining model state space. Two data mining algorithms, Apriori and FP-Growth, have been presented as part of this solution.

The proposed solution has been implemented. We integrated our approach into the widely-used model checker GROOVE. This adaptation allows us to effectively apply our techniques and evaluate their performance. To demonstrate its effectiveness, the proposed solution has been compared with related state-of-the-art techniques. The test results in section 5 confirm the excellent performance of the proposed solution with respect to coverage and test suite size.

As future work, we are aiming at:

- Dynamically setting the Minsup parameter value: The Minsup parameter, which greatly influences the extracted repetitive algorithm, is fixed in the proposed solution. As potential future work, one can explore the possibility of setting the value of this parameter dynamically and adjust it during program execution.
- Utilizing mutation testing: Employing jump testing to assess the effectiveness of the provided test cases in revealing software errors.
- Strengthening the proposed solution through the combination of data mining algorithms.

Declarations

- **Funding** This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) with the Grant # RGPIN-2021-03298.

- **Conflict of interest** The authors declare no competing interests.
- **Authors' contributions** Maryam Asgari Araghi and Vahid Rafe defined the problem. Maryam Asgari Araghi performed the initial research and discussed with Vahid Rafe, the initial ideas, the solution and algorithms. Maryam Asgari Araghi and Ferhat Khendek worked together to improve the potential of the solution and the algorithms. Maryam Asgari Araghi implemented the work and performed the experiments. All authors discussed, analyzed and refined the experiments. All authors participated actively in the writing of the paper after a first draft written by Maryam Asgari Araghi starting with an outline agreed on by all the authors. All authors participated in the final analysis and in all the rounds of the paper writing.
- **Availability of data and materials** Not applicable.

References

Acharya AA, Mahali P, Mohapatra DP (2015) Model based test case prioritization using association rule mining

Agrawal R, Srikant R (2000) Fast algorithms for mining association rules. Proc 20th Int Conf Very Large Data Bases VLDB 1215

Agrawal R, Imieli'nski T, Swami A (1993) Mining association rules between sets of items in large databases. pp 207–216, https://doi.org/10.1145/170035.170072

Albanese M (2019) Dependency Graphs, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 1–3. https://doi.org/10.1007/978-3-642-27739-9_1771-1, URL https://doi.org/10.1007/978-3-642-27739-9_1771-1

Ammann P, Offutt J (2008) Introduction to Software Testing, 1st edn. Cambridge University Press, USA

Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. pp 1 – 10, https://doi.org/10.1145/1985793.1985795

Baier C, Katoen JP (2008) Principles of Model Checking, vol 26202649

Beizer B (1990) Software Testing Techniques (2nd Ed.). Van Nostrand Reinhold Co., USA

Ehrig H, Engels G, Parisi-Presicce F, et al (2004) Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28-October 1, 2004, Proceedings, vol 3256. Springer

Engels G, Güldali B, Lohmann M (2007) Towards model-driven unit testing. In: Kühne T (ed) Models in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 182–192

Enoiu E, Causevic A, Ostrand T, et al (2014) Automated test generation using model-checking: An industrial evaluation. International Journal on Software Tools for Technology Transfer https://doi.org/10.1007/s10009-014-0355-9

Fraser G, Wotawa F, Ammann P (2009) Issues in using model checkers for test case generation. Journal of Systems and Software 82:1403–1418. https://doi.org/10.1016/j.jss.2009.05.016

Gargantini A, Heitmeyer C (1999) Using model checking to generate tests from requirements specifications. SIGSOFT Softw Eng Notes 24(6):146–162. https://doi.org/10.1145/318774.318939, URL https://doi.org/10.1145/318774.318939

Gönczy L, Heckel R, Varró D (2007) Model-based testing of service infrastructure components. In: Petrenko A, Veanes M, Tretmans J, et al (eds) Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings, Lecture Notes in Computer Science, vol

4581. Springer, pp 155–170, https://doi.org/10.1007/978-3-540-73066-8_11, URL https://doi.org/10.1007/978-3-540-73066-8_11

Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. Sigmod Record 29:1–12. https://doi.org/10.1145/342009.335372

Han J, Kamber M, Pei J (eds) (2012) Preface, third edition edn. The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Boston, https://doi.org/https://doi.org/10.1016/B978-0-12-381479-1.00020-4, URL https://www.sciencedirect.com/science/article/pii/B9780123814791000204

Heckel R (2006) Graph transformation in a nutshell. Electronic Notes in Theoretical Computer Science 148:187–198. https://doi.org/10.1016/j.entcs.2005.12.018

Heckel R, Mariani L (2004) Component integration testing by graph transformations. In: International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications, Cairo

Heckel R, Khan T, Machado R (2011) Towards test coverage criteria for visual contracts. ECEASST 41. https://doi.org/10.14279/tuj.eceasst.41.667

Herman PM (1976) A data flow analysis approach to program testing. Aust Comput J 8:92–96

Ilkhani A, Abaei G (2010) Extraction test cases by using data mining; reducing the cost of testing. pp 620 – 625, https://doi.org/10.1109/CISIM.2010.5643525

