



City Research Online

City St George's, University of London

Citation: Ghasemi, S., Rafe, V., Bahrampour, A. & Heckel, R. (2025). Whole test suite generation from graph transformation specifications using ant colony optimization. *Software Quality Journal*, 33(1), 10. doi: 10.1007/s11219-024-09702-3

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/34634/>

Link to published version: <https://doi.org/10.1007/s11219-024-09702-3>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).

Whole Test Suite Generation from Graph Transformation Specifications using Ant Colony Optimization

Simin Ghasemi^{1,*}, Vahid Rafe², Anvar Bahrapour³, Reiko Heckel⁴

¹ Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran
simin-ghasemi@pnu.ac.ir

² Centre for Software Reliability, Department of Computer Engineering, City, University of London, EC1V 0HB, UK
vahid.rafe@city.ac.uk

³ Department of Computer Engineering, Sanandaj Branch, Islamic Azad University, Sanandaj, Iran
bahrapour@iausdj.ac.ir

⁴ School of Computing and Mathematical Sciences, University of Leicester, UK
rh122@le.ac.uk

*Corresponding author
simin-ghasemi@pnu.ac.ir

Abstract. Model-based testing is an automated process to generate tests from behavioral models of a system under test. Model checking is a verification technique to prove/falsify properties through exploring a state system's space. In the literature, model-based testing often uses model checking to generate execution paths as test cases. However, due to state space explosion, exploring the whole state space may not be possible. Recently, methods based on meta-heuristics have been proposed to cope with this challenge, including evolutionary approaches. In these methods, a tolerable portion of the state space is explored heuristically, optimizing the generated paths to cover the test objectives. Generally, these methods result in large test suites that are hard to evaluate manually and less useful in practice.

In this paper, a novel method based on the ant colony optimization is proposed for systems specified through graph transformations. Our approach generates and evaluates the test suite as a whole, aiming to cover test objectives along test paths as soon as possible. This is the first method specifically designed for model-based whole test suite generation, offering a fresh perspective on optimizing test coverage in complex systems. The method is implemented in GROOVE, an open-source toolset for designing and model checking graph transformation systems. Experimental results on well-known case studies show that we generate smaller test suites with better coverage while the speed of convergence is significantly improved.

Keywords: Model Checking-based Testing, Whole Test Suite Generation, Graph Transformation Systems, Ant Colony Optimization.

1 Introduction

Software quality is one of the most challenging aspects of software development process. While software testing is one of the most common methods for identifying system defects, it is often expensive and time-consuming [1], [2]. Models, which are abstract representations of systems, offer a more cost-effective means of analyzing software. In software testing, models play a crucial role: they can describe the system under test (SUT), generate test suites (TS) from execution paths, and serve as test oracles. Model-Based Testing (MBT) is a gray-box technique that automatically generates test cases (TCs) from models [3]. The key principles of MBT include modeling tenets for test generation, reusability of modeled requirements, test selection criteria, test generation strategies, and the transformation of abstract tests into concrete, executable tests. Among these, achieving full coverage of test objectives is the most critical aspect.

Model Checking-based Testing (MCT) is a type of MBT that uses model checking [4] to generate tests from counterexamples, i.e., specific paths demonstrating the violation of certain properties. However, exploring the state space to meet test objectives often encounters the problem of state-space explosion. This can result in the state space generation and exploration to be incomplete so that test objectives may remain uncovered. To be applicable to real and large-scale models, MCT approaches limit the costs, leading to tradeoffs with their aims of covering test objectives by valid test cases and limiting redundancy arising from duplicate or equivalent test cases.

To mitigate the issue of state-space explosion and uncovered test objectives, evolutionary algorithms such as Genetic Algorithm (GA) [5], Particle Swarm Optimization (PSO) [6], Bayesian Optimization [7] and Non-dominated Sorting Genetic Algorithm (NSGA-II) [8] have been proposed for test case generation. These approaches explore the state-space heuristically, producing test data with some level of coverage. Although these approaches have smoothed the state space explosion problem, they often become trapped in local optima, resulting in large test suites with uncovered test requirements. Search operators in the mentioned approaches over candidate test paths usually leads to invalid execution paths, and they are fixed randomly according to path exploration logic in the state space. This randomly behavior leads to imbalanced

tradeoff between exploration and exploitation of the search approach. In other words, invalid resulting test paths of search operators disturb the convergence, and exploitation of the search process and this leads to lengthy tests with less test objectives covered.

For example, consider the state space graph and two sample test paths depicted in red and green in Fig. 1(a). In the state space graph, each node represents a state of the SUT, and transitions represent operations applied to that state. If a GA-based approach like the one referenced in [5] performs a crossover on the two test cases from the second position (as shown in Fig. 1a), the resulting child test cases, shown in Fig. 1(c), will be invalid because state S_{11} is not reachable from state S_5 (transition $t_{6,1}$ cannot be applied to S_5). Similarly, mutating the first test case could result in an invalid test case by changing $t_{1,1}$ to $t_{3,1}$. As a result, these operators tend to search the state space blindly, often generating infeasible test cases and test suites. Approaches that rely on heuristics, such as GA and NSGA-II, explore the state space in a semi-random, haphazard manner, leading to inefficiencies.

