Using Memetic algorithm for Testing of Contract-based Software Models

Anvar Bahrampour, Vahid Rafe

Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 3815688349, Iran
a-bahrampour@phd.araku.ac.ir, v-rafe@araku.ac.ir

Abstract

Graph Transformation System (GTS) can formally specify the behavioral aspects of complex systems through graph-based contracts. Test suite generation under normal conditions from GTS specifications is a task well-suited to evolutionary algorithms such as Genetic and Particle Swarm Optimization (PSO) metaheuristics. However, testing the vulnerabilities of a system under unexpected events such as invalid inputs is essential. Furthermore, the mentioned global search algorithms tend to make big jumps in the system's state-space that are not concentrated on particular test goals. In this paper, we extend the HGAPSO approach into a cost-aware Memetic Algorithm (MA) by making small local changes through a proposed local search operator to optimize coverage score and testing costs. Moreover, we test GTS specifications not only under normal events but also under unexpected situations. So, three coverage-based testing strategies are investigated, including normal testing, robustness testing, and a hybrid strategy. The effectiveness of the proposed test generation algorithm and the testing strategies are evaluated through a type of mutation analysis at the model-level. Our experimental results show that 1) the hybrid testing strategy outperforms normal and robustness testing strategies in terms of fault-detection capability, 2) the robustness testing is the most cost-efficient strategy, and 3) the proposed MA with the hybrid testing strategy outperforms the state-of-the-art global search algorithms.

Keywords: Robustness testing; Model testing; Graph transformation specification; Specification testing; Coverage criteria;

1. Introduction

The complexity of safety-critical systems is now growing, and assuring their functionalities is a challenge in various aspects. Several approaches for testing and formal verification of such systems have already been proposed in the literature [1, 2]. Specification testing concentrates on the behavioral accuracy of the System Under Consideration (SUC), which attempts to reveal defects of the specified components [3]. Although specification validation substantiates that the model meets its intended purposes, the specification may describe only normal conditions and does not define the behavior of the system under unexpected situations such as invalid inputs or inappropriate execution scenarios. A system that works correctly under normal conditions could not necessarily handle unexpected situations [4, 5].

In complex systems, it is not feasible to cover all possible inputs (valid and invalid inputs) and conduct a complete test of the specification [6]. In these systems, the state-space-explosion problem is a common challenge in covering test goals, and there is a need for scalable approaches to handle this problem [7]. However, evolutionary testing is a promising search-based approach to handle the test generation as an optimization problem [8, 9]. Moreover, robustness testing is a well-known approach that evaluates "the degree to which a system or a component can function correctly in the presence of invalid inputs or stressful environmental conditions" [10]. In other words, a robust system does not crash despite exceptional or inappropriate function calls. In formal modeling, if the specification considers all normal and abnormal conditions, robustness testing acts as functional testing [4].

Design by contract is a software development methodology that represents the functionalities through contracts [11, 12]. Graph Transformation System (GTS) is a formalism that could specify the behavioral aspects of software components through well-defined graph transformation rules [13, 14]. In this formalism, the pre- and post-condition of each transformation rule is defined through a visual graph representation as a mathematical tool for model analysis and execution. In GTS, graph elements (nodes/edges) could have a set of attributes and computational/conditional expressions to specify data processing components at different levels of abstraction [15]. The flexibility of the GTS modeling framework causes growing attention to it as a visual specification tool for simulating and reasoning behavior of software systems [16, 17].

In the literature, there are several Model Based Testing (MBT) approaches for GTS specifications [18-21]. Most of the proposed approaches use a type of data-dependency among transformation rules as coverage criteria to guide the test generation process. Although there are types of data-conflicts among transformation rules, none of the existing approaches cover them. With the best of our knowledge, all existing GTS testing approaches try to evaluate the functionality of the system under normal conditions. In [21], the state space exploration capability of a model checker and a set of global search algorithms such as Genetic Algorithm (GA)[22], Particle Swarm Optimization (PSO)[23], and a Hybrid version of GA and PSO i.e. HGAPSO, have been utilized successfully to handle the state space explosion problem in MBT. In this research, the test generation is defined as an optimization problem that aims to maximize the coverage score of data-dependency test objectives. However, the mentioned global search algorithms tend to make large changes in the system state-space that are not concentrated on covering particular test goals and interactions.

In this paper, all possible types of data-centric relationships (data-dependencies/-conflicts) among GTS transformation rules are investigated. A set of data-conflict relationships is proposed as coverage criteria to guide the robustness testing of GTS specifications. In MBT, robustness testing aims to test what is the behavior of the SUC in the presence of the pre-condition violation [24, 25]. In GTS, this means that the necessity of each precondition's component or its completeness is evaluated for all transformation rules. Furthermore, we extend the HGAPSO approach to cover both data-dependency and data-conflict relationships as coverage criteria within three testing strategies i.e. normal testing, robustness testing, and normal with robustness testing as a hybrid strategy. Moreover, we use a memetic algorithm which equips GA global search metaheuristic with a proposed small local search operator to concentrate on particular test objectives. To make more effective integration of local search with global search operators, and cover all experienced test goals over the whole search process a restoring coverage technique is used. The goals of this optimization process are to maximizes coverage score and minimize testing costs.

This testing approach was implemented in the GROOVE (GRaph-based Object-Oriented VErification) toolset. This toolset is already used as a model checker for object-oriented systems specified through GTS formalism [26]. It can generate the whole state-space of the SUC if there is enough memory. To evaluate the efficiency of the test generation approach and the effectiveness of the introduced data-conflict coverage criteria, a series of experiments have been conducted on five well-known case studies [19, 27, 28]. The effectiveness of the generated tests is evaluated in terms of fault detection capability. To this aim, a type of mutation analysis is used at the GTS specification level. The experimental results demonstrate that 1) the proposed hybrid testing strategy outperforms the simple normal and robustness testing strategies in terms of the fault detection capability, 2) the costs of the robustness testing strategy is less than the others in terms of the number of rule applications required per killed mutant, and 3) the proposed MA with the hybrid testing strategy outperforms the state-of-the-art techniques. We summarized the main contributions of this research as follows:

1- A set of coverage criteria based on data-conflicts among transformation rules is proposed for robustness testing of contract-based software models specified through GTS.
2- A local search operator is devised to improve test cases in the sense of covering new test objectives for the first time.
3- A cost-aware MA as an integration of the proposed local search operator and Genetic global search operators is proposed for test-suite generation based on robustness, normal, and hybrid testing strategies from GTS specifications.
4- The effectiveness of data-dependency and data-conflict coverage criteria are investigated through a type of mutation analysis in terms of fault detection capability at the specification level.
5- The performance of the proposed strategies is evaluated in terms of coverage score, fault detection capability, and cost-effectiveness using well-known case studies, and it is compared with the state-of-the-art.

The rest of this paper is organized as follows. Section 2 represents the basic concepts of modeling with GTS formalism and presents a brief review of the HGAPSO approach for testing GTS specifications. Section 3 surveys state-of-the-art. Section 4 investigates all possible data-relationships among transformation rules and describes the proposed data-conflicts between rules as coverage criteria for robustness testing. Then, the search-based test generation algorithm is described in detail. The evaluations of the test generation approach at the model level and the experimental results are presented in Section 5. Section 6 concludes the paper and suggests some future works.

2. Backgrounds

In this Section, we describe some preliminaries such as the basic concepts of GTS formalism and a brief review of the HGAPSO approach for testing using GTS specifications.

2.1. Graph transformation system

Graph is a powerful mathematical tool for modeling complex systems. GTS is a graph-based formalism that is capable of simulating systems in both structural and behavioral aspects. The main features of the GTS formalism are introduced in [14, 16, 21, 29, 30]. In GTS, the behavior of the SUC is specified through production rules, while the configuration of the system is represented by a state graph. The initial state of the SUC is described by a host graph. Graph elements (nodes/edges) in state graphs or production rules may have data-attributes of various data types and store any possible value. A sequence of GTS rule transitions is mapped to a sequence of method calls in the corresponding implementation of the SUC. The following definitions represent the required background of the GTS formally.

Definition 1 (Graph, Graph Morphism). G = (N, E, *src*, *trg*) is a graph where N and E are finite sets of nodes and edges, respectively. *src*: E → N and *trg*: E → N are functions that define the source and target of an edge, respectively. Graph morphism *f*: G → H is defined as a mapping of the graph G to the graph H where $f = (f_N, f_E)$, $f_N: N_G \rightarrow N_H$, and $f_E: E_G \rightarrow E_H$ such that $f_N \circ src_G = src_H \circ f_E$ and $f_N \circ trg_G = trg_H \circ f_E$.

Definition 2 (Production Rule). A production rule is defined as $P: L \overset{N}{\rightarrow} R$, where L is the Left-Hand Side (LHS), R is the Right-Hand Side (RHS), and N is a Negative Application Condition (NAC). L and N define the pre-condition of the production rule, and R describes its post-condition.

The LHS, RHS, and NAC are attributed-graphs. A rule application includes finding a match for the LHS in the current state graph by graph morphism and replace with the RHS when there are no occurrences of NAC elements. In other words, all graph elements matched by LHS\RHS are deleted, and an image of RHS\LHS (referred to as Creators) is added to the instance graph. In the context of the GTS, we refer to LHS\RHS ∩ RHS\LHS as Updaters, and (LHS\RHS)\Updaters as Erasers. The elements that exist in both LHS and RHS without any difference are called *Readers*, while *Creators* are elements that exist only in RHS. Several matches (morphisms) of a rule are differentiated through their parameters, which are defined in rule signature P(x) where P is the name of the production rule, and x is a set of input/output parameters. In this research, production rules are referred to as software contracts that describe the behavior of the SUC.

Definition 3 (Graph Transformation System GTS). GTS is a triple (TG, HG, R) where TG=(TN,TE, src, trg) is an attributed type graph, in which, TN is a set of Node types and TE is a set of Edge types and src,trg: E → N are functions that define the source and target of an edge type, respectively, HG is an instance of TG called host graph, and R is a finite set of production rules.

The system configuration transforms from the current state to the next one by a transformation step. A transformation step is defined as a rule application *p* with the match *m* in the GTS. It is represented by $G \overset{p.m}{\Rightarrow} H$. The state-space of the SUC can be generated through the applications of various GTS rules repeatedly. In GTS, the state-space of the SUC is represented by a transition system. A transition system is a directed graph where the nodes represent the states, and the edges represent the transitions. Each path of the state-space could be utilized as a test case (sequence of method invocations).

Definition 4 (Transition System TS). A Transition System TS = (S, Act, →, I) where:

1) S is a set of states.
2) Act is a set of actions.
3) → is a transition relation that is a subset of S×Act×S.
4) I is the set of initial states (the subset of S).

In the rest of the paper, we use a Hotel Management System (HMS) represented in [21] with the same functionality but a bit different GTS specification (to make well descriptive) to explain our contributions. In the HMS, a hotel initially has several *Rooms* and registered *Guests*. Every guest can book any vacant room, and his/her bill will be maintained automatically. Before the guest leaving the room, the bill should be paid, and the

guest could check out. Figure 1 shows a simple state graph (initial state) of the HMS in the GROOVE toolset. In this state graph, each node has several attributes that define the states of the corresponding objects. The production rules of the HMS are illustrated in Figure 2, including *BookRoom*, *CheckOut, ClearBill, UpdateBill, and OccupyRoom*. In the GROOVE toolset, *Readers* are shown by a solid black line, E*rasers* are blue dashed or double-bordered lines elements, *Creators are* represented by green solid lines, and the NAC elements are indicated by red double-bordered/dashed lines.

Figure 2.f represents a simple path of the state-space of the HMS in which a sequence of production rules has been applied to the initial state represented in Figure 1. In this scenario, at first, room "1" is booked by "Daniel Castro", then it is occupied, and a bill "1023" is created. After the bill is updated and cleared by the corresponding rule applications, the guest was successfully checked out.

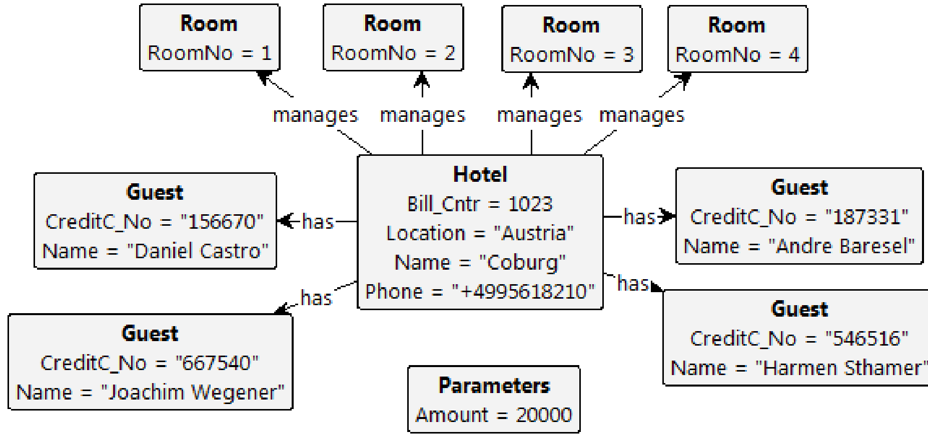


Figure 1. A simple state graph for HMS in the GROOVE toolset

2.2. Test generation from GTS specification (The HGAPSO approach)

In HGAPSO [21], to cope with the state-space explosion problem in test-suite generation for complex systems using model checkers, the test generation task is defined as an optimization problem, and a hybrid search-based approach is proposed. This approach uses a type of data-dependency as coverage criteria to guide the search process. In this section, we provide a brief review of the HGAPSO approach.