Kalaee A, Rafe V (2019) Model-based test suite generation for graph transformation system using model simulation and search-based techniques. Information and Software Technology 108:1–29. https://doi.org/10.1016/j.infsof.2018.12.001

Kastenberg H, Rensink A (2006) Model checking dynamic states in groove. pp 299–305, https://doi.org/10.1007/11691617_19

Khan T, Runge O, Heckel R (2012) Testing against visual contracts: Model-based coverage. pp 279–293, https://doi.org/10.1007/978-3-642-33654-6_19

Lan Q, Zhang D, Wu B (2009) A new algorithm for frequent itemsets mining based on apriori and fp-tree. 2010 Second WRI Global Congress on Intelligent Systems 2:360–364. https://doi.org/10.1109/GCIS.2009.387

Lara Jd, Vangheluwe H (2002) Atom3: A tool for multi-formalism and meta-modelling. In: Kutsche RD, Weber H (eds) Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 174–188

Last M (2005) Data Mining for Software Testing, pp 1239–1248. https://doi.org/10.1007/0-387-25465-X_59

Mohalik S, Gadkari A, Yeolekar A, et al (2014) Automatic test case generation from simulink/stateflow models using model checking. Software Testing, Verification and Reliability 24. https://doi.org/10.1002/stvr.1489

Muthyala K, Naidu R (2011) A novel approach to test suite reduction using data mining. Indian Journal of Computer Science and Engineering 2

Naik K, Tripathy P (2011) Software testing and quality assurance: theory and practice. John Wiley & Sons

Pira E, Rafe V, Nikanjam A (2016) Emcdm: Efficient model checking by data mining for verification of complex software systems specified through architectural styles. Applied Soft Computing 49. https://doi.org/10.1016/j.asoc.2016.06.039

Pira E, Rafe V, Nikanjam A (2018) Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations. Information and Software Technology 97. https://doi.org/10.1016/j.infsof.2018.01.004

Rafe V (2013) Scenario-driven analysis of systems specified through graph transformations. Journal of Visual Languages & Computing 24. https://doi.org/10.1016/j.jvlc.2012.12.002

Rafe V, Mohammady S, Cuevas E (2022) Using bayesian optimization algorithm for model-based integration testing. Soft Comput 26(7):3503–3525. https://doi.org/10.1007/s00500-021-06476-9, URL https://doi.org/10.1007/s00500-021-06476-9

Rapps S, Weyuker E (1985) Selecting software test data using data flow information. Software Engineering, IEEE Transactions on SE-11:367– 375. https://doi.org/10.1109/TSE.1985.232226

Rayadurgam S, Heimdahl M (2001) Coverage based test-case generation using model checkers. pp 83–, https://doi.org/10.1109/ECBS.2001.922409

Rensink A (2004) The groove simulator: A tool for state space generation. In: Pfaltz JL, Nagl M, Böhlen B (eds) Applications of Graph Transformations with Industrial Relevance. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 479–485

Rensink A, Boneva I, Kastenberg H, et al (2010) User manual for the groove tool set. Department of Computer Science, University of Twente, The Netherlands

Runge O, Khan TA, Heckel R (2013) Test case generation using visual contracts. Electronic Communications of the EASST 58

Saifan A, Alsukhni E, Alawneh H, et al (2016) Test case reduction using data mining technique. International Journal of Software Innovation 4:56–70. https://doi.org/10.4018/IJSI.2016100104

Schieferdecker I (2012) Model-based testing. IEEE Software 29:14–18. https://doi.org/10.1109/MS.2012.13

Taentzer G (2003) Agg: A graph transformation environment for modeling and validation of software. pp 446–453, https://doi.org/10.1007/978-3-540-25959-6_35

Thöne S (2005) Dynamic software architectures: a style based modeling and refinement technique with graph transformations

Utting M, Legeard B (2006) Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Utting M, Legeard B, Bouquet F, et al (2016) Recent Advances in Model-Based Testing, pp 53–120. https://doi.org/10.1016/bs.adcom.2015.11.004

Varro D, Balogh A (2007) The model transformation language of the viatra2 framework. Science of Computer Programming 68:214–234. https://doi.org/10.1016/j.scico.2007.05.004

Villani E, Pastl R, Coracini G, et al (2019) Integrating model checking and model based testing for industrial software development. Computers in Industry 104:88–102. https://doi.org/10.1016/j.compind.2018.08.003

Wan C, Wang L, Phoha VV (2018) A survey on gait recognition. ACM Comput Surv 51(5). https://doi.org/10.1145/3230633, URL https://doi.org/10.1145/3230633

Witten I, Frank I (2002) Data mining - practical machine learning tools and techniques with java implementations. Morgan Kaufmann 31

Wu B, Zhang D, Lan Q, et al (2008) An efficient frequent patterns mining algorithm based on apriori algorithm and the fp-tree structure. Convergence Information Technology, International Conference on 1:1099–1102. https://doi.org/10.1109/ICCIT.2008.109