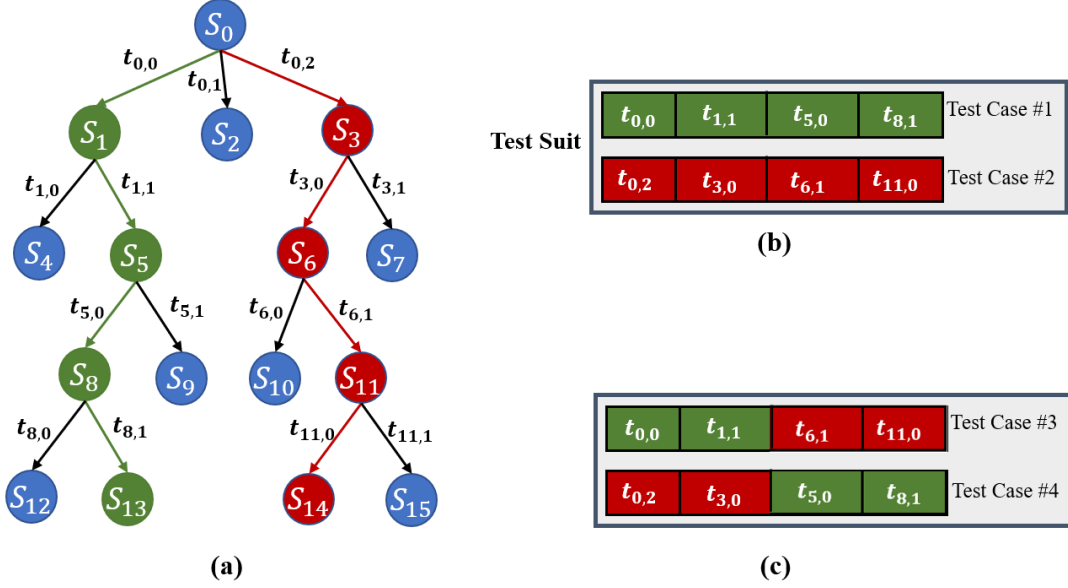


Fig. 1: a) A portion of an example state space graph, b) Structure of a Test Suite and c) Invalid test cases produced by crossover operator

To address the issue of redundancy and test cost management in MCT, some researchers employ reduction procedures to minimize the size of the test suite [6] or eliminate redundant test cases. Although these procedures can mitigate the redundancy problem, they can be costly and may not always yield the desired outcome. To understand this problem, it should be noted that most approaches in the literature generate test cases individually and then combine them into a single test suite. This can make it difficult to predict and control the size of the resulting test suites, and leads to coverage redundancy if a single test objective is covered by multiple test cases. Moreover, multiple test cases may be required to cover all possible combinations of inputs, actions, or scenarios necessary to ensure full coverage of the objectives. To address these challenges, [9] suggests generating and evaluating the test suite as a whole. This approach is utilized in [5] using a genetic algorithm with test models specified through Graph Transformation Systems (GTS). However, as mentioned above, the crossover and mutation operators in this approach often disrupt execution paths, leading to blind recombination or mutation of tests without effectively reducing their length. As a result, in approaches like [5] not originally designed for whole test suite generation but individual test cases, it is necessary to perform additional procedures to verify test case validity and to reduce the size of the test suite.

While the approaches in the literature have partially addressed the aforementioned problems, there is no comprehensive approach that reduces state space explosion, covers most test objectives and maintains an acceptable test size, while limiting the costs of test case generation even for large models. In this paper, we propose a heuristic approach to generate efficient test suites using Ant Colony Optimization (ACO) [10] and whole test suite generation. The ACO metaheuristic, inspired by the foraging behavior of real ants, is capable of finding near-optimal solutions with acceptable resource consumption. We utilize ACO to generate the entire test suite rather than individual test cases. In this approach, each ant generates all the

test cases of a test suite simultaneously, computing their combined coverage score, and aiming to cover the test objectives as early as possible along their execution paths. Despite being GA-based, this allows us to consider only reachable states, avoiding the generation of infeasible test cases and suites. **To be more precise, we employ the ACO metaheuristic to explore the state space and cover a higher proportion of test objectives than in earlier work. We use ACO to generate whole test suites, thus eliminate the redundancy problem. This combination results in an effective test generation procedure which achieves a high coverage score while maintaining a small test suite size and so lowering costs. Our approach also converges rapidly to the maximum coverage score.**

The main contributions of this paper are as follows:

- A revised ACO algorithm is developed at model level to generate test suites for systems specified through graph transformation systems.
- A novel local search strategy is proposed to cover test objectives in a more efficient way.
- The first approach specifically designed for model-based whole test suite generation.

The rest of this paper is organized as follows. Section 2 describes the basic concepts of GTS and MBT, as well as ACO. Section 3 introduces the state of the art in MBT and reviews existing work on testing based on GTS. Section 4 describes our approach in detail. The experimental evaluation is presented in Section 5. Finally, section 6 concludes the paper and suggests future work.

2 Background

In this section, we briefly survey the basic concepts of the proposed approach.

2.1 Graph Transformation Systems

Graphs are a natural way to describe and model complex systems. The components can be described by nodes and the interactions can be represented by edges. Graph transformation system is a graph-based formalism that allows us to model both static and dynamic aspects of complex systems. Graph transformation can be used in various domains of computer science, including software engineering and the development of concurrent and distributed systems [11]. The fundamental features of the GTS formalism are introduced in [12]. In GTS, rules are used to specify the behavior of the System Under Consideration (SUC), and each state graph represents a configuration of the system. The initial state is described by a host graph. Graph elements (nodes and edges) and their variables in state graphs can be updated through the application of rules. More formally:

Definition 1: (Rule) A rule is given as $R = (LHS, RHS, NAC)$ where LHS is the Left-Hand Side, RHS is the Right-Hand Side, and NAC is a Negative Application Condition. LHS and NAC define the pre-conditions of the rule application, and RHS describes its post-conditions.

Definition 2: (GTS) A GTS is defined as $GTS = (TG, HG, R)$ where TG is a type graph that describes the types of node and edge types and their relations, HG is a host graph that represents the initial state of the system and is typed over TG , and R is a set of transformation rules.

In this section, we introduce a Hotel Management System (HMS) that will serve as a running example throughout the rest of the paper. The HMS was initially introduced in [13] and we have made some modifications to the original specification to align it with our proposed approach. In the HMS, a hotel consists of several rooms and registered guests. Each guest can book any available room, with the corresponding bill automatically generated. In our example, we assume that each guest can book a maximum of one room. When a guest wishes to check out of a booked room, they must first settle the bill before completing the checkout process.

Figure 2 shows an initial host graph representing the HMS in the notation of the open-source GROOVE graph transformation tool [14]. In this host graph, attributes associated with each node describe the characteristics of the respective object. Figure 3 illustrates the rules of the HMS, which include *BookRoom*, *CheckOut*, *ClearBill*, *UpdateBill*, and *OccupyRoom*.