2.2.1. Problem representation

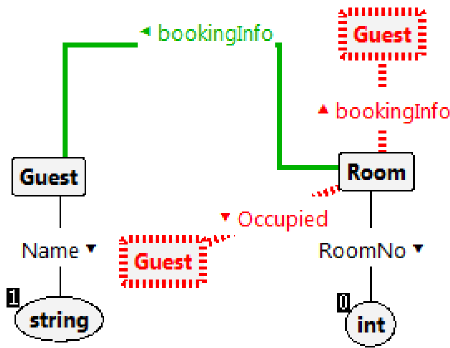
In the HGAPSO, a chromosome of the search process is a test suite, which is defined as a set of test cases. A test case t in GTS specification is defined as a tuple t = (P, O, S_0), where P is a sequence of transformation steps <$P_1(x_1)$, $P_2(x_2)$, …, $P_n(x_n)$>. $P_i(x_i)$ denotes a rule signature in GTS specification, O is a test path <S_0, S_1, S_2, …, S_n> in the state-space, such that S_{i-1} is the source state of the rule transition P_i and S_i represents the target state of the transformation step ($1 \leq i \leq n$). S_0 is a start state or host graph. Since the state graph S_i includes the post-condition of the transformation step $P_i(x_i)$, O = <S_1, S_2, …, S_n> is the test oracle at the model level for test sequence P. For example in Figure 2.f, P=<BookRoom(1,"Daniel Castro"), OccupyRoom(1,"Daniel Castro",1023), UpdateBill(1023,20000), ClearBill(1023), Checkout(1,"Daniel Castro",1023) > is a test sequence, and the test oracle is O=<S_1, S_2, S_3, S_4, S_3, S_4, S_5>. The length of the test case t is defined as the number of its transformation steps. Hence, the length of the test suite T is defined as the sum of the lengths of its test cases i.e. $length(T) = \sum_{t \epsilon T} length(\mathrm{t})$.

A chromosome is created initially through a random walk into the state-space of the SUC. Figure 3.a shows a test suite, which consists of two test cases. Test Case 1 has a length of 5, and this is 7 for Test Case 2. Hence the length of the test suite is 12. Figure 3.b shows the encoded representation of the mentioned chromosome. Each value in the encoded test case shows the number of an outgoing transition of the source state.

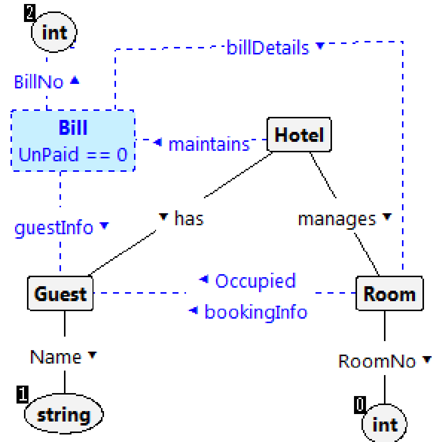
2.2.2. Coverage criteria

In GTS, rules interact with each other through a type of data sharing. For example, according to Figure 2.a, the application of the rule *BookRoom* assigns a guest to a vacant room by creating an edge labeled "bookingInfo" from the room to the guest, which could be read later by the production rule *OccupyRoom.* In other words, the application of the rule *OccupyRoom* depends on the successful use of *BookRoom* to the system state. For another

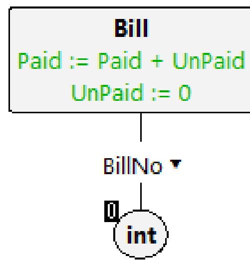
example, the application of *CheckOut* depends on successfully applied the *OccupyRoom* to provide needed objects such as a node *Bill* and an edge *Occupied*. This type of rule-dependency, where one rule creates/updates an entity (i.e. creates node/edge or update an attribute of a node), and another rule uses (i.e. read/delete) it, is referred to as data-dependency in GTS. This notion is defined formally in Definition 5.



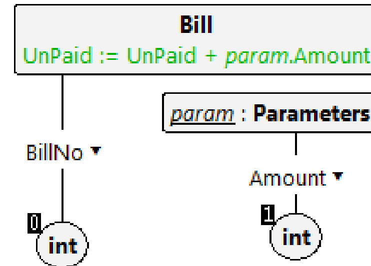
(a) BookRoom(int RoomNo, String Name)



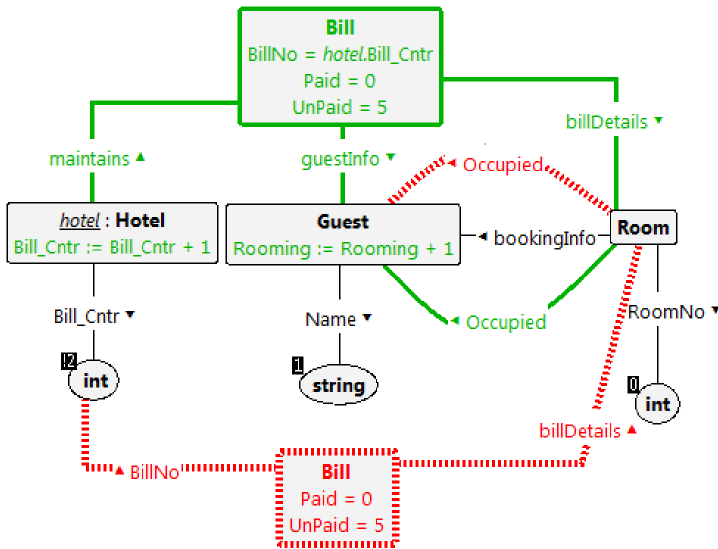
(b) CheckOut(int RoomNo, String Name, int BillNo)



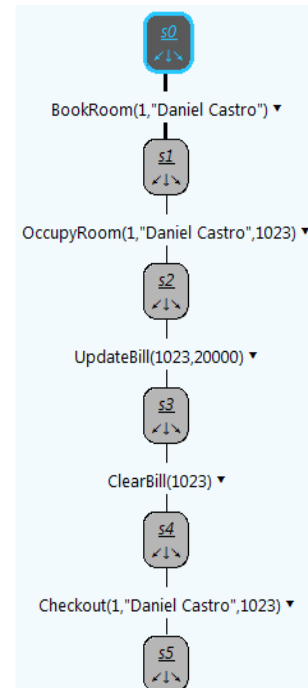
(c) *ClearBill(int BillNo)*



(d) UpdateBill(int BillNo, int Amount)



(e) OccupyRoom(int RoomNo, String Name, int Bill_Cntr)



(f) Simple execution path

Figure 2. Production rules of the HMS example in the GROOVE toolset

Test Suite

Test Case 1	***BookRoom (1, "Daniel Castro"),*** ***OccupyRoom (1, "Daniel Castro", 1023),*** ***UpdateBill (1023, 20000),*** ***ClearBill (1023),*** ***Checkout(1, "Daniel Castro", 1023)***
Test Case 2	***BookRoom (1, "Daniel Castro"),*** ***BookRoom (4, "Andre Baresel")*** ***OccupyRoom (4, "Andre Baresel", 1023),*** ***BookRoom (3, "Daniel Castro"),*** ***UpdateBill (1023, 20000),*** ***ClearBill (1023),*** ***Checkout(4, "Andre Baresel", 1023)***

(a) A test suite, which includes two test cases

{[1,6,5,3,1], [1,3,7,4,4,1,4]}

(b) Chromosome representation

Figure 3. Chromosome encoding.

Table 1. Data dependency relationships extracted from the HMS.

Relation type	Dependent rules	Entity of the relation
Create _ Read (cr)	*(BookRoom, OccupyRoom)*	*bookingInfo*
	(OccupyRoom, UpdateBill), (OccupyRoom, ClearBill)	*Bill, BillNo*
Create _ Delete (cd)	*(BookRoom, Checkout)*	*bookingInfo*
	(OccupyRoom, Checkout)	*Bill, billDetails, Occupied, guestInfo, maintains, BillNo*
	(ClearBill, Checkout)	*Paid, UnPaid*
Create _ Update (cu)	*(UpdateBill, UpdateBill), (UpdateBill, ClearBill) (OccupyRoom, UpdateBill)*	*UnPaid*
	(OccupyRoom, OccupyRoom)	*Bill_Cntr, Rooming*
	(OccupyRoom, ClearBill), (ClearBill, ClearBill)	*Paid, UnPaid*
	(ClearBill, UpdateBill)	*UnPaid*

Definition 5 (*Rule-Dependency*). Given two production rules p_1 and p_2, we say p_2 is dependent on p_1 if there are transformation steps $t_i = (G \overset{p_i.m_i}{\Rightarrow} G_i)$ and $t_j = (G_i \overset{p_j.m_j}{\Rightarrow} G_j)$ such that t_i enables t_j.

In [31], various types of data-dependencies are introduced as coverage criteria for testing visual contracts. A data-dependency relationship is referred to as enabler relation which means that the first rule of the relationship enables the second one. The relationships *Create_Read*, *Create_Delete*, *Create_Update*, *Update_Read,* and *Update_Update* have been introduced as asymmetric rule dependencies. The rule R_j can read from R_i, if $Readers_j \cap Creators_i$ is not empty, and there may be a *Create-Read* relation between rules R_j and R_i through the element type $e \in Readers_j \cap Creators_i$. If $Erasers_j \cap Creators_i$ is not empty, the rule R_j can delete an element e from

R_i, and there may exist a *Create-Delete* relation between the two rules. *Create-Update* is another relationship that would happen between two transformation rules, where the nonempty set $Updaters_j \cap Creators_i$ suggests that R_j may update an element, which created by R_i.

In the execution of Figure 2.f, the application of *BookRoom(1, "Daniel Castro")* leads to the creation of an edge labeled with "bookingInfo" among the room "Room 1" and the guest "Daniel Castro" of the host graph. This edge is needed for applying the next step, say *OccupyRoom(1, "Daniel Castro", 1023). This step* prevents reapplying *BookRoom(1, "Daniel Castro")* through the NAC elements of the rule *BookRoom.* The node *Bill* and its related edges are created as well as an edge "occupied" among the room "Room 1" and the guest element "Daniel Castro" by applying *OccupyRoom(1, "Daniel Castro", 1023)*. The created elements in this stage are used in the subsequent steps of the execution path. However, it is apparent that there are types of data-dependencies (enabling subsequent steps) and data-conflicts (disabling some rule applications) among GTS production rules. In the case of data-dependencies, a rule creates/updates an entity in which another rule uses it, and in the case of data-conflict, a rule creates/deletes an entity which forbidden/used in another rule.

The HGAPSO approach uses these rule-dependency relationships, in which the first step defines an object (or objects) and the second one uses it (them) in its pre-condition, as an estimation of *def-use* relationship in code-based testing [19, 21]. In this approach, *Update_Read,* and *Update_Update* relationships are considered as *Create_Read* and *Create_Update* respectively. The HGAPSO uses these rule-dependencies as coverage criteria to guide the search process in the state-space of the SUC. Table 1 shows all the data-dependencies of the above types for the HMS. The HGAPSO search algorithm is aimed to achieve a high rule-dependency coverage score concerning the selected coverage criterion. Relation 1 states how fitness is measured for rule-dependency coverage criteria.

1) $F_{Dep} = |\bigcup_{T \in Ch} Data_Ralations(T)|$ *where* $Data_Relation \in \{$ *Create_Read*, *Create_Delete*, *Create_Update* $\}$

In GTS, all operations that could be applied to the elements of the production rules are {*create, read, update, delete, forbid*}. In other words, in a production rule, an element (node, edge, and attribute) could be created, updated, read, or deleted by rule application or forbidden to applying the rule. Hence, according to these operations, the set of all possible data dependency/conflict types among production rules is defined as a subset of $\{c, r, u, d, f\} \times \{c, r, u, d, f\}$.

Some types of rule dependencies and conflicts have been proposed as coverage criteria for MBT using visual contracts in [31], but as we will discuss in the next section, it is not a complete relation set for GTS. In the literature, rule-dependency relationships are used as an estimation of *def-use* in code-based testing. Based on this notion, data-centric rule-dependency relationships proposed in [31] have been used successfully for MBT in [18, 19, 21, 31], but with the best of our knowledge, there is no research work generates test suite based on data-conflict relationships.

3. Related Works

MBT is one of the increasingly popular testing approaches which uses the formal or semi-formal specification of the SUT. Model-based robustness testing utilizes pre-condition violation of the functional specification of the software components under test [24, 25]. There are several approaches for validation and verification of formal specifications, including formal verification, specification testing, specification simulation, and specification animation [32]. Formal specification testing techniques utilize mathematical representation of the component's functionality and theoretical analysis to generate effective test cases [33]. In [34], several specification testing approaches have been addressed using a finite state machine (FSM) and labeled transition system formalisms.

A type of mutation testing for extended time automata specifications has been proposed in [35], which uses symbolic execution over the model. In [36], an approach for functional testing of B specifications is proposed. This approach uses operation coverage in which a finite coverage graph is generated so that each operation tested at least once. A robustness specification testing approach was proposed in [4] that uses model-based and mutation testing. In this research, the pre-conditions of the events are mutated to test invalid ones. Louzaoui and Benlhachmi [37] proposed a robustness testing approach for object-oriented models based on invalid input data that violate precondition of the function under test.

In the literature, there are several search-based software testing (SBST) techniques [38, 39]. Fraser and Arcuri [40] proposed a whole test suite generation approach that uses a genetic algorithm for the optimization of both branch coverage and length of the resultant test. Nardo et al. [7] proposed a search-based robustness testing approach using data-model and mutation data to generate a robustness test suite at the system-level. In this

research, a multi-objective evolutionary algorithm has been used with both model-based and code-based coverage criteria. Fraser et al. [41] proposed an approach based on the integration of global and local search algorithms to allow the individuals of a population in the global search algorithm for local improvement through a local search. In this research, a set of local search operators are devised to efficiently optimize primitive values, such as integers and doubles, characters that appear in strings, and arrays of primitive values in code-based testing.

Several MBT approaches have been proposed for systems specified with GTS at various levels and architectures [42]. Heckel and Mariani [43] proposed an approach for model-based integration testing of GTS specifications. Heckel et al. [31] introduced a set of data-dependencies and data-conflicts among production rules of the GTS specification as coverage criteria for testing purposes. Dynamic evaluation of the proposed data-dependency coverage criteria in AGG toolset has been introduced by Khan et al. [18]. Runge et al. [19] introduced a model-based test generation approach based on static analysis of rule-dependencies of GTS specifications. Kalaee et. al. [21] proposed a search-based test generation approach (HGAPSO) based on static analysis of data-dependencies among GTS rules. With the best of our knowledge, the robustness testing of GTS specification is not researched in the literature.