In the GROOVE tool, a solid black line denotes Readers, blue dashed or double-bordered lines elements symbolize Erasers, green solid lines represent Creators, and the NAC elements are indicated by red double-bordered/dashed lines. Eraser elements delete the object or relation after the application of the rule, while NACs represent forbidden structures that must not be present for a rule to apply.

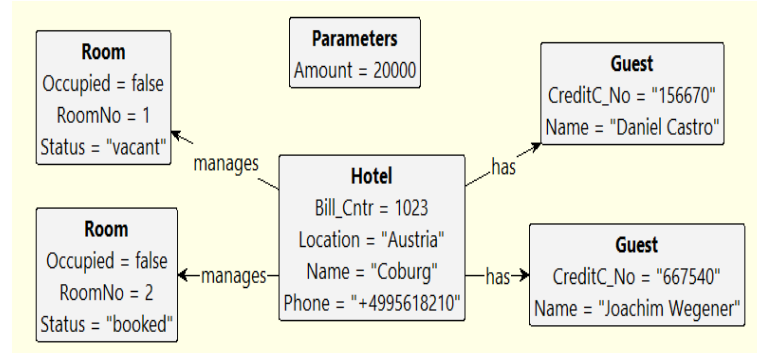


Fig. 2: A simple host graph for HMS in GROOVE toolset

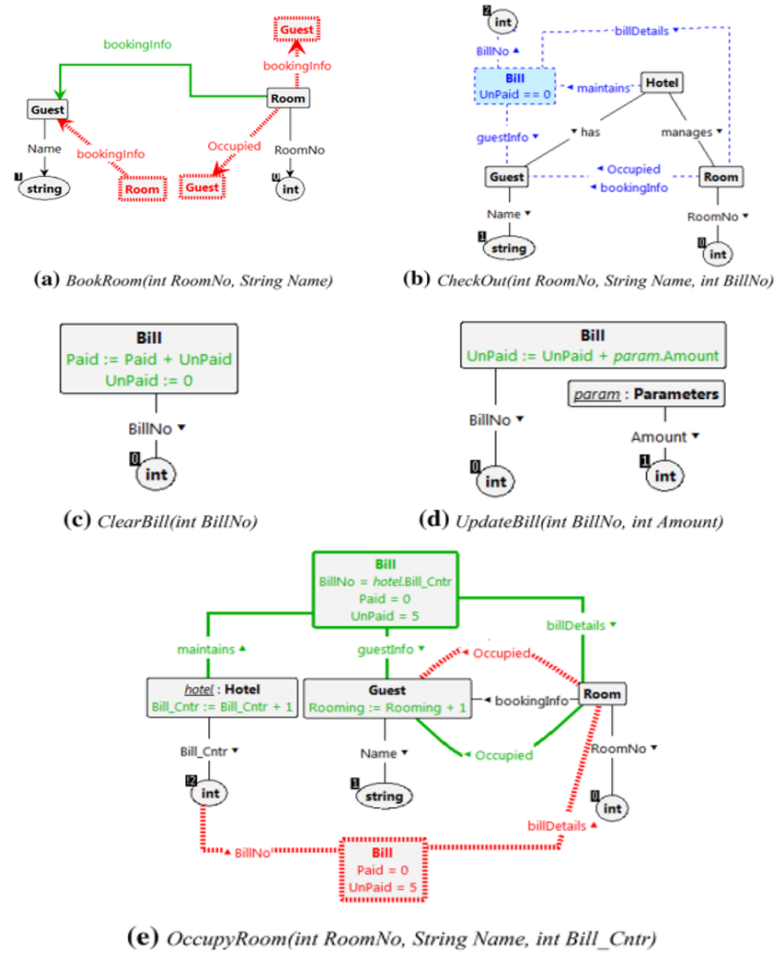


Fig. 3: Production rules of the HMS example in the GROOVE toolset [5]

The behavior of a GTS is represented by a Labeled Transition System (LTS). An LTS is a directed graph in which nodes represent states and edges represent the transitions, formally:

Definition 3: (LTS) A Labeled Transition System is defined as $LTS = (S, Act, \rightarrow, I, AP, L)$ where:

- S is a set of states.
- Act is a set of actions.
- $\rightarrow$ is a subset of $S \times Act \times S$ known as the transition relation.
- I is a subset of S as the set of initial states.
- AP is a set of atomic propositions.
- L is a labeling function.

Fig. 4(a) shows a portion of the state space or LTS for the running example generated by GROOVE and Fig. 4(b) shows an execution path through the LTS. In this path, starting from the initial state, a sequence

of rules has been selected and applied. According to this scenario, room 1 is booked by *J W*, then it is occupied, and a bill 1023 is created. After the bill is updated and cleared through the corresponding rule applications, the guest successfully checks out.

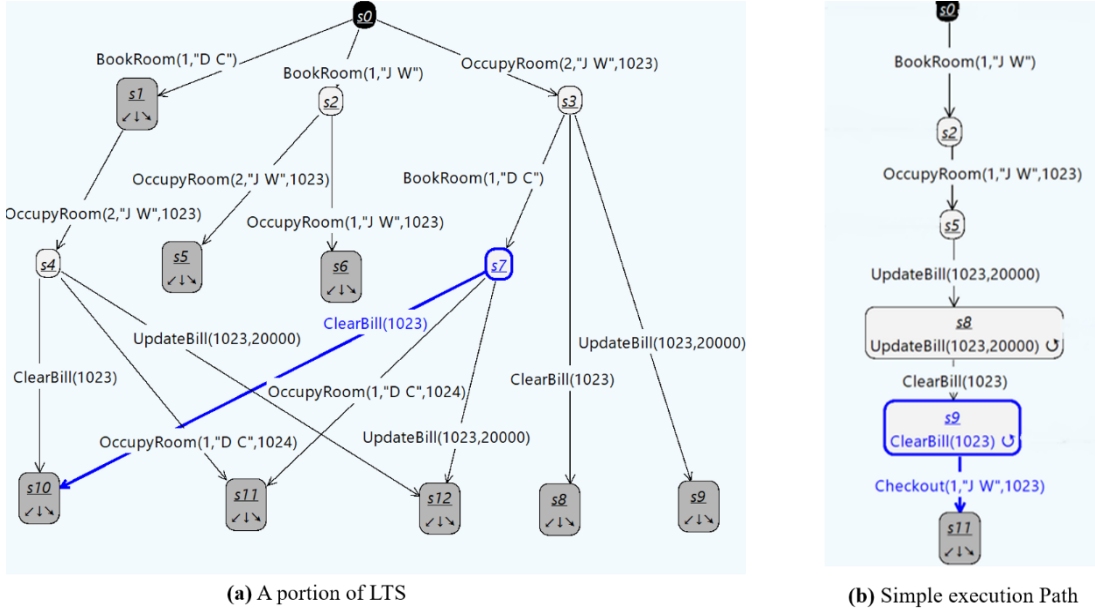


Fig. 4: A portion of state space graph of the HMS example generated by GROOVE.

2.2 Testing Graph Transformation Systems

In software engineering, a model is an abstract description of a system, usually focusing on its conceptual and functional aspects rather than specific implementation details. This description can be helpful to analyze and test the system before committing resources into its actual implementation. In MBT, the model describes the dedicated behavior of the SUT, and test cases are generated from the model [15]. Model-checking [4] is a powerful technique that can be used to automatically generate test cases from model by exploring the state space to identify defects. In the MCT, a model of the considered system and a set of test objectives, expressed as desired reachability properties, are provided to the model checker. The model checker then generates counterexamples or witnesses that either violate or satisfy these properties, respectively. A counterexample or witness is an executable path that starts from the initial state and reaches a final state or a state that satisfies or refutes the property. In MCT, these paths are interpreted as test cases [11]. However, this approach has several drawbacks, the most significant being state space explosion and test case redundancy.

In GTS, the behavior of the system can be represented by LTS. This allows the use of MCT by deriving test cases from the sequences of labels of paths in the LTS [16]. In this section, we detail the structure of test cases and evaluation mechanisms.

2.2.1 The Structure of Test Suite in LTS

In testing GTS specifications, a test case is described as a finite path within the state space that begins at the initial host graph and traverses through the LTS, depending on the choices of rules and matches applicable in each state graph. As previously mentioned, the nodes in the state space represent reachable state graphs, and their outgoing transitions are determined by the application of rules at that state. Consequently, a test case is defined by a sequence of transitions.

Fig. 1 shows a portion of an example state space and two instance test cases or executable paths. According to Fig. 1(a), for an internal state like S_3 , there are two transitions $t_{3,0}$ and $t_{3,1}$. The first number of the transition denotes the source state index and the second one is the unique number of the selected transition. Two highlighted paths with red and green colors show instance paths that are considered as test cases. Indeed, the sequence of the selected transitions from the initial state is considered as a test case, and a test suite is a set of test cases. In Fig. 1(b), sample test cases and test suites have been shown respectively.

2.2.2 Test Case Evaluation

In software testing, there are different criteria to evaluate test suites and related test cases. More specifically, in MBT approaches specified through GTS, there are several coverage criteria such as data-dependency coverage, state coverage, transition coverage, path coverage and rule coverage to guide the test generation process [5]. Data-dependency coverage criteria in testing GTS specifications are widely used in the literature.

Data dependency coverage analyses dependencies between rules. A data dependency in a GTS is a condition where one rule creates or updates an entity (creates a node or edge or updates an attribute of a node) and another rule uses (reads or deletes) it. In such a relationship, the first rule of the relationship provides a prerequisite to the second rule, enabling its application, so this relation is referred to as enabler relation. The following definition formally describes this relation.

Definition 4: (Rule Dependency) Given two production rules p_i and p_j , we say p_j is dependent on p_i if there are transformation steps $t_i = (G \ p_i \Rightarrow G_i)$ and $t_j = (G_i \ p_j \Rightarrow G_j)$ such that t_i enables t_j .

Different types of data dependencies have been defined in [13], which are used as coverage criteria in the testing literature. These relationships include *Create_Read* (C_R), *Create_Delete* (C_D), *Create_Update* (C_U), *Delete_Nac* (D_N), *Update_Read* (U_R) and *Update_Update* (U_U). As the names denote, in C_R a rule creates an object (edge, node or attribute) that another rule reads, in C_D a rule creates an object which another rule deletes, and so on. Since the update procedure of an attribute includes reading the value of the attribute, deleting it, and finally creating the new value, U_R and U_U dependencies can be considered as C_R and C_U respectively. Therefore, in this research we consider U_R and U_U relationships analogous to C_R and C_U.

To obtain test objectives for a given model, there are static and dynamic approaches [5]. In a static approach such as [6], the relations between rules will be extracted from the rule structure without considering their actual parameters or exploring the state space. By analyzing how rules could overlap in the pre and postconditions, potentially dependent pairs will be obtained as test objectives to be covered by the test suite. Static approach can lead to false positive and false negative test objectives [5]. Instead, using the state space provides more accurate rule relations based on the states and transitions that are actually reachable in the system, leads to valid test objectives. In dynamic approach, analyzing and recognizing the test objectives will be done through exploring the current state space. We use the following definition for Test Objectives (TO):

$$TO = \{ (r_i, r_j) \mid r_i \text{ enables } r_j \text{ through dynamic } \{C_R, C_D, C_U, D_N\} \text{ dependencies} \}$$

Table 1: Data dependency relations in HMS example extracted by Definition 4

Relation type	Dependent rules	Related entity
C_R	(BookRoom, OccupyRoom)	bookingInfo
C_D	(BookRoom, CheckOut)	bookingInfo
	(OccupyRoom, CheckOut)	Occupied
	(OccupyRoom, CheckOut)	billDetails
	(OccupyRoom, CheckOut)	maintains
	(OccupyRoom, CheckOut)	guestInfo
	(OccupyRoom, CheckOut)	Bill
C_U	(OccupyRoom, ClearBill)	Bill
	(OccupyRoom, UpdateBill)	Bill
D_N	(Checkout, BookRoom)	bookingInfo
	(Checkout, BookRoom)	Occupied

Table 1 lists all possible (i.e., static) data-dependencies in the HMS example according to Definition 4. For example, when a guest books a vacant room using *BookRoom* rule, the *bookingInfo* edge will be created between the guest and the room. Then, in the *OccupyRoom* rule, *bookingInfo* edge is read to ensure the booked room for the desired guest. So, there is a C_R dependency between *BookRoom* and *OccupyRoom*.