4. The proposed Testing Approach

As mentioned in the previous section, GTS formalism has been used in the literature to specify software behaviors by production rules as visual contracts. Furthermore, several data-dependencies among production rules of the GTS specifications are used as coverage criteria to guide the test generation process [18, 19, 21, 31]. In this section, we will propose an approach to generate robustness tests, that uses three different types of data-conflict among production rules to guide the search algorithm as well as a hybrid testing strategy that uses both data-dependencies and data-conflicts as coverage criteria. Furthermore, to optimize the test suite concerning both coverage level and test size, we will propose amemetic algorithmto generate a whole test suite from GTS specification.

4.1. Robustness testing

The purpose of test cases in normal testing approaches is to evaluate whether the target state of each transformation step is achieved when its precondition satisfied. As aforementioned, the other aspect of testing is to test what is the behavior of the SUC in the presence of the precondition violation. Indeed, in this type of model testing, the necessity of each precondition's component (or precondition completeness) is investigated. With this aim, we define the robustness testing of GTS specifications as the following definition.

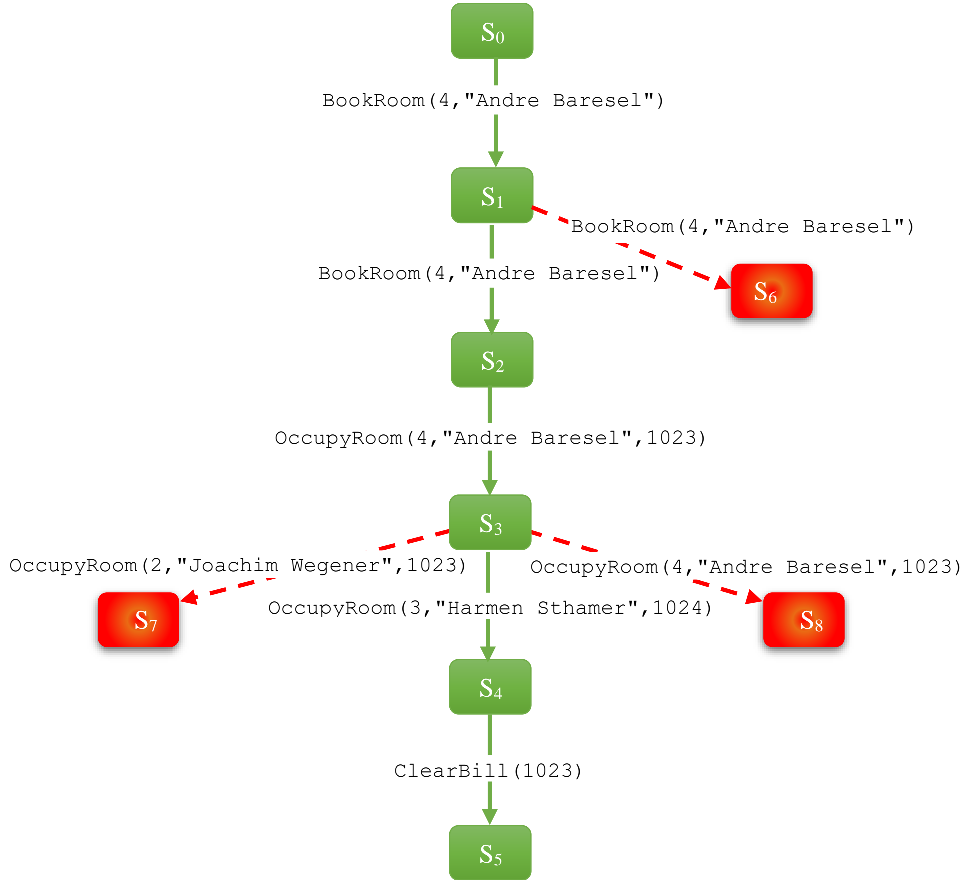
Definition 6 (*Robustness Test Case*). A robustness test case RTC= (P, IS S_0, E) where:

1) P is a normal sequence of transformation steps $<P_0(x_0), P_1(x_1), \ldots, P_n(x_n)>$ where $P_i(x_i)$ represents the signature of the applied rule P_i with parameter set x_i ($0 \leq i \leq n$);
2) IS is a sequence of sets of invalid transformation steps $<IS_0, IS_1, \ldots, IS_n>$ such that $IS_i = < P_{i0}(x_{i0}), P_{i1}(x_{i1}), \ldots, P_{im}(x_{im})>$ where $P_{ij}(x_{ij})$ represents the signature of an invalid transformation step (the list of the actual parameters that violates the precondition of the transformation step) in the *i*th step of the sequence P.
3) $S_0 \in I$ is a start state;
4) $E = <S_1, S_2, \ldots, S_n>$ is an execution path of the corresponding TS in which S_i ($1 \leq i \leq n$) is a target state of the transformation step $P_{i-1}(x_{i-1})$.

This definition does not state the test oracles for invalid steps. It is trivial because these steps should be failed. In other words, the successful application of any steps of the IS_i means that there is a fault in the SUC. Figure 4.a represents a robustness test case for the HMS. This test case consists of a normal path (green steps) that should be applied successfully on the start graph, and a set (possibly empty set) of invalid transformation steps (red ones) in each step of the normal path. Figure 4.b illustrates the corresponding test script along with P and IS sequences. In each step of the normal sequence, although there are probably infinite invalid transformation steps to be included in the corresponding IS set, most of them could not reveal any fault of the SUC. Moreover, most of them detect the same set of fault types. Therefore, in the following section, we will introduce a set of data-centric coverage criteria to effectively guide the robustness test generation process.

4.2. The proposed data-centric coverage criteria

As mentioned in section 2.2, all possible data-centric relation types among transformation steps of the test sequence are included in {*c, r, u, d, f*}×{ *c, r, u, d, f* }. According to this Cartesian multiplication, Table 2 shows all possible combinations of GTS operations. Some of these combinations are meaningless and do not imply any logical dependency/conflict relationship among their transformation steps. For example, the relationships {*rc, rr, ru, rd, rf*} in which a data item was read by the former step, while the same data item was created, read, updated, deleted, or forbidden by the later transformation step, imply no meaningful dependency/conflict relationship. The relationships {*cr, cu, cd, cf*}, in which the source transformation step creates an object (data item) while the target one uses it (read, update, delete, or forbid the created object), relate valid data-relationships.



(a) A simple robustness test case

```
BookRoom(4,"Andre Baresel") {}
BookRoom(1,"Harmen Sthamer") {BookRoom(4,"Andre Baresel")}
OccupyRoom(4,"Andre Baresel",1023) {}
OccupyRoom(3,"Harmen Sthamer",1024){OccupyRoom(2,"Joachim Wegener",1023),
                                    OccupyRoom(4,"Andre Baresel",1023)}
ClearBill(1023) {}

P=[BookRoom(4,"Andre Baresel"),BookRoom(1,"Harmen Sthamer"),
   OccupyRoom(4,"Andre Baresel",1023),OccupyRoom(3,"Harmen Sthamer",1024),
   ClearBill(1023)]

IS=[{}, {BookRoom(4,"Andre Baresel")}, {}, {OccupyRoom(2,"Joachim
                  Wegener",1023), OccupyRoom(4,"Andre Baresel",1023)}]
```

(b) A simple robustness test script, P, and IS sequences

Figure 4. A sample of robustness test case, and its test script.

The highlighted combinations in Table 2 are valid dependency/conflict relationships. Therefore, all potential dependencies are {*cr, cd, cu, ur, ud, uu, df*}. Some other possible relationships such as *cf, uf, uu, dr, du,* and *dd* are categorized as rule-conflicts because the first transformation step in each of these relationships disables the second one. Other relationships in the above Cartesian multiplication, *e.g. cc, uc, fc, and ff,* imply no dependency or conflict between the corresponding transformation steps.

It is worth noting that in GTS, an update operation is realized by reading its value, deleting it, and then creating a new one with probably a new value. Therefore, in the rest of the paper, we use *cr* for both *cr* and *ur*, *cd* for both *cd* and *ud*, and *cu* for both *cu* and *uu*. In other words, since an attribute is updated by a transformation step, it has behaved as a created new one. Furthermore, in data-conflict combinations, *dr* stands for both *dr* and *du*. Moreover, the data-conflict relationship *uu* could be recognized as a *dd* along with a *cc*. With these considerations, all possible data dependency/conflict relations, also known as enabler/disabler relations, are:

2) *Dependenc*ies = {*cr, cd, cu, df*}.
3) *conflicts* = {*dr, dd, cf*}.

We formally define data-dependency and data-conflict relationships as the following definitions:

Table 2. all possible combinations relationships among production rules {*c, r, u, d, f}x{ c, r, u, d, f }.*

Combination	Description	*Data-Dependency*	*Data-Conflict*
Cc	*Create_ Create*	-	-
Cr	***Create_ Read***	√	-
Cu	***Create_ Update***	√	-
Cd	***Create_ Delete***	√	-
Cf	***Create_ Forbid***	-	√
Rc	*Read_ Create*	-	-
Rr	*Read_ Read*	-	-
Ru	*Read_ Update*	-	-
Rd	*Read_ Delete*	-	-
Rf	*Read_ Forbid*	-	-
Uc	*Update _Create*	-	-
Ur	***Update _Read***	√	-
Uu	***Update _ Update***	√	-
Ud	***Update _Delete***	√	-
Uf	***Update _ Forbid***	-	√
Dc	*Delete_ Create*	-	-
Dr	***Delete_ Read***	-	√
Du	***Delete_ Update***	-	√
Dd	***Delete_ Delete***	-	√
Df	***Delete_ Forbid***	√	
Fc	*Forbid _Create*	-	
Fr	*Forbid_ Read*	-	
Fu	*Forbid_ Update*	-	
Fd	*Forbid_ Delete*	-	
Ff	*Forbid_ Forbid*	-	

Definition 7: (*Data-dependency).* Given two production rules p_i and p_j, $i<j$, we say that p_j is dependent on p_i if there is a sequence of transformation steps t=($G_0 \overset{p_1.m_1}{\Rightarrow} G_1 \dots \overset{p_i.m_i}{\Rightarrow} G_i \dots \overset{p_{j-1}.m_{j-1}}{\Rightarrow} G_{j-1} \overset{p_j.m_j}{\Rightarrow} G_j$) from the start state such that:

- There exists an element (node, edge, or an attribute) e in m_j *on* G_{j-1} created/updated by t_i=($G_{i-1} \overset{p_i.m_i}{\Rightarrow} G_i$) in sequence *t*.

or

- There exists an element e in NAC_j where an exact image of it is deleted by t_i=($G_{i-1} \overset{p_i.m_i}{\Rightarrow} G_i$).

The later defines *df* dependency, while the former comprises all other dependency types.

Definition 8: (*Data-conflict)*. Given two production rules p_i and p_j, $i<j$, we say that p_i disables p_j if there is a sequence of transformation steps $t=(G_0 \overset{p_1.m_1}{\Rightarrow} G_1 \ldots \overset{p_i.m_i}{\Rightarrow} G_i \ldots \overset{p_{j-1}.m_{j-1}}{\Rightarrow} G_{j-1} \overset{p_j.m_j}{\Rightarrow} G_j)$ from the start state such that:

- There exists an element (node, edge, or an attribute) e on G_{j-1} created/updated by $t_i=(G_{i-1} \overset{p_i.m_i}{\Rightarrow} G_i)$ in sequence t, which is forbidden in m_j and prevents applying t_j.

or

- There exist an element (node, edge, or an attribute) e in m_j on G_{j-1} deleted by $t_i=(G_{i-1} \overset{p_i.m_i}{\Rightarrow} G_i)$ in sequence t, which is deleted/read in t_j and prevents applying t_j.

The former defines *cf* conflicts while the later defines *dr/dd* conflicts.

Table 3. Data dependency relations in HMS extracted by definitions 9 and 10.

Relation type	Dependent rules	Entity of the relation
Create _ Read (cr)	*(BookRoom, OccupyRoom)*	*bookingInfo*
	(OccupyRoom, UpdateBill), (OccupyRoom, ClearBill)	*Bill, BillNo*
Create _ Delete (cd)	*(BookRoom, Checkout)*	*bookingInfo*
	(OccupyRoom, Checkout)	*Bill, billDetails, Occupied, guestInfo, maintains, BillNo*
	(ClearBill, Checkout)	*Paid, UnPaid*
Create _ Update (cu)	*(UpdateBill, UpdateBill), (UpdateBill, ClearBill) (OccupyRoom, UpdateBill)*	*UnPaid*
	(OccupyRoom, OccupyRoom)	*Bill_Cntr, Rooming*
	(OccupyRoom, ClearBill), (ClearBill, ClearBill)	*Paid, UnPaid*
	(ClearBill, UpdateBill)	*UnPaid*
Delete_forbidden (dn)	*(Checkout, BookRoom)*	*bookingInfo, Occupied*
	(Checkout,OccupyRoom)	*BillNo, billDetails, Paid, UnPaid, Occupied*
Delete_Read (dr)	*(Checkout, UpdateBill), (Checkout,ClearBill)*	*Bill, BillNo*
Delete_Delete (dd)	*(Checkout, Checkout)*	*Bill, BillNo, Paid, billDetails, UnPaid, bookingInfo, Occupied, guestInfo, maintains*
	(Checkout, UpdateBill), (Checkout, Checkout)	*UnPaid*
	(Checkout, ClearBill)	*Paid, UnPaid*
	(OccupyRoom, OccupyRoom)	*Bill_Cntr, Rooming*
Create_forbidden (cf)	*(BookRoom, BookRoom)*	*bookingInfo*

As related earlier, in GTS specifications, the functionalities of the SUC (methods of an object or a service interface) are modeled as GTS production rules. In this paper, the interoperability of production rules in feasible scenarios (or infeasible in robustness testing) is exercised as well as in integration testing of the software components. In the literature, the rule-dependency relationships {*cr, cu, cd*} are used as *def-use* relationship in MBT [18, 19, 21, 31]. In this research, in addition to dependency relationships, we use data conflict relationships to guide the robustness test generation process.

Table 3 listed all feasible data-dependencies and data-conflicts of the HMS according to dependency/conflict types of relation 1/2. For example, the first row states that *OccupyRoom* reads an edge *bookingInfo* from *BookRoom*. However, as described in Definition 9, we use data-dependency/-conflict relations as coverage criteria for normal/robustness testing of GTS specifications.

Definition 9: (*Data-flow/-conflict coverage).* a test suite for data-flow/-conflict coverage criteria is a set of test cases where each of the feasible data dependency/conflict relations among transformation rules of the model including *{cr, cd, cu, df}/{dr, dd, cf}* is covered by at least one test case.

4.3. Extended fitness function

In practice, all types of the mentioned data-dependencies and data-conflicts can be used separately as a coverage criterion to guide the test generation process. In this research, the data-dependencies are used in normal testing, while robustness strategy uses the data-conflicts as coverage criteria. Relation 4 and Relation 5 calculate the coverage score of a test suite in normal testing and robustness testing, respectively. As Relation 6 shows, the union of all experienced data-dependencies and data-conflicts is used to calculate the coverage score in the hybrid testing strategy. The proposed MA tries to optimize both the coverage level and the test size. The coverage level is the primary goal in each testing strategy, and the length of the chromosome is the secondary objective of the search algorithm. Relation 7 calculates the length of a chromosome as a summation of the size of normal test paths. In these relations, *Ch* stands for chromosome, and *T* indicates a test case.

4) $F_{NormalTesting} = |\bigcup_{T \in Ch} Data_Dependency(T)|\ where\ Data_Dependency \in \{cr.cu.cd.\ df\}$
5) $F_{RobustnessTesting} = |\bigcup_{T \in Ch} Data_Conflicts(T)|\ where\ \ Data_Conflicts \in \{dr.dd.cf\}$
6) $F_{HybridTesting} = F_{NormalTesting} + F_{NormalTesting}$
7) $Length(Ch) = \sum_{T\ \epsilon\ Ch}(L_T)$

Algorithm 1. Calculate the *Fitness, Exercised Data Dependencies* and *Conflicts* within a test suite

Input: *TestSuite*
Output: *Fitness, Exercised Data Dependencies {cr, cd, cu, df}*
Exercised Data Conflicts {dr, dd, cf}

```
1:   cr, cd, cu, df, dr, dd, cf ← {}
2:   ForEach TestCase tc ϵ TestSuite
3:            cr, cd, cu, df ← compute_dataDependencies(tc) // alg 2.
4:            dr, dd, cf ← compute_dataConflicts(tc) // alg 4.
5:            cr ← cr ∪ tc.cr
6:            cd← cd ∪ tc.cd
7:            cu← cu ∪ tc.cu
8:            df← df ∪ tc.df

9:            dr ←dr ∪ tc.dr
10:           dd ← dd ∪ tc.dd
11:           cf ← cf ∪ tc.cf

12:  Fitness=| cr |+| cd |+| cu |+| df |+| dr |+| dd |+| cf |
```

Algorithm 1 shows how the fitness of a test suite is calculated. The data-dependencies of each test case are extracted by Algorithm 2. In this algorithm, to keep track of the data-flow in the normal execution path of the test case, we will record the creator rule of created and updated elements of each transformation step as a feature of an element. The union of detected data-dependencies from test cases of a test suite forms the set of the experienced data-flow of the test suite.

Algorithm 2 shows how each read/created/updated element of a transformation step bear its creator rule. So, it is very simple to detect *cr, cd, and cu* relations. But, the detection of *df* relations is a bit complex. As shown in Algorithm 3, we should check for each transition *t,* how the elements deleted in the previous transitions enable transition *t* through the provision of its forbidden elements.

Algorithm 2. Computation of the *Exercised Data Dependencies* of a test case

Input: *TestCase*
Output: *Exercised Data Dependencies cr, cd, cu, and df*

```
1:  cr, cd, cu, df ← {}
2:  CurrentPos=0;
3:  CurrentState=StartState;
4:  While CurrentPos < |normalTestPath| do
5:       Transition  t  ← Apply(normalTestPath [CurrentPos]);
6:       Rule  R1  ← t.AppliedRule
7:       ForEach Element e ϵ t.Preserved
8:            IF e.CreatorRule <> null
9:                 Rule  R  ← e.CreatorRule
10:                cr ← cr ∪ {(R, R1, e)}

11:      ForEach Element e ϵ t.Deleted
12:           IF e.CreatorRule <> null
13:                Rule R  ← e.CreatorRule
14:                cd← cd ∪ {(R, R1, e)}

15:      ForEach Element e ϵ t.Updated
16:           IF e.CreatorRule <> null
17:                Rule R  ← e.CreatorRule
18:                cu ← cu ∪ {(R, R1, e)}
19:
20:      IF  R1.NACs<>null
21:           df ← df  ∪ Compute_DeleteNAC(TestCase, t) // Alg. 6.

22:      ForEach Element e ϵ t.Created
23:           e.CreatorRule  ← R1
24:      CurrentState  ←t.TargetState
```

Algorithm 3. Computation of the exercised data-dependency *Delete_Forbid* of a test case

Input: *TestCase , current transition t*
Output: *Exercised Data Dependencies df*

```
1.   df ← {}
2.   ForEach previous Transition tp ϵ TestCase
3.        deleted ← tp.Deleted elements which are images of t.NACs
4.        IF deleted<>null
5.             StateGraph ←   t.sourceStateGraph ∪ deleted
6.             IF reDo t from StateGraph faild
7.                  ForEach Element e ϵ deleted
8.                       Rule R1  ← tp.AppliedRule
9.                       Rule R2  ← t.AppliedRule
10.                      df ← df  ∪ {(R1, R2, e)}
```

As aforementioned, the robustness components of a test case will be generated by finding a proper set of invalid transformation steps in each step of the normal test path. It is trivial that there are infinite invalid transitions in each state. Hence, we use the data-conflict relations among production rules to select a small subset

of invalid transition for robustness testing. The purpose of the data-conflict component of the fitness function is to exercise each data-conflict relation among production rules at least once to promote the effectiveness of the output test suite. On the other hand, to optimize the cost of robustness testing, it should be avoided to experience data-conflicts redundantly.

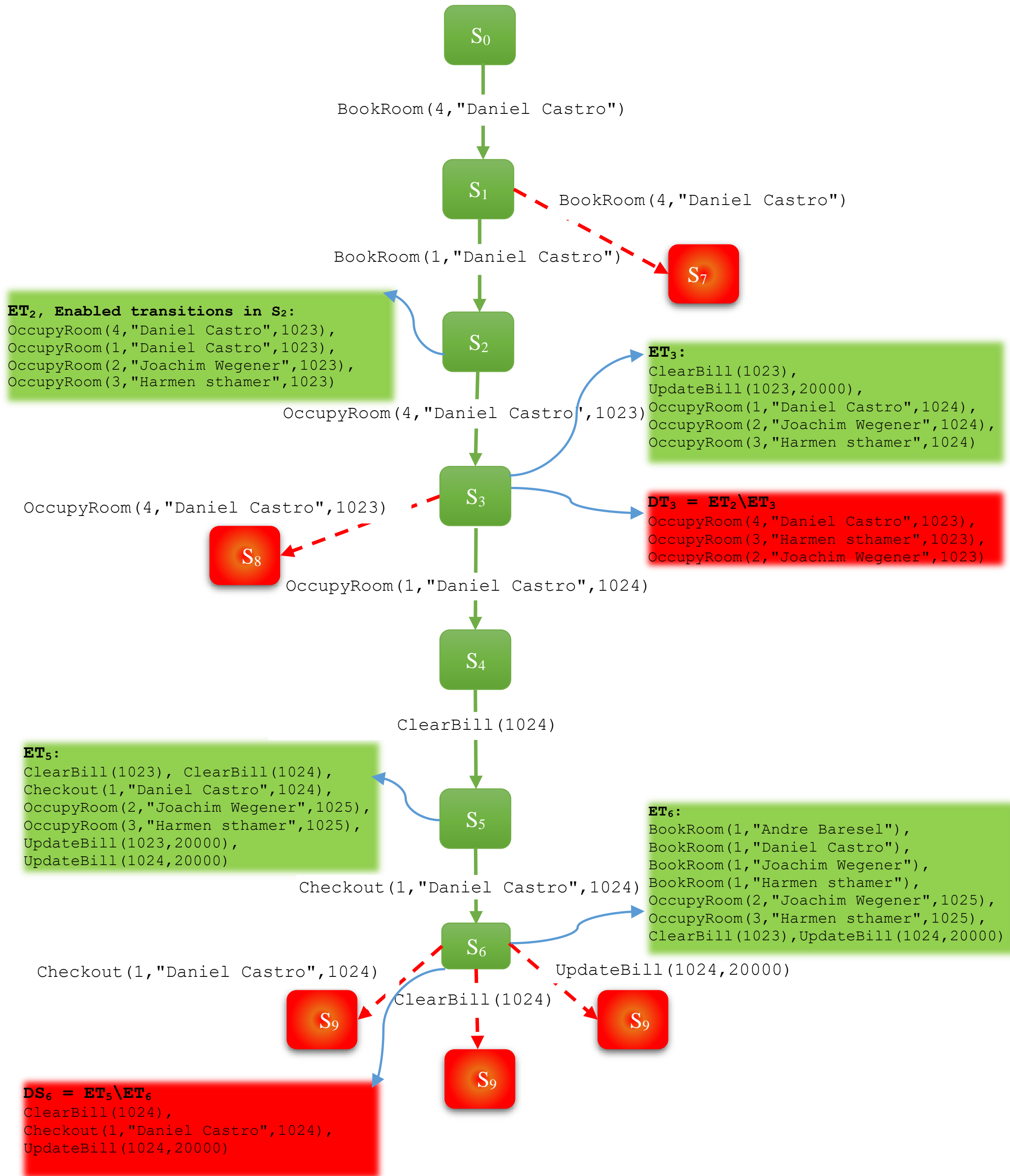


Figure 5. Detection of robustness transformation steps.

Figure 5 shows a simple robustness test case for the running example. As this figure showed, in order to find invalid steps, we use the conflicts that happen by each normal transformation step of the test case. For example, in S_2 there are four enable transitions (ET) i.e. ET_2={ *OccupyRoom(4,"Daniel Castro",1023), OccupyRoom(1,"Daniel Castro",1023), OccupyRoom(2, "Joachim Wegener",1023), OccupyRoom(3,"Harmen sthamer",1023)*}, while some of them i.e. DT_3={ *OccupyRoom(4,"Daniel Castro",1023), OccupyRoom(3,"Harmen sthamer",1023), OccupyRoom(2, "Joachim Wegener",1023)*} are disabled (DT) in S_3

by firing the transformation step *OccupyRoom(4,"Daniel Castro",1023)*. Therefore, disabled transitions could be used as robustness components of the test case, but it is possible that some of the disabled steps do not cover new data-conflict relationship and could not reveal more faults in the model. For example, in state S_3 the invalid transition *OccupyRoom(4,"Daniel Castro",1023)* covers all data-conflicts covered by the other two invalid steps in DT_3. Hence, the only invalid step *OccupyRoom(4,"Daniel Castro",1023)* in S_3 is used as robustness transformation step, but in S_6 all of the three invalid steps in DT_6 are needed as robustness components of the test case because each of them experiences data-conflict relations that do not cover by others in DT_6 or by all previous invalid steps of the test case.
Algorithm 4 investigates that what transformation steps are disabled (invalid steps) by firing each transformation step of the normal test path. Moreover, it shows what is the source of each invalid step in terms of the data-conflicts. In this algorithm, also the robustness component of the test case was generated by the detection of effective invalid steps for the normal test path.

Algorithm 4. Computation of the *Exercised Data-conflicts* of a test case

Input: *TestCase*
Output: *Exercised Data Conflicts dr, dd, and cf*

```
1:   dr, dd, cf ← {}
2:   CurrentPos=0;
3:   CurrentState=StartState;
4:   MatchResults previousMatches ← { }
5:   While CurrentPos < |normalTestPath| do
6:       Transition t ← Apply(normalTestPath [CurrentPos]);
7:       Rule R ← t.AppliedRule
8:       Matches curMatches ← {CurrentState.getMatches()};
9:       Matches disabledMatches= previousMatches\ curMatches;
10:      ForEach Match M ϵ disabledMatches
11:          appliedDeleteSet ← t.getRemovedElements();
12:          appliedCreateSet ← t.getCreatedElements();
13:          disabledDeleteSet ← M.getRemoedElements();
14:          disabledReadSet ← M.getReadElements();
15:          disabledForbidSet ← M.getForbiddenElements();
16:          Rule R1 ← M.AppliedRule
17:          ForEach Element e ϵ (appliedDeleteSet ∩ disabledReadSet)
18:              IF !dr.contains(<R,R1,e>)
19:                  dr ← dr ∪ {<R,R1,e>};
20:                  IS[CurrentPos] ← IS[CurrentPos] ∪ M.getTransformationStep();
21:                  // IS[CurrentPos] is a set of invalid transformation steps in the step
                     CurrentPos of normal testsequence
22:          ForEach Element e ϵ (appliedDeleteSet ∩ disabledDeleteSet)
23:              IF !dd.contains(<R,R1,e>)
24:                  dd ← dd ∪ {<R,R1,e>};
25:                  IS[CurrentPos] ← IS[CurrentPos] ∪ M.getTransformationStep();
26:          ForEach Element e ϵ (appliedCreateSet ∩ disabledForbidSet )
27:              IF !cf.contains(<R,R1,e>)
28:                  cf ← cf ∪ {<R,R1,e>};
29:                  IS[CurrentPos] ← IS[CurrentPos] ∪ M.getTransformationStep();
30:      CurrentState ←t.TargetState
```

4.4. Proposed memetic algorithm

In evolutionary algorithms, it is possible, individuals improve with any local refinements[44]. In the local search algorithms, the search process concentrates on neighbors of a candidate solution to find a better solution. While the global search algorithms make large changes in individuals to overcome local optima and find more globally optimal solutions. In this section, we present the proposed local search refinements as well as an integration of local and global search algorithms for test suite generation from GTS specification.