2.3 Ant Colony Optimization

Ant Colony Optimization [10] is a probabilistic technique for solving computational problems which can be reduced to finding suitable paths through graphs. ACO is a multi-agent method inspired by the behavior of real ants for the detection of shortest paths to find food resources. Real ants lay down pheromones directing each other to resources while exploring their environment. In ACO artificial ants find optimal solutions by moving through a parameter space representing all possible solutions. ACO is a form of swarm intelligence and the optimal solution will be obtained by the collaboration between agents.

In general, an ACO problem [10] consists of the three procedures *ConstructionAntSolutions*, *UpdatePheromone* and *DaemonActions*. These procedures are executed until a given stopping criterion is satisfied, such as reaching a given number of iterations or a final or desired state. In the first procedure, a set of fresh artificial ants (i.e., colony) seek a path between their colony and a source of food in a so-called construction graph that provides all solutions to the problem by some probability. The construction graph G includes a set of edges L which connect nodes C with each other. Each edge between nodes i and j involves the pheromone amount $\tau(i, j)$ and the heuristic value $\eta(i, j)$ which are two factors helping the ants to select the next node in the construction phase. The heuristic value can be the ant's prediction of this selection according to a heuristic algorithm or prior experiences. Therefore, ant k in node i will select node j with respect to the probability function presented in Equation 1, in which N_i is a set of next nodes of i , α and β are two parameters of the algorithm determining the relative effect of pheromones and heuristic values on the path construction, respectively [17].

$$P_k(i, j) = \frac{[\tau(i, j)]^\alpha \cdot [\eta(i, j)]^\beta}{\sum_{l \in N_i^k} [\tau(i, l)]^\alpha \cdot [\eta(i, l)]^\beta} \quad (1)$$

Using this Equation, each ant randomly selects one of the neighboring nodes according to their probabilities, and moves on to that node in the construction graph. This process is repeated until a desired solution is built.

In the second phase, the *UpdatePheromone* procedure, the pheromone trails on different paths are updated using Equation 2. In this Equation, ρ indicates a pheromone evaporation rate between 0 and 1 that determines the fade rate of the previous pheromone on the path. The m value defines the colony size. The amount of pheromone spread by each ant on an edge is specified by $\Delta\tau_k(i, j)$ and calculated by Equation 3, where the length of the traversed path by ant k is indicated by L_k . Q is the pheromone value for ant A_k which is divided by the length L_k to calculate the average pheromone.

$$\tau(i, j) = \rho \cdot \tau(i, j) + \sum_{k=1}^m \Delta\tau_k(i, j) \quad (2)$$

$$\Delta\tau_k(i, j) = \frac{Q}{L_k(t)} \quad (3)$$

The third procedure, *DaemonAction*, is an optional procedure to implement specific and centralized actions, which cannot be performed by individual ants. For example, a local search algorithm can be used in this phase to improve the solution for every ant.

3 Related Work

In the realm of model-based testing, significant improvements have been made, such as early and cost-effective testing; however, several challenges and research gaps persist. As mentioned in the Introduction, state space explosion is one of the most notable challenges. Due to limitations in computational resources, this issue restricts the scalability of existing methods for real-world and complex systems [18]. Effective cost management in the testing process could help mitigate this challenge. Furthermore, while various algorithms have been employed to enhance test coverage, there remains a lack of robust mechanisms to address most test objectives and ensure the validity and reliability of the generated test cases.

In this paper, we propose a model-based testing approach that employs state space generation based on GTS models to support a meta-heuristic method generating test suites holistically. To study the related work relevant to our proposed approach, we conducted a comprehensive and multilateral review of the literature. Although there is significant research in this field, only a few approaches are comparable to our proposal. We selected the approaches we found most relevant and articulated their differences from ours, reviewing their key contributions, describing their methodologies, results and limitations to provide a clearer understanding of the current literature.

We begin with brief overview of the most relevant work that will be compared with our proposed method in the evaluation section. After that, we will explore these references alongside other related studies in more

detail in the following subsections. We categorized the work in the subsections into those that utilized whole test suite generation, those employing an MCT approach using GTS-based models, and those using on search and optimization techniques.

In [6], the authors propose a hybrid search-based algorithm using GA and PSO, referred to as HGAPSO, for generating test cases from models defined using GTS. They use static test objectives and incorporate a minimization phase. This work is one of the selected studies for comparison with our proposed approach. Another selected work is [5] in which the authors proposed an automated test generation approach using GA for testing software models specified through GTS. They computed test objectives dynamically and also employed a reduction phase for test case minimization. The work in [8] is another selected approach for comparison; although it does not utilize GTS, it is still considered model-based testing. The authors in [8] propose a hyper-heuristic approach for test case generation called ISR-MCF, which implements three search operators: NSGA-II-LLH, SPEA2-LLH, and PSO-LLH. It uses reinforcement learning to evaluate the output of each operator and then selects the most appropriate operator to generate the test case.