4.4.1. Restore covered test objectives

In search-based testing, usually, the fitness function shows the overall coverage score of a test suite. However, fitness does not indicate that what test goals and rules are covered. Hence it may be happening that a test suite with lower fitness score cover test goals that are not covered by a test suite with better fitness. To overcome this problem, we use an external archive of test cases to restore ones that cover new test goals for the first time. When we find a test suite with high coverage score but it does not cover some of the previously determined test goals, it could be enhanced by adding related test cases.

4.4.2. Local search improvement

The aim of the local search is making small changes in the test suite to concentrate on covering particular test goals and interactions. With this aim we define the local search at the test case level. When a test case is selected for local search, in each step of the test case we search for a better substitution among enabled transitions. As stated in the previous section, we keep track of the all covered test goals, so we could simply determine which test goals are not covered so far. Hence, in each step, a transition that covers new test goal or uncovered rule is a candidate for substitution. We us a greedy approach showed in Algorithm 5 for doing local search. Since fitness computation for each available transition in each step of a test case is a time consuming task, this is done for enabled/disabled transitions by the previous transformation step (Line 4). Substitution for each transformation step could be done if there is a better (valid) transition (Lines 5 to 10).

Algorithm 5. Local search on an individual(Test Suite) //a greedy algorithm

---

Input: Test-Suite T, uncovered RuleSet UR, covered rule_dependencies CRD
Output: *An improved Test_Suite*

1: **ForEach** *Test_Case Tc of T* **do**
2: curState ← S_0

3: **While** there is a *Transformation_Step t of Tc* **DO**
4: t' ← select a transition in curState w.r.t covering a new rule (not in UR), covering new test goal (not in CRD), or cover new rule of UR.
5: **IF** t' is valid
6: Apply t', and change t with t' in *Tc*
7: **Else IF** t is valid
8: Apply t
9: **Else**
10: Select an available transition t'' randomly, apply t'', and change t with t'' in *Tc*
11: **IF** *Tc* improved
12: **ADD** *Tc* to external archive

13: **Return** *T*

4.4.3. Test-suite generation

In this section, we extend a regular genetic algorithm by equipping it with the proposed local search operator before applying the regular global search operators. Algorithm 6 illustrates the proposed MA. The algorithm initially applied to a random population of chromosomes. Each chromosome represents a set of possible test paths of the SUT in the corresponding state-space. A chromosome initially generated through the application of a random feasible sequence of transformation steps on the start graph of the SUC. The local search operator is applied to a portion of the population with a predefined probability (Lines 5 to 8). Since in the proposed approach, test goals determined through the search process and minimal test length could not be predicted, the population evolves through the search operators until the search budget is used up (Lines 3 to 26). Line 13 and Lines 15 to 21 apply the global search operators i.e. crossover and mutation respectively. In our search-based test generation algorithm, the fitness score is the dominant goal of the optimization process, where the length of the test suite is the secondary goal. This means that we use a single objective genetic algorithm, but in the selection of chromosomes (test suites) for the next generation, the lengths of chromosomes are considered to rank chromosomes (chromosomes with the same fitness value are ranked by their lengths (Line 23). The length of a chromosome is the sum of the lengths of all the included test paths, which is calculated by relation 7 in section 4.3. At the end of the search process, the best

chromosome (a chromosome with the maximum fitness and minimum test length) is introduced as a best-searched test suite.

Algorithm 6. Generation of the Test Suite with MA

Input: A GTS model of the SUC
maxIterations, popSize, CrossOverRate, MutationRate, LocalSearchRate, maxTestSuiteLength, maxTestSequenceLength, elitSize, ImprovementRate
Output: *A set of Test Paths as a Test Suite*

1: *Population* ← *Generate initial random population (test cases of length maxTestSequenceLength)*
2: *BestIndividual* ← the best chromosome of the Population w.r.t fitness and test length
3: **For** *iteration* ← *1* to *maxIterations* **do**
    // Local Search
4:     **IF** *probability* < *LocalSearchRate*
5:         LocalSearchPopulation ← *Select Individuals for local search by ImprovementRate*
6:         **For each** *Individual of* LocalSearchPopulation **do**
7:             *newIndividual* ← **Do local search on** *Individual* by Algorithm 5.
8:             *Population* ← *Population* ∪ {*newIndividual*}\{*Individual*}

    // Global Search Genetic Algorithm
9:     *newPopulation* ← *elite(Best Individuals)*
10:     **While** |newPopulation | < *popSizse* **do**
11:         Par_1,Par_2 ←*select two chromosomes of Population by Tournament selection*

        // CrossOver
12:         **IF** *probability* < *CrossOverRate*
13:             Ch_1, Ch_2 ← CrossOver(Par_1,Par_2)
14:         **else**
15:             Ch_1, Ch_2 ← Par_1,Par_2

        // Mutation
16:         **For each** *step of test-cases in* Ch_1 **do**
17:             **IF** *probability* < *MutationRate*
18:                 Change the step by a random available transition, insert new step, or delete current step with the same probability.
19:         **For each** *step of test-cases in* Ch_2 **do**
20:             **IF** *probability* < *MutationRate*
21:                 Change the step by a random available transition, insert new step, or delete current step with the same probability.

22:         *Calculate fitness and length of* Ch_1 *and* Ch_2
23:         I_1,I_2 ← Rank {$Child_1$, $Child_2$, $Parent_1$, $Parent_2$ } *According to their fitness and length, then return two individuals by Tournament selection*

24:         *newPopulation* ← *newPopulation* ∪ { I_1,I_2 }
25:     *Population* ← *newPopulation*
26:     *Update(BestIndividual)*
27: **Return** *BestIndividual*

4.4.4. Crossover

In the proposed evolutionary test generation process, two distinct crossover operators are designed, namely the test suite level and test case level crossover.

Test suite level crossover: In the test suite level crossover, two chromosomes (P_1, P_2) as parents are recombined, and two offspring ($child_1$, $child_2$) are generated. Figure 6 shows how test suite level crossover applied to a couple

of chromosomes. In this operator, a cross point c_1 and c_2 are assumed as random integer values in the range of *1* to the number of test cases of P_1 ($|P_1|$) and P_2 ($|P_1|$) respectively. The combination of the first c_1 -1 test cases of P_1 and the last $/P_2/$- c_2 test cases of P_2 composes the $child_1$, and the combination of the first c_2 -1 test cases of P_2 and the last $/P_1/$- c_1 test cases of P_1 creates the $child_2$. As showed in Figure 6, the resultant children may have different size with respect to the parents. Hence, the size of individuals may be changed through the search process.

Test case level crossover: In the test case level crossover, two test cases, i.e. $parent_1$ and $parent_2$, are recombined to construct two new children, say $child_1$ and $child_2$. In the test case level crossover, the cross point p is a random number in [$1 .. min(length(parent_1), length(parent_1))$], and the combination of the first p -1 transformation steps of $parent_1$ and the last $length(parent_2)$- p steps of $parent_2$ composes the $child_1$, and the combination of the first p -1 transformation steps of $parent_2$ and the last $length(P_1)$- p steps of $parent_1$ creates the $child_2$. In this operator, the generated test cases (i.e. offspring) may be invalid test cases through the precondition violation of transformation steps that comes after the cross point, hence they are repaired by random transformation steps in fitness calculation.

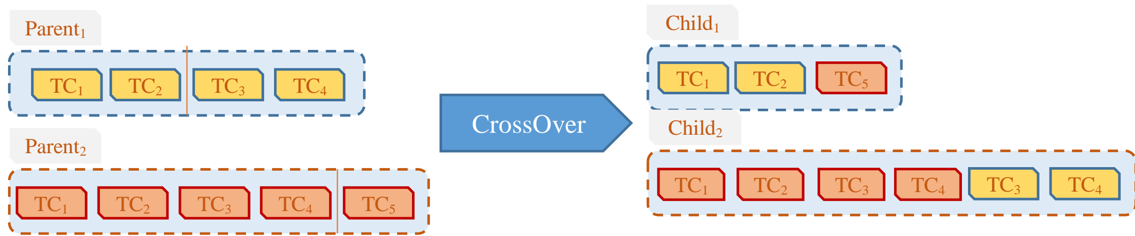


Figure 6. Test-suite-level Crossover.

5. Experiments

In this paper, a robustness testing and a hybrid strategy based on the existing normal data-flow and the proposed data-conflicts are introduced for software models specified through graph transformation system. Furthermore, to search in a large state-space of complex systems, a memetic algorithm is used. In this section, we evaluate the fault detection capability and cost-effectiveness of the proposed MA and the proposed testing strategies. Specifically, we need to address the following research questions:

Q1. What are the fault detection capabilities of the proposed testing strategies?
Q2. What are the testing costs of the proposed testing strategies?
Q3. Does local search improve the coverage score and testing costs of the output tests?
Q4. How does the test generation algorithm converge?
Q5. How does the proposed approach improve the fault detection capability w.r.t the state-of-the-art?
Q6. To what extent the proposed approach affects the testing cost w.r.t the state-of-the-art?

In this section, to assess the fault detection capability of the proposed testing strategies, a type of mutation analysis is used. This type of analysis is described in the next subsection. Then the effectiveness, and performance of the proposed approach will be evaluated. Subsection 5.2 describes the case studies and parameter settings used in our evaluation. Then, the performance of the proposed testing strategies (Q1 and Q2) are evaluated in subsection 5.3, while the performance of the MA (Q3 and Q4) is evaluated in subsection 5.4. improvements achieved by the proposed approach concerning Q5 and Q6 are described in subsection 5.5.

5.1. Fault detection assessment

One of the well-known techniques for assessment of the fault detection capability of testing approaches is the mutation analysis technique [45]. In our evaluation process, mutation analysis should be done at the model-level for GTS specifications. A faulty version of the specification is called a mutant. A mutant that contains only a simple fault is the first-order mutant, while higher-order mutant contains more faults [46]. When a test case T reveals differences between a mutant M and the original behavior, we say that test case T can kill the mutant *M*. In other words, the execution of *T* over *M* differs from that the application over the original model. A test suite that kills first-order mutants, it is likely to kill higher-order ones [46, 47]. Hence, first-order mutation analysis could evaluate

the test generation approach as well as needed. It is worth noting that all mutants should be syntactically correct specification according to the type graph, but semantically different from the original specification.

The production rules as the main components of GTS specification, and the start graph (host graph) construct a formal specification of the SUC. Hence, production rules are the best component of the specification for fault injection to generate mutants. In this section, we introduce a set of mutation operators based on common mistakes that could occur in GTS specifications to alter production rules in mutants.

As mentioned in section 2, a production rule contains elements that operate as Creator/Reader/Eraser/Forbidden. Change the role of an element of the production rule, and remove/insert an element from/to the rule are the main sources of the fault injection to the production rule. In the GTS context, a mutant is generated by creating a copy of the original model, followed by inserting a simple fault to one of the rules. Table 4 illustrates all the possible types of mutation operators. For example, the first row of the table indicates that we can delete a creator *C* from a production rule (Creator Deletion Operator CDR), and replace a creator *C* by a reader/eraser/forbidden component of the same element type (Creator Operator Replacement COR). Moreover, in a production rule, attributes of nodes updated through arithmetic/ logical/relational/string operators. Another type of fault that alters the production rules includes an operator replacement in the mentioned expressions, such as the MuJava operator replacement [48]. Table 5 lists all the possible operators used in GTS. This type of mutant is generated by an operator replacement with the same operator type and an expression omission/insertion from/into the production rules.

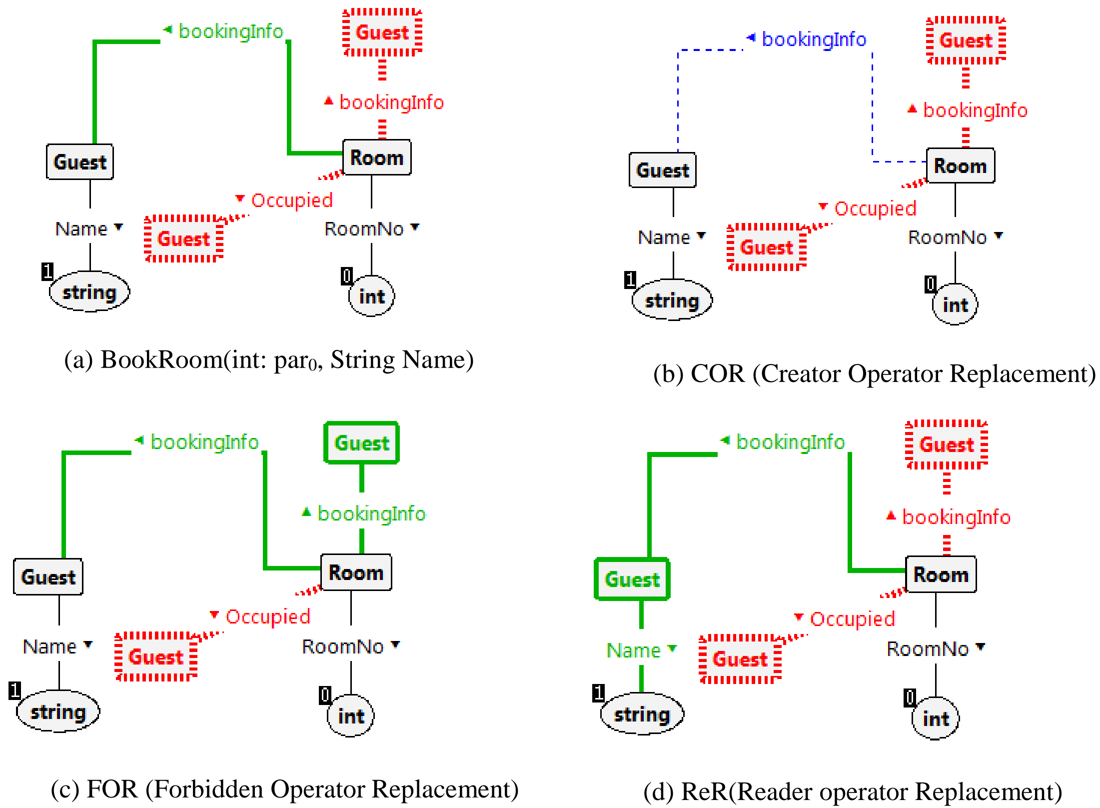
Table 4. All the possible types of faults based on the roles of the elements in a production rule.