3.1 Whole Test Suite Generation

Most of the proposals in the literature produce test cases individually, and after completing the test case generation, they are placed in a test suite. This approach often results in repetitively covered test objectives, redundant test cases and an ineffective test suite. For this reason, the entire test suite has been produced as a whole in [9] with the aim of covering all test objectives at the same time while keeping the total size as small as possible. Whole test suite generation aims to generate a set of test cases to cover common objectives and prevent redundant test cases. This eliminates the need to reduce the size of the test suite after its generation. The authors in [9] used whole test suite generation for code-based testing and based on branch-coverage criteria. Their experiments were validated using code from open-source libraries and an industrial case study. The results show that their method achieves higher coverage scores and smaller test suites. Fraser and Arcuri in [19] proposed two novel approaches to avoid redundant test executions by monitoring state infection conditions. They subsequently use whole test suite generation to optimize test suites. Their experiments show that their approach scales well, and their proposed approach makes mutation testing an option to achieve the same or similar code coverage but a much higher mutation score.

Some work in the literature uses whole test suite generation alongside search-based approaches to generate effective tests. For instance, the authors in [20] introduced two novel multi-criteria and combinatorial artificial bee colony algorithms that utilize mutation and crossover operators to create a test suite aimed at maximizing a fitness function that integrates multiple objectives for object-oriented software. This method employs the whole test suite strategy by iteratively generating individual test cases. This helps to ensure that the process is unaffected by the ordering and complexity of the coverage requirements, thereby minimizing the risk of unintended coverage overlaps.

In [21] the authors extend the GA used in the whole test suite generation approach to a memetic algorithm. They produce a complete test suite which maximizes coverage goals while minimizing its size. In software testing, mutation testing has been suggested as a robust alternative. However, the process of generating tests based on mutation is often hindered by too many mutants that are either easily eliminated or equivalent to one another. Consequently, any effort spent on test generation for these mutants is essentially futile. To address this issue, the authors in [22] propose search-based EvoSuite test generation. This prevents unnecessary test executions on mutants by tracking state infection conditions and utilizing whole test suite generation to enhance test suites for maximizing the number of mutants killed, rather than focusing on individual mutants.

The aforementioned work uses whole test suite generation in code-based testing. To the best of our knowledge, the only work in MBT using the whole test suite paradigm is [23]. The authors in [23] employ a cost-aware memetic algorithm to optimize coverage score and testing costs. They consider a test suite as a chromosome to treat all test cases at once and evaluate them holistically by a fitness function. Although using whole test suite generation, it requires significant computational resources for first generating new test cases and then minimize the test suite in an additional reduction phase. Furthermore, the use of GA can lead to invalid test cases requiring extra validation.

3.2 Model Checking-based Testing using Graph Transformation System

Model checking-based testing is a type of MBT that uses the principles of model checking [4], [24] for software testing. This technique involves the automatic generation and verification of test cases based on formal models of the SUT. By systematically exploring the state space of a model, model checking can identify potential errors and ensure that the system adheres to its specifications. Indeed, a model checker

generates a counterexample/witness to violate/satisfy the reachability property. This witness can be considered as a test case [11]. One of the primary works in the literature that has utilized model-checkers for test generation is [25]. It explores the application of model checking techniques to automate the software testing process, discusses how model checking can be employed to identify and rectify errors in software systems, highlighting the advantages of this approach in enhancing the accuracy and efficiency of software testing.

The authors in [26], provide an extensive overview of the state-of-the-art in model checking and its applications, including testing. They state the importance of integrating model checking with testing to address state space explosion [27] and to enhance the reliability of software systems. The use of model checking for software testing encounters various challenges arisen from the fact that model checkers were not originally designed for this purpose [18]. The primary disadvantages include concerns about quality, usability, and performance, as well as redundancy and duplicated test cases that address similar testing objectives. The authors of [18] describe these drawbacks and also present potential methods for enhancing test generation in model checking-based testing.

Formal specifications can be used to verify a system’s properties at the design level, thus reducing the risk of implementing an ill-conceived system. Several approaches have emerged for the validation and verification of formal specifications [28], [29], [30]. In this paper, we utilize formal specifications based on GTS. In the rest of this subsection, we discuss work employing GTS in model checking-based testing.

GTS[12] have emerged as a powerful framework for modeling and analyzing complex systems, particularly in the realm of software testing. For instance, the authors in [31] emphasize the importance of understanding the basic concepts of GTS as a foundation for exploring more advanced applications and theoretical developments in the field. They conclude that GTS provide a powerful and flexible approach for modeling problems in computer science and software engineering. Additionally, in [32] the authors aim to provide a comprehensive insight into the mathematically challenging area of graph grammars, which is well-motivated by its many applications in computer science and beyond. The research reported in [33] has laid a foundation for the use of GTS to enhance MBT.

Several publications utilize model checking with GTS formalism for the software testing [5], [6], [7], [23], [34], [35], all based on the GROOVE toolset. GROOVE is an open-source tool designed for editing, visualizing and executing GTS, generating their state space and analyzing them through model checking. By employing this tool, they generate the state space and subsequently define relevant properties to produce counterexamples/witnesses that can either prove or reject these properties. Each path from an initial state to the state space that leads to a desired final state is considered a test case. For instance, in [7], the exploration of the state space is conducted with the goal of identifying promising states. A promising state, in this context, is one where a significant number of test objectives are covered. This approach not only aids in verifying system properties but also enhances the efficiency of finding optimal solutions within complex systems. In [35], the authors apply partial ordering to execution paths and subsequently reduce the obtained sequences to derive proper test cases. Although they do not present experimental results, a closer examination reveals that the approach imposes a high level of overhead and cost on the test generation process, making it less desirable for real-world models. Additionally, they use a reduction phase to extract proper test cases for test oracles, which incurs further costs in the testing process.

In summary, the approaches discussed in this subsection utilize GTS alongside model checkers for software testing, mostly to explore the state space and then, by defining temporal properties, generating paths as test cases to validate or refute them. Our approach shares the use of GTS; however, we employ a whole test suite paradigm, which none of the reviewed works in this subsection except [23] have utilized. The proposed approach in [23] generates the state space from a GTS and then uses whole test suite generation similar to ours. However, our approach generates all test cases simultaneously to optimize both coverage and test suite size at the same time and avoids an extra verification step, making it the first comprehensive approach to whole test suite MBT.