Operators	Mutations
Creator *C*	Delete (CDO) or convert to { Reader, Eraser, Forbidden} (COR)
Reader *R*	Delete (RDO) or convert to { Creator, Eraser, Forbidden} (ROR)
Eraser *E*	Delete (EDO) or convert to { Creator, Reader, Forbidden} (EOR)
Forbidden *F*	Delete (FDO) or convert to { Creator, Reader, Eraser} (FOR)

Table 5. Proposed mutation operators for GTS specification.

Type	Mutation operator	Description
Node- and Edge-Based Mutation	RDO	Reader Deletion Operator
	CDO	Creator Deletion Operator
	EDO	Eraser Deletion Operator
	FDO	Forbidden Deletion Operator
Reader Mutation	ReR	Reader operator Replacement
Creator Mutation	COR	Creator Operator Replacement
Eraser Mutation	EOR	Eraser Operator Replacement
Forbidden Mutation	FOR	Forbidden Operator Replacement
Arithmetic Mutation	AOR	Arithmetic Operator Replacement{add, sub, mul, div, mod, min, max, let/test }
Relational Mutation	ROR	Relational Operator Replacement { lt, le, gt, ge, eq, neg, toString }
Logical Mutation	LOR	Logical Operator Replacement {and, or, not, eq, true, false, let/test, exists/forallx }
String Mutation	SOR	String Operator Replacement { concat, lt, le, gt, ge, eq, let/test }

According to the above discussion, for each element (Reader, Creator, Eraser, and Forbidden) of a production rule, there are two possible operations to inject a fault in a production rule include deletion and replacement. Table 4 lists the used mutation operators for mutant generation from the original GTS specification of the SUC. In our experiments, we create mutants for each target system by applying mutation operators to the elements of the production rules by an automatic mutant generator developed in the GROOVE toolset. Each mutant is generated through the application of a mutation operator to one element of a production rule of the model (first-order mutation). The application of a mutation operator on a rule of the GTS specification should generate syntactically correct mutants. Hence, mutants with syntactic errors would be removed automatically by the mutant generator. Since each operator changes exactly an element of a production rule and each element of the rule mutated by an operator only once, the resultant mutant differs from the original and other mutants. Therefore, the mutant generator does not generate equivalent mutants.



(a) BookRoom(int: par$_0$, String Name)

(b) COR (Creator Operator Replacement)

(c) FOR (Forbidden Operator Replacement)

(d) ReR(Reader operator Replacement)

Figure 7. Some of the mutants of the running example.

Table 6. Rules and mutants of the case studies.

Case study	Number of rules	Number of mutants
Hotel Management Service (HMS)	7	123
Dining Philosophers (DPs)	6	142
Online Shopping System (OSS)	20	395
Bug Tracker System (BTS)	34	432
Travel Agency System (TAS)	43	609
Total	110	1701

Table 7. Parameter setting of the test generation algorithm.

Parameter	Value
maxIterations	100
popSize	30
maxTestSuiteLength	6
maxTestCaseLength	100
elitSize	2
CrossOverRate	0.7
MutationRate	0.05
LocalSearchRate	0.4
ImprovementRate	0.2
W (PSO)	0.8
C_1, C_2 *(PSO)*	0.2
MaxVelocity (PSO)	0.2

For example, Figure 7 shows several generated mutants for our running example based on the production rule *BookRoom*. Figure 7.a shows the original specification of the rule. Figure 7.b is the result of the application of the *COR* operator on the original production rule in which the *new* operator is replaced by *the del* operator from the *bookingInfo* edge. Figure 7.c shows a mutant that is generated through the application of the FOR operator on the forbidden *bookingInfo* edge and corresponding *Guest* Node in which the operator *not* is replaced

by *new* (creator operator). In Figure 7.d, the *Reader* element *Guest* is converted to the *creator* element through the application of *the ReR* operator.

In the proposed mutation analysis, a mutant is weakly killed by a test case if the path from the execution of the mutant differs from the resulting sequence of the original model. Weakly checking of mutants is a costly task, because it needs to a comparison of every state of the execution path in both mutant application and original specification application. On the other hand, when a mutant lead to a different final state, it is said strongly killed. However, the efforts required to evaluate strongly includes a comparison of the final states in the mutant application and the original model. In practice, the evaluation of the mutants is very costly and error-prone, particularly in the case of detecting defects that do not change the final state/output. Therefore, it is preferable to design test cases that kill mutants strongly. The proposed process of mutation analysis for each case study include:

1- specify the SUC through the graph transformation system.
2- Generate all possible mutants of the system by the application of the mutant generator.
3- Generate a test suite based on the strategy under evaluation through the proposed evolutionary approach.
4- Execute test cases against each mutant and check whether the mutant is killed based on each killed strategy or not.

5.2. Case studies and experimental settings

Five well-known case studies are used for the evaluation of the proposed testing strategies. The first case study is the running example, Hotel Management Service (HMS) [19], which described in section 2.2. The second case study is a Dining Philosophers (DPs) problem [49]. In this problem, several philosophers sit around a table and do their philosophical work. When each philosopher goes hungry, he/she eats from the food located on the table. Besides the food, there is a fork between every two philosophers. Each philosopher should take two forks, namely the *left* and *right* ones, to eat food. At first, all philosophers are in the *thinking* state, and after a moment, some of the philosophers may get *hungry*. Each *hungry* philosopher tries to eat using his/her forks. The *right* fork is taken after taking the *left* one. When the *right* fork is busy (it has been taken by another philosopher), the *left* fork is released, and the corresponding philosopher remains in the *hungry* state. After a while, he/she tries again. After *eating* enough food, the philosopher releases both forks and goes to the *thinking* state. The other case studies are Online Shopping System (OSS), described in [27], Bug Tracker System (BTS), described in [19], and Travel Agency System (TAS), presented in [28]. Table 6 lists the main metrics on these case studies.

Although the first and the second case studies are rather small, they are illustrative enough to indicate the differences of our proposed test generation strategies. Moreover, they have very large state spaces when applied to the big start graphs. A state-space-explosion problem occurs in DPs when we define the host graph with about 20 or more philosophers. This configuration can evaluate the performance of the proposed test generation algorithm. Furthermore, OSS, BTS, and TAS have large state-spaces and the state-space-explosion problem occurs in them. They are also well-known case studies in the literature for evaluating graph GTS-based test generation algorithms [21].

Search-based algorithms have several parameters that affect the performance of the algorithm. Table 7 listed the general parameters in our test generation algorithm. For many of these parameters there are common settings in the literature, or there are best practices based on the past experiences. In this research we use the common settings for standard global search operators such as *CrossOverRate*, and *MutationRate* in Genetic, or all parameters of the PSO algorithm. Other parameters such as size of the population and number of generations are set based on the past experience [21]. Moreover, based on the past experience we limit the length of test cases to 100 transformation steps, and 6 test cases are allowed to be in a test suite.

There are two new parameters in the local search component of the proposed algorithm i.e. *LocalSearchRate*, and *ImprovementRate*, in which appropriate setting could balance between exploration and exploitation. Generally, high rate selection for local search parameters leads to better coverage, but in our investigation, we gain the best coverage score by *LocalSearchRate*=0.4 and *ImprovementRate*=0.2, while higher settings do not lead to better coverage. It is worth noting, that since in local search we just search in enabled/disabled transitions for each transformation step, it does not very time consuming task. However, more tuning investigations and time analysis could lead to find better settings, but this task is computational expensive. Hence, in this research we preferred to focus on showing that the proposed coverage criteria, based on data-conflicts (and related testing strategies i.e. robustness and hybrid testing), are capable to reveal some faults in GTS specifications that could not be determined by the existing normal testing. Moreover, we focus on illustrating that the proposed local search component could lead to better coverage score. Therefore, we postpone more tuning investigations to the future work.

5.3. Performance of the proposed testing strategies

In this section, we explain our findings from the experiments concerning Q1 and Q2 research questions. The research question Q1 evaluates the effectiveness of the proposed testing strategies (robustness and hybrid testing) and compares them with the existing normal testing. The cost of each testing strategy is the subject of research question Q2.

Table 8. Comparison of an average and standard deviation of coverage score/fault-detection-capability of the proposed testing strategies versus existing normal testing strategy.

Case study		Normal testing	Robustness testing	Hybrid testing
HMS	Coverage	30.8 ± 0.1	26.7 ± 0.5	**57.8 ± 0.4**
	#Weakly killed	92.2 ± 2.8	98.6 ± 1.6	**104.2 ± 2.6**
	#Strongly killed	74.3 ± 1.2	76.2 ± 2.5	**80.7 ± 1.2**
DPs	Coverage	18 ± 0	11 ± 0	**28.5 ± 0.4**
	#Weakly killed	113.2 ± 0.97	130.5 ± 0.9	**130.6 ± 0.8**
	#Strongly killed	110.2 ± 1.3	109.9± 1.3	**110.6 ±0.9**
OSS	Coverage	218.6 ± 1.3	104.2 ± 2.5	**316.1 ± 4.7**
	#Weakly killed	348.4 ± 0.7	359.2 ± 6.2	**368.7 ± 9.6**
	#Strongly killed	338.3 ± 2.5	341.9 ± 6.2	**345.8 ± 7.3**
BTS	Coverage	266.4 ±8.2	195.3 ± 8.9	**465.2 ± 9.4**
	#Weakly killed	351.2 ± 13.1	321.2 ± 11.9	**385.3 ± 12.1**
	#Strongly killed	333.9 ± 9.2	295.9 ± 12.5	**344.1 ± 6.2**
TAS	Coverage	459.4 ±8.2	287.3 ± 8.9	**737.4 ± 12.2**
	#Weakly killed	491.4 ± 16.4	443.2 ± 17.4	**543.4 ± 22.5**
	#Strongly killed	462.3 ± 19.3	395.6 ± 15.7	**508.8 ± 18.4**

Table 9. Comparison of the average and standard deviation of test-size, transformation-steps needed per killed mutant of the proposed testing strategies versus the normal testing strategy in both weakly and strongly fault detection methods.

Case study	Metric	Normal testing	Robustness testing	Hybrid testing
HMS	Test size	33.4 ± 9	**29.5 ± 8.7**	36.4 ± 9.6
	TS/WKM	0.36	**0.30**	0.35
	TS/SKM	0.45	**0.39**	0.45
DPs	Test size	80.0 ± 4.9	**56.2 ± 13.2**	107.2 ± 14.9
	TS/WKM	0.73	**0.51**	0.97
	TS/SKM	0.71	**0.43**	0.81
OSS	Test size	158.8 ± 6.6	**115.1 ± 11.4**	193.8 ± 15.8
	TS/WKM	0.46	**0.32**	0.53
	TS/SKM	0.47	**0.34**	0.56
BTS	Test size	275.2 ± 14.4	**173.4 ± 16.7**	292.3 ± 15.6
	TS/WKM	0.78	**0.54**	0.76
	TS/SKM	0.82	**0.59**	0.85
TAS	Test size	225.1 ± 13.8	**167.5 ± 12.3**	262.5 ± 19.7
	TS/WKM	0.46	**0.38**	0.48
	TS/SKM	0.49	**0.42**	0.52

Q1. What are the fault detection capabilities of the proposed testing strategies?

Table 8 shows the fault detection capability of three testing strategies. The first column of the table represents the case study under the experiment. The second column states the metrics evaluated in the next columns. The Coverage row describes to what extent each testing strategy covers the data-dependencies/conflicts between the transformation rules. In each experiment, we evaluate the fault detection capability against the introduced types of fault detection strategy, *i.e.* the weak- and the strong fault detection strategies. The other columns (numerical columns) of the table show the mean and the standard deviations of the mentioned evaluation criteria for 10 times of test suite generation, execution, and evaluations.

As stated in the coverage row of Table 8, the hybrid testing strategy covers more data-relations (dependencies and conflicts) than the other testing strategies in all case studies. Hence, it is obvious that the mentioned strategy capable of detecting more faults than the others. This is illustrated in the table by highlighted cells in comparison with other cells in the same row for both fault detection methods (weakly and strongly killing methods). Another observation is that the robustness testing strategy could detect more faults than normal testing for DPs and OSS case studies, while this is not valid for the other cases. Although the fault detection capability of the normal testing and the robustness testing have no significant difference in HMS, the notable point is that they could not detect the same set of faults. In other words, robustness testing identifies some faults that could not be detected by normal testing and vice versa. This is supported by using both strategies in the hybrid testing, as showed by the results that the hybrid testing identifies faults that have been detected either by robustness testing or normal testing.

Q2. What are the testing costs of the proposed testing strategies?

Table 9 shows a comparison between the proposed strategies based on the average and standard deviation of the test-size and testing costs, in both fault detection methods (weakly and strongly), in terms of the number of transformation steps required per killed mutant. For example, in the DPs case study, the row labeled by "TS/WKM" states that to weakly kill a mutant, in the normal testing strategy, on average 0.36 transformation steps are needed. As the highlighted cells show, the robustness testing strategy is less costly than the other strategies for all case studies in both weakly and strongly fault detection methods. As mentioned in section 4, a robustness test case includes a normal test sequence along with invalid transitions. So, the normal component of the test case covers some data-dependencies. Therefore, covering data-conflicts through the robustness test cases subsume some of the data-dependencies. Hence, the robustness test cases can detect faults related to some of the data-dependencies other than data-conflicts.