3.3 Search and Optimization-based Testing

While model checker-based methods for testing are quite useful, they encounter several challenges, most prominently state space explosion and the high cost of test generation, which render these methods impractical for complex models. To address these challenges, various search- and optimization-based methods have been employed in the literature, including Bee Colony, Genetic Algorithm, Particle Swarm Optimization, Bat Algorithm (BA), Gravitational Search Algorithm (GS), Bayesian Optimization Algorithm (BOA), and hybrid heuristics [5], [6], [7], [8], [23], [34], [36], [37]. These efforts focus on partially exploring the state space and extracting specific paths to construct test suites that provide compelling coverage.

In [6], a hybrid search-based algorithm using GA and PSO is proposed for generating test cases from models defined using GTS. In this research, test generation is defined as an optimization problem that aims to maximize the coverage of data-dependency test objectives. This method is implemented in the GROOVE toolset. Empirical findings show that the proposed solution effectively generates tests for intricate models and mitigates the state space explosion problem. The authors compute the test objectives statically and assert that all specified test objectives are covered. However, some dynamic test objectives remain unaddressed, making complete coverage the main challenge within this research endeavor. Additionally, they employ a reduction phase to minimize the size of the produced test suite, which imposes further costs on the testing process.

In [23], the authors extend the method of [6] into a cost-aware memetic algorithm to optimize coverage scores and testing costs. Their research introduces a robustness testing approach aimed at identifying vulnerabilities in systems defined via GTS. The effectiveness of this strategy is evaluated using mutation analysis at the model level, with experimental results demonstrating that the proposed testing approach outperforms traditional methods in terms of fault-detection capability. However, since they used data conflicts as a coverage criterion, this work is not directly comparable to our approach. Furthermore, this work generates test cases and test suites using memetic algorithm operators, which can lead to invalid outputs and necessitate additional operations to eliminate them. In [5], the authors propose an automated test generation approach for software models specified through GTS. This approach utilizes a GA to enhance the coverage of rules, states, transitions, and data-flow criteria. They computed the test objectives dynamically and also generated the test suite holistically. Experimental results indicate that data-dependency coverage can outperform other strategies in complex systems with large state spaces and interrelated rule sets. However, this approach has certain limitations, including the significant computational resources required for generating new test cases and the length of the test cases. They employ an additional reduction phase to minimize the generated test suites. Furthermore, the utilization of GA operators leads to some invalid test cases and test suites, which require an extra validation phase. In [7], the authors extend the works of [6] and using BOA instead of HGAPSO. They obtain the test objectives statically and included an extra reduction phase, which makes it unfavorable for large-scale systems.

Several studies in the literature propose MBT alongside heuristic approaches, but they do not use GTS for test suite generation [38], [39]. In the most relevant work [8] the authors propose a hyper-heuristic approach to MBT, called ISR-MCF, which implements three search operators: NSGA-II-LLH, SPEA2-LLH, and PSO-LLH. This approach uses reinforcement learning to evaluate the output of each operator and then selects the most appropriate operator to generate the test case. The implementation demonstrates that ISR-MCF, when combined with NSGA-II-LLH, outperforms the other operators. This research uses several heuristics in a multi-objective environment and can serve as a reference the state of the art in search-based MBT. The granularity of test generation in this work, like that of most other reviewed studies, is at the test case level and thus faces the associated challenges. In [37], the authors proposed an approach that generates test cases for embedded systems using a UML model. They employed heuristics such as divide-and-conquer to produce effective test cases. Although these contributions are not based on GTS, their methods could guide research in our field. For instance, the divide-and-conquer approach can be used to divide the state space and then, alongside other meta-heuristic approaches, could lead to an efficient method.

In summary, the approaches discussed in this subsection utilize search-based or optimization-based methods to heuristically explore the state space. They aim to eliminate redundant paths and cover a high ratio of test objectives with small test suites. However, they suffer from the size of the test suite and the associated costs. Our proposed approach, like these methods, employs a metaheuristic strategy, ACO, to mitigate the state space explosion problem but its use for whole test suite generation results in smaller test suites with lower testing costs.

4 Whole Test Suite Generation using Graph Transformation Systems

In this section, we describe our approach in detail. Most approaches in the literature generate each test case individually. A major drawback of this method is that test objectives covered in different test cases within a test suite may overlap, leading to redundant coverage. The whole test suite approach [9], [19] was developed to address this issue. Instead of focusing on individual test cases, the whole test suite approach evolves the entire suite using a fitness function that considers all test objectives covered by all test cases. Promising results from previous research [9], [19] have demonstrated the effectiveness of this approach in code-level test suite generation.

In this research, we apply the concept of whole test suite generation using the ACO algorithm. Our approach, named WholeAcoT, simultaneously evolves all test cases within a test suite, aiming to cover more test objectives at each step along the corresponding test paths and thereby reduce the overall length of the test suite. This results in an automated approach for generating a whole test suite for systems specified through GTS.

Fig. 5 illustrates the flowchart of the proposed approach. The process begins with the initialization of the required parameters for the Ant Colony Optimization (ACO) algorithm. Afterward, the initial configuration of the ants and the pheromone values are set on an empty construction graph. It should be noted that the construction graph differs from the LTS; instead, it consists of nodes representing sequences of states and edges representing sequences of transitions from the LTS. After initial configuration, the ants then traverse the state space one by one, with the pheromones left behind guiding subsequent ants to make more informed decisions in future iterations.

In this approach, each ant represents a test suite that includes various test paths or test cases, as described in Section 2.2.1. Initially, each ant is positioned at the start state. Depending on the outgoing transitions, it selects a target state along a test path. By employing a whole test suite generation technique, all the test paths within an ant traverse the state space graph simultaneously. The available movement options are determined by the outgoing transition set from their current state. If the next state is a final state, or if the length of the traversed paths exceeds the maximum allowed depth, the movement halts. This process is repeated independently for each ant, ultimately producing a complete test suite.