Figure 8 and Figure 9 show the cost-effectiveness of each testing strategy in both weakly and strongly fault detection methods for each case study and an average of them. As these figures illustrate, the robustness testing strategy is the most cost-effective strategy in all case studies and on the average cost of all case studies.

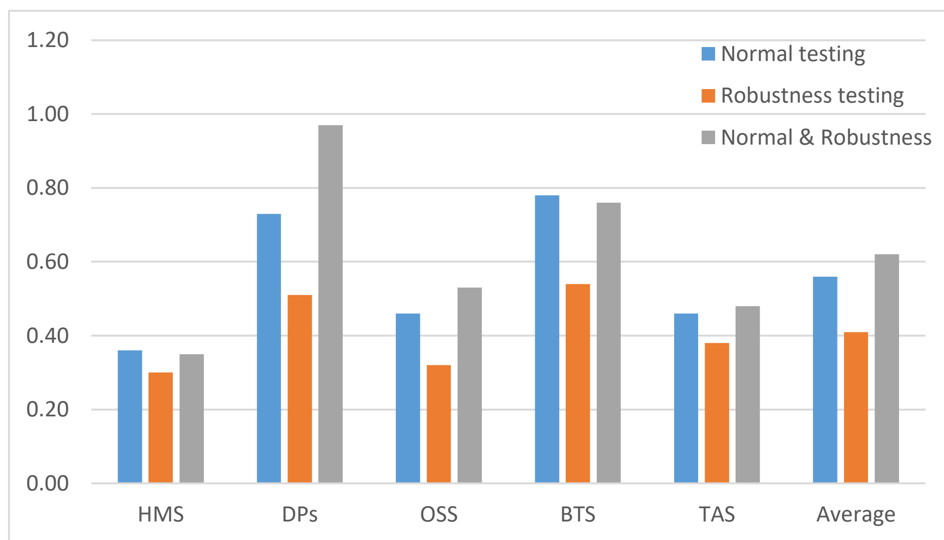


Figure 8. Cost of each testing strategy in weakly fault detection method for each case study and an average of them.

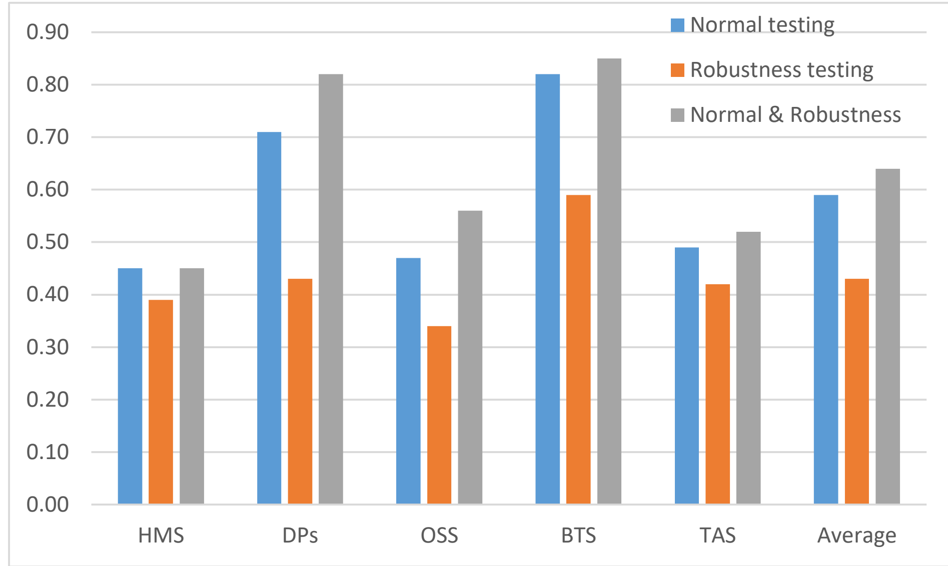


Figure 9. Cost of each testing strategy in strongly fault detection method for each case study and an average of them.

5.4. Performance of the proposed memetic algorithm

In the previous section, we have seen that the robustness testing based on data-conflict relationships identifies some faults that could not be detected by normal testing and vice versa. Moreover, the hybrid testing identifies faults that have been detected either by robustness testing or normal testing. So, in this section we evaluate the proposed test generation algorithm i.e. MA, based on the hybrid testing strategy to show the effectiveness of the proposed local search refinements and it cost-effectiveness (Q3). With this aim, we compare MA with three search-based algorithms that have been used in testing based on GTS specifications, including GA, PSO, and HGAPSO. Statistical difference is evaluated with a two-tailed Mann–Whitney–Wilcoxon U-test, while improvements of MA is measured with the Vargha–Delaney ($\hat{A}_{12}$) effect size at 0.05 significant level. Moreover, we focus on the convergence of MA (Q4). As the coverage score is the primary objective in the proposed MA, most of the evaluations are performed based on this metric.

Q3. Does local search improve the coverage score and testing costs of the output tests?

Table 10 shows a comparison between MA and selected search algorithms i.e. GA, PSO, and HGAPSO based on average and standard deviation of coverage score, length of the output tests, and the number of Transformation Steps needed per a Covered Test Objective (TS/CTO) as cost of covering a test goal. This experiment is done based on the hybrid testing strategy for all selected algorithms for 10 times of test suite generation, execution, and evaluations.

As stated in the coverage row of Table 10, MA covers more data-relations (dependencies and conflicts) than the other testing strategies in all case studies except DPs case study, which has no significant difference with HGAPSO algorithm. This is also true for the third row (TS/CTG) of each case study, which shows the cost-effectiveness of search algorithms in terms of the number of transformation steps required to cover a test objective.

Results in Table 11 answer Q3 by clearly showing, with high statistical confidence, that the MA outperforms the selected algorithms in many, case studies. In this table, Effect sizes with statistically significant difference at 0.05 level are shown in bold. There are no significant differences between MA, GA, and HGAPSO for small case studies i.e. HMS and DPs, that have small state space. But, there are a significant difference between the proposed MA and the other search algorithms (MA; other) for large case studies.

Q4. How does the test generation algorithm converge?

To answer the research question Q4, we conduct experiments in which the convergence of MA is investigated. In these experiments, the test generation algorithm was applied on large instances of the case studies (e.g. DPs (30) with 30 philosophers) that have very large state-space. As HMS is a small system, it is not considered in this evaluation. Figure 10 shows how MA converges in the selected instances based on the average coverage score. Each curve depicts the convergence of a case study with the average coverage of 10 experiments, which is carried out based on the hybrid testing strategies.

Table 10. Comparison of an average and standard deviation of coverage score, length of the output tests, and the number of Transformation Steps needed per a Covered Test Objective (TS/CTO) of MA versus other search-based algorithms.

Case study	Metric	GA	PSO	HGAPSO	MA
HMS	Coverage	57.1 ± 1.1	51.9 ± 2.8	57.6 ± 0.2	**57.8 ± 0.4**
	Test size	38.2 ± 10.4	46.9 ± 10.8	39.2 ± 12.3	36.4 ± 9.6
	TS/CTO	0.67	0.9	0.68	**0.63**
DPs	Coverage	28.2 ± 0.8	25.2 ± 1.3	**28.6 ± 0.5**	28.5 ± 0.4
	Test size	112.1 ± 18.3	110.1 ± 17.8	101.8 ± 11.2	107.2 ± 14.9
	TS/CTO	3.97	4.37	**3.56**	3.76
OSS	Coverage	298.1 ± 8.2	264.1 ± 14.2	301.8 ± 6.8	**316.1 ± 4.7**
	Test size	211.6 ± 17.3	163.7 ± 25.8	203.5 ± 19.2	193.8 ± 15.8
	TS/CTO	0.71	0.62	0.67	**0.61**
BTS	Coverage	431.6 ± 19.3	395.2 ± 22.1	446.7 ± 12.9	**465.2 ± 9.4**
	Test size	324.9 ± 32.1	274.8 ± 17.3	312.1 ± 21.7	292.3 ± 15.6
	TS/CTO	0.75	0.69	0.69	**0.63**
TAS	Coverage	696.9 ± 21.5	642.5 ± 39.4	721.2 ± 15.3	**737.4 ± 12.2**
	Test size	288.7 ± 33.1	247.6 ± 30.5	269.8 ± 23.2	262.5 ± 19.7
	TS/CTO	0.41	0.38	0.37	**0.36**

Table 11. Comparison of the effect size ($\hat{A}_{12}$) of the average coverage between the proposed MA and the other search algorithms (MA; other). Effect sizes with statistically significant difference at 0.05 level are shown in bold.

Algorithm	HMS	DPs	OSS	BTS	**TAS**
GA	0.52	0.55	**0.85**	**0.91**	**0.82**
PSO	**0.97**	**0.89**	**1**	**1**	**1**
HGAPSO	0.51	0.49	0.63	**0.73**	**0.71**

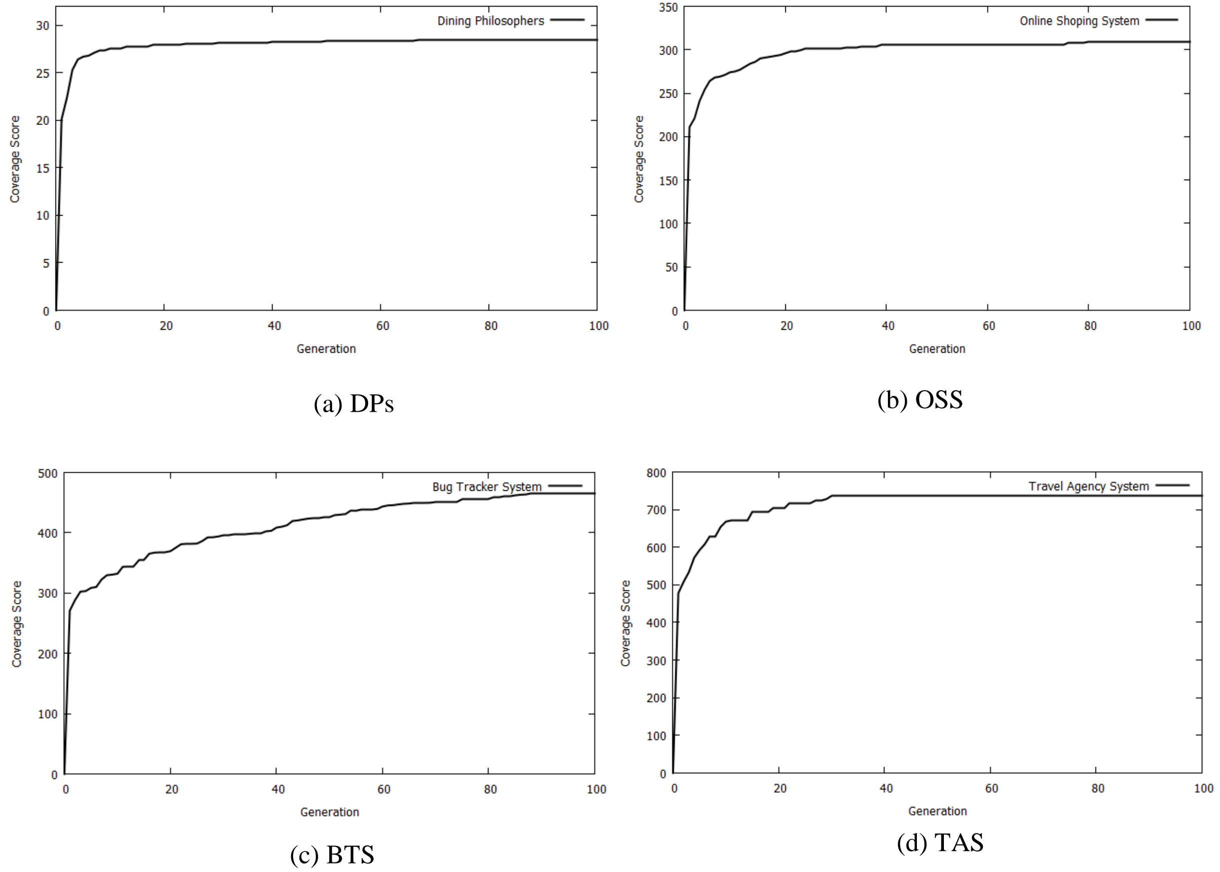
5.5. Comparison with the state-of-the-art

As aforementioned, VCT [19] and HGAPSO (data-flow) [21] are the most related works to the proposed testing approach. In this section, we do a comparison between the proposed testing strategies and these related works, based on the coverage score and the cost of testing, with respect to Q5 and Q6 research questions.

Q5. How does the proposed approach improve the fault detection capability w.r.t the state-of-the-art?

Q6. To what extent the proposed approach affects the testing cost w.r.t the state-of-the-art?

Table 13, with the same structure as Table 8, reports the new experiments to show how the proposed approach improves the fault detection capability of the GTS specification testing with respect to the mentioned related works. It is notable that, in this table, the coverage score shows the covered rule-dependencies/-conflicts instead of data-dependencies/-conflicts. This is because the available implementations of the related works measure the rule-dependencies. If the application of rule A depends on the application of B, the application of A may need several data elements created, updated, or deleted by B. So, usually there are several data-relationships per rule-dependency/conflict. In testing, each of the shared data between production rules could be a source of a fault. Therefore, it is reasonable to investigate all data-relations, and this is considered in the proposed approach. According to the mutants killed by each detection method, the hybrid strategy has a significant difference with the related works for all case studies. Hence, this strategy outperforms related works (Q5). This is expected because, according to the coverage score, the hybrid strategy covers more data-relationships than related works. In other words, the faults related to the uncovered data-dependencies/-conflicts could not be detected by the VCT/HGAPSO approach.



(a) DPs

(b) OSS

(c) BTS

(d) TAS

Figure 10. Convergence of the test generation algorithm in the case studies introduced, based on the first objective (fitness).

Table 13. Comparison of average and standard deviation of coverage score/fault-detection capability of the proposed testing strategies versus related works. Coverage metric is calculated as the number of the covered rule-dependencies/-conflicts

Case study		VCT	HGAPSO (data-flow)	Robustness testing	Hybrid testing
HMS	Coverage	13	8 ± 0	11.6 ± 0.5	**28.4 ± 0.6**
	#Weakly killed	71	69.2 ± 4.8	98.6 ± 1.6	**104.2 ± 2.6**
	#Strongly killed	67	55.4 ± 5.2	76.2 ± 2.5	**80.7 ± 1.2**
DPS	Coverage	5	2 ± 0	10 ± 0	**27.5 ± 0.4**
	#Weakly killed	91	84.7 ± 7.5	130.5 ± 0.9	**130.6 ± 0.8**
	#Strongly killed	89	79.7 ± 6.5	109.9± 1.3	**110.6 ±0.9**
OSS	Coverage	36	46.8 ± 0.4	32.1 ± 1.5	**106.1 ± 2.7**
	#Weakly killed	295	312.7 ± 5.2	359.2 ± 6.2	**368.7 ± 9.6**
	#Strongly killed	283	297.6 ± 6.8	341.9 ± 6.2	**345.8 ± 7.3**
BTS	Coverage	32	84.6 ± 1.3	62.3 ± 1.9	**163.2 ± 2.4**
	#Weakly killed	247	325.6 ± 8.7	321.2 ± 11.9	**385.3 ± 12.1**
	#Strongly killed	223	309.5 ± 7.2	295.9 ± 12.5	**344.1 ± 6.2**
TAS	Coverage	39	85.4 ±0.5	71.3 ± 2.9	**196.4 ± 2.1**
	#Weakly killed	309	451.5 ± 11.1	443.2 ± 17.4	**543.4 ± 22.5**
	#Strongly killed	276	402.6 ± 7.8	395.6 ± 15.7	**508.8 ± 18.4**

Table 14. Comparison of average and standard deviation of coverage-level, test-size, fault-detection-capability/ transformation-step of the proposed testing strategies versus existing normal testing strategy in both weakly and strongly mutant kill methods.

Case study	Metric	VCT	HGAPSO (data-flow)	Robustness testing	Hybrid testing
HMS	Test size	51	33.4 ± 9	**29.5 ± 8.7**	36.4 ± 9.6
	TS/WKM	0.72	0.48	**0.30**	0.35
	TS/SKM	0.76	0.6	**0.39**	0.45
DPs	Test size	64	102.0 ± 14.9	**56.2 ± 13.2**	107.2 ± 14.9
	TS/WKM	0.7	1.2	**0.51**	0.97
	TS/SKM	0.76	1.3	**0.43**	0.81
OSS	Test size	59	86.2 ± 10.6	115.1 ± 11.4	193.8 ± 15.8
	TS/WKM	**0.2**	0.27	0.32	0.53
	TS/SKM	**0.21**	0.29	0.34	0.56
BTS	Test size	117	178.6 ± 19.9	173.4 ± 16.7	292.3 ± 15.6
	TS/WKM	**0.47**	0.55	0.54	0.76
	TS/SKM	**0.52**	0.59	0.59	0.85
TAS	Test size	109	161.1 ± 11.2	167.5 ± 12.3	262.5 ± 19.7
	TS/WKM	**0.29**	0.36	0.38	0.48
	TS/SKM	**0.32**	0.4	0.42	0.52

Table 14 shows the comparison of the cost of fault detection in the proposed strategies and the related works. The cost is measured as the number of transformation steps required per killed mutant. The comparison of the cost in the proposed hybrid testing with the VCT/HGAPSO shows that the proposed strategy is costlier than others for all case studies. Indeed, to cover some of the data-dependencies/-conflicts, some rules should be applied repeatedly, which leads to lengthy test cases. In other words, not all faults have the same cost to detect. On the other hand, the robustness strategy is less costly than other strategies in some case studies. This is again expected because covering conflict is far simpler than covering all data-dependencies. However, VCT has the least cost in three case study, but it covers a small subset of data-dependencies, and detect far less faults the proposed strategies (Q6).

6. Conclusion and future works

Model testing is performed to obtain some level of confidence in the validity of the model against its intended purposes and ensure the quality of models. Model testing approaches try to reveal the faults of the model under test. Robustness testing is a well-known approach in the software testing context, which evaluates vulnerabilities of a system under unexpected events. In this paper, a set of data-conflict coverage criteria in the context of GTS specification is introduced to guide the robustness test generation process. Furthermore, a robustness test generation approach is introduced to testing software models specified through GTS formalism. Then, we have investigated the combination of the existing normal testing based on data-dependencies and robustness testing as a hybrid testing strategy. The hybrid strategy could cover both data-dependencies and data-conflicts among production rules. Moreover, a search-based testing process MA is proposed for all strategies based on some local search refinements and global search genetic algorithm to maximize the coverage score in each strategy and minimize the length of the output test.

The effectiveness of the proposed testing strategies and the introduced search-based test generation process (MA) are evaluated through a type of mutation analysis at the model-level. Our experiments based on five well-known case studies show that the hybrid testing strategy outperforms the existing normal testing approach and the proposed robustness testing in terms of fault-detection capability, while the robustness testing is more cost-efficient than others. Moreover, the proposed hybrid evolutionary testing outperforms the most related works in terms of fault detection capability.

There are several directions for improvement in future works. For example, there are general enhancements in the literature for the adaptive parameter setting of search algorithms [50, 51], which could be used in the proposed test

generation algorithm for the achievement of better coverage. Furthermore, approximate approaches [52-54] are able to generate near optimal solutions more accurately.

References

1. Hutchison, C., et al. *Robustness testing of autonomy software*. in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 2018. ACM.
2. Mattiello-Francisco, F., et al., *InRob: An approach for testing interoperability and robustness of real-time embedded software.* Journal of Systems and Software, 2012. **85**(1): p. 3-15.
3. Balci, O., *Verification, validation, and testing.* Handbook of simulation, 1998. **10**: p. 335-393.
4. Savary, A., et al. *Model-based robustness testing in Event-B using mutation*. in *SEFM 2015 Collocated Workshops*. 2015. Springer.
5. Strug, J., *Mutation testing approach to negative testing.* Journal of Engineering, 2016. **2016**.
6. Rafe, V., M. Rahmani, and K. Rashidi, *A survey on coping with the state space explosion problem in model checking.* International Research Journal of Applied and Basic Sciences, 2013. **4**(6): p. 1379-1384.
7. Di Nardo, D., et al. *Evolutionary Robustness Testing of Data Processing Systems Using Models and Data Mutation (T)*. in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015. IEEE.
8. Vos, T.E., et al., *Evolutionary functional black-box testing in an industrial setting.* Software Quality Journal, 2013. **21**(2): p. 259-288.
9. Ferrer, J., et al., *Search based algorithms for test sequence generation in functional testing.* Information and Software Technology, 2015. **58**: p. 419-432.
10. Committee, I.S.C., *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos.* CA: IEEE Computer Society, 1990. **169**.
11. Araujo, W., L.C. Briand, and Y. Labiche, *On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software.* IEEE Transactions on Software Engineering, 2014. **40**(10): p. 971-992.
12. Meyer, B., *Object-oriented software construction*. Vol. 2. 1988: Prentice hall New York.
13. Ehrig, H. and K. Ehrig, *Overview of formal concepts for model transformations based on typed attributed graph transformation.* Electronic Notes in Theoretical Computer Science, 2006. **152**: p. 3-22.
14. Heckel, R., *Graph transformation in a nutshell.* Electronic notes in theoretical computer science, 2006. **148**(1): p. 187-198.
15. Taentzer, G. *AGG: A graph transformation environment for modeling and validation of software*. in *International Workshop on Applications of Graph Transformations with Industrial Relevance*. 2003. Springer.
16. König, B., et al., *A tutorial on graph transformation*, in *Graph Transformation, Specifications, and Nets*. 2018, Springer. p. 83-104.
17. Machado, R., L. Ribeiro, and R. Heckel, *Rule-based transformation of graph rewriting rules: towards higher-order graph grammars.* Theoretical Computer Science, 2015. **594**: p. 1-23.
18. Khan, T.A., O. Runge, and R. Heckel. *Testing against visual contracts: Model-based coverage*. in *International Conference on Graph Transformation*. 2012. Springer.
19. Runge, O., T.A. Khan, and R. Heckel, *Test case generation using visual contracts.* Electronic Communications of the EASST, 2013. **58**.
20. de Bruijn, V., *Model-Based Testing with Graph Grammars.* 2013.
21. Kalaee, A. and V. Rafe, *Model-based Test Suite Generation for Graph Transformation System Using Model Simulation and Search-based Techniques.* Information and Software Technology, 2018.
22. Haupt, R.L. and S. Ellen Haupt, *Practical genetic algorithms.* 2004.
23. Eberhart, R. and J. Kennedy. *Particle swarm optimization*. in *Proceedings of the IEEE international conference on neural networks*. 1995. Citeseer.
24. Dias Neto, A.C., et al. *A survey on model-based testing approaches: a systematic review*. in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. 2007. ACM.
25. Jonsson, M.B.B., J.-P.K.M. Leucker, and A. Pretschner, *Model-based testing of reactive systems*. 2005, Berlin, Germany, Springer.
26. Rensink, A. *The GROOVE simulator: A tool for state space generation*. in *International Workshop on Applications of Graph Transformations with Industrial Relevance*. 2003. Springer.
27. Engels, G., B. Güldali, and M. Lohmann. *Towards model-driven unit testing*. in *International Conference on Model Driven Engineering Languages and Systems*. 2006. Springer.
28. Rafe, V., *Scenario-driven analysis of systems specified through graph transformations.* Journal of Visual Languages & Computing, 2013. **24**(2): p. 136-145.
29. Ehrig, H., G. Rozenberg, and H.-J. rg Kreowski, *Handbook of graph grammars and computing by graph transformation*. Vol. 3. 1999: world Scientific.
30. Ehrig, H., U. Prange, and G. Taentzer. *Fundamental theory for typed attributed graph transformation*. in *International conference on graph transformation*. 2004. Springer.
31. Heckel, R., T.A. Khan, and R. Machado, *Towards test coverage criteria for visual contracts.* Electronic Communications of the EASST, 2011. **41**.
32. Liu, S. *Validating formal specifications using testing-based specification animation*. in *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*. 2016. ACM.
33. Böhmer, K. and S. Rinderle-Ma, *A systematic literature review on process model testing: Approaches, challenges, and research directions.* arXiv preprint arXiv:1509.04076, 2015.
34. Anand, S., et al., *An orchestrated survey of methodologies for automated software test case generation.* Journal of Systems and Software, 2013. **86**(8): p. 1978-2001.
35. Aichernig, B.K., F. Lorber, and D. Nickovic, *Model-based mutation testing with timed automata.* Graz University of Technology, Graz, 2013.
36. Satpathy, M., et al. *Automatic testing from formal specifications*. in *International Conference on Tests and Proofs*. 2007. Springer.
37. Louzaoui, K. and K. Benlhachmi, *A Robustness Testing Approach for an Object Oriented Model.* JCP, 2017. **12**(4): p. 335-353.

38. Harman, M., Y. Jia, and Y. Zhang. *Achievements, open problems and challenges for search based software testing*. in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015. IEEE.
39. Khari, M. and P. Kumar, *An extensive evaluation of search-based software testing: a review.* Soft Computing, 2017: p. 1-14.
40. Fraser, G. and A. Arcuri, *Whole test suite generation.* IEEE Transactions on Software Engineering, 2013. **39**(2): p. 276-291.
41. Fraser, G., A. Arcuri, and P. McMinn, *A memetic algorithm for whole test suite generation.* Journal of Systems and Software, 2015. **103**: p. 311-327.
42. Gönczy, L., R. Heckel, and D. Varró, *Model-based testing of service infrastructure components*, in *Testing of software and communicating systems*. 2007, Springer. p. 155-170.
43. Heckel, R. and L. Mariani. *Component Integration Testing by Graph Transformations*. in *International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications, Cairo*. 2004. Citeseer.
44. Whitley, D., V.S. Gordon, and K. Mathias. *Lamarckian evolution, the Baldwin effect and function optimization*. in *International Conference on Parallel Problem Solving from Nature*. 1994. Springer.
45. Grechanik, M. and G. Devanla. *Mutation integration testing*. in *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. 2016. IEEE.
46. Offutt, A.J., *Investigations of the software testing coupling effect.* ACM Transactions on Software Engineering and Methodology (TOSEM), 1992. **1**(1): p. 5-20.
47. Belli, F., et al., *Model-based mutation testing—approach and case studies.* Science of Computer Programming, 2016. **120**: p. 25-48.
48. Ma, Y.-S., J. Offutt, and Y.-R. Kwon. *MuJava: a mutation system for Java*. in *Proceedings of the 28th international conference on Software engineering*. 2006. ACM.
49. Schmidt, A. and D. Varró. *CheckVML: A tool for model checking visual modeling languages*. in *International Conference on the Unified Modeling Language*. 2003. Springer.
50. Lobo, F., C.F. Lima, and Z. Michalewicz, *Parameter setting in evolutionary algorithms*. Vol. 54. 2007: Springer Science & Business Media.
51. Lobo, F.G. and C.F. Lima, *Adaptive population sizing schemes in genetic algorithms*, in *Parameter setting in evolutionary algorithms*. 2007, Springer. p. 185-204.
52. Arqub, O.A., *Adaptation of reproducing kernel algorithm for solving fuzzy Fredholm–Volterra integrodifferential equations.* Neural Computing and Applications, 2017. **28**(7): p. 1591-1610.
53. Arqub, O.A., et al., *Numerical solutions of fuzzy differential equations using reproducing kernel Hilbert space method.* Soft Computing, 2016. **20**(8): p. 3283-3302.
54. Arqub, O.A., et al., *Application of reproducing kernel algorithm for solving second-order, two-point fuzzy boundary value problems.* Soft Computing, 2017. **21**(23): p. 7191-7206.