Once all ants have completed their movement, the remaining pheromone is evaporated according to the specified evaporation rate. The fitness of each ant (test suite) is then calculated based on a data-dependency coverage score, as defined in Section 2.2.2. Using the fitness values, the pheromones along all paths traversed by the ants are updated. Finally, the best-performing ant is selected, and its test suite is stored as the best result thus far. These steps are repeated iteratively until the maximum allowed number of iterations is reached. Each phase of this process is detailed in the following sections.

To illustrate our approach, a running example using the HMS model is explained through this section.

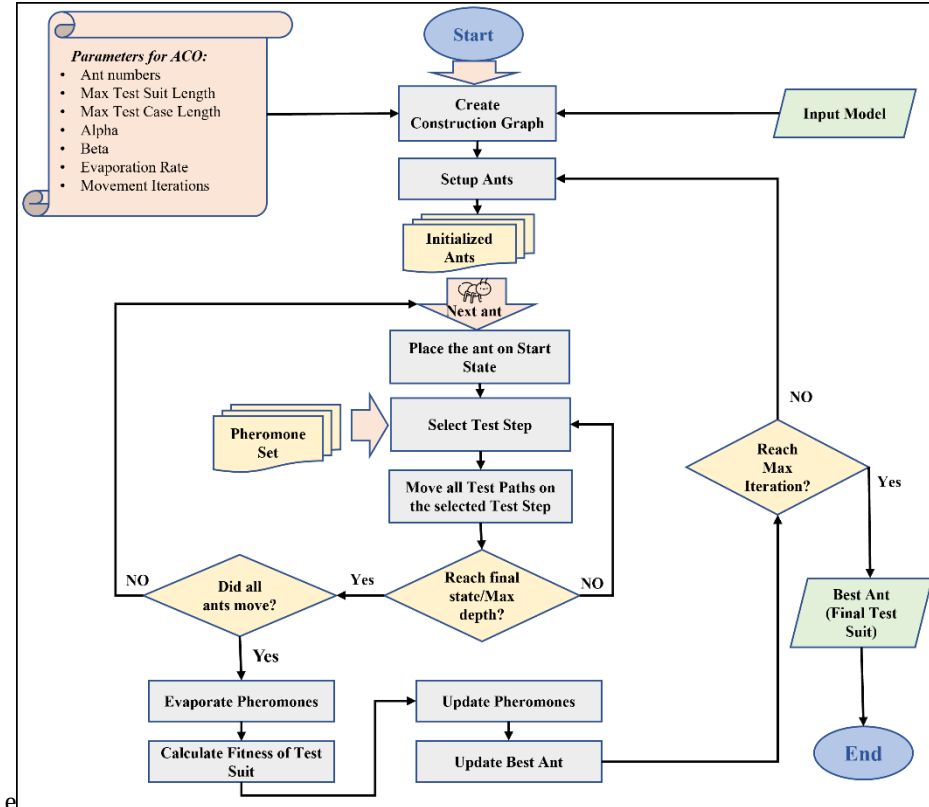


Fig. 5: Flowchart of the proposed approach (WholeAcoT)

Also, the overall structure of WholeAcoT is described in the following pseudo code.

WholeAcoT Algorithm

//Input: A GTS model of the SUT, AntNum, MaxTSLen, MaxTCLen, Alpha, Beta, Evaporation Rate, Movement Iterations.

//Output: A set of Test Paths as a Test Suite

- 1: **Create** an empty construction graph (Super_States, Test_Steps)
 - 2: **For** *iteration* $\leftarrow 1$ to *maxIterations* **do**
 - 3: **Initialize** Ants in the Start_State and construct the first test step TS_0 for each ant; *// set TPs of each ant by random outgoing transitions of S_0*
 - 4: **For each** *ant* **do**
 - 5: **While** *MaxTCLen* not reached **do**
 - 6: **Construct/Recognize** new test steps
 - 7: **Select** test step *TS* according to pheromone and its coverage set
 - 8: **Move to target super state of TS**
 - 9: **Update pheromone** for all ants
 - 10: **Update BestAnt**
 - 11: **Return BestAnt**
-

4.1 Construction graph

To find the optimal solution in ant colony optimization, a construction graph is used to hold the search experience in the search space. In our case, the construction graph shows the test suits traversed by the ant set. Fig. 6 shows a generic construction graph of the proposed WholeAcoT. In this graph, each node represents a Super State (SS) that includes a list of graph states from the LTS. Relation 4 defines a super state of the construction graph formally. The edges of the construction graph show construction steps which we call test steps. Each test step includes a transition per test path of the ant, as shown in the middle path of Fig. 6. Each path of the construction graph indicates a test suite that is traversed by an ant. Therefore, a test suite is also a list of test steps.

To find the optimal solution in ant colony optimization, a construction graph is used to capture the search experience within the search space. In this research, the construction graph represents the test suites traversed by the ant set. Fig. 6 illustrates a generic construction graph for the proposed WholeAcoT. In this graph, each node, referred to as a Super State (SS), contains a list of graph states derived from the LTS. Relation 4 formally defines a Super State within the construction graph. The edges of the construction graph represent construction steps, which we refer to as test steps. Each test step includes a transition corresponding to each test path of the ant, as shown in the middle path of Figure 6. Each path within the construction graph indicates a test suite traversed by an ant. Therefore, a test suite can be understood as a sequence of test steps.

$$SS_i = \{ S_{ij} \mid S_{ij} \text{ is the target state of the } i\text{th transition of the } j\text{th test path of the ant, } j: 1..MaxTSLen \} \quad (4)$$

The construction graph is used to keep pheromone trails and influence random walks of the ants. As shown in the Fig. 6 by dashed red line in test steps, each test step has a specific amount of pheromone. Generally, a path that has more pheromone is better solution. In WholeAcoT, initially, we define an empty construction graph and gradually it will grow as the ants determine new test steps. After all of the ants construct their own solutions, they should update their pheromone trails.

In some paths, test cases may reach a final state before the final step and they will have a shorter length than the *MaxTCLen*. For example, in Fig. 6, state S_{22} is a final state and the related test case is a path with two transitions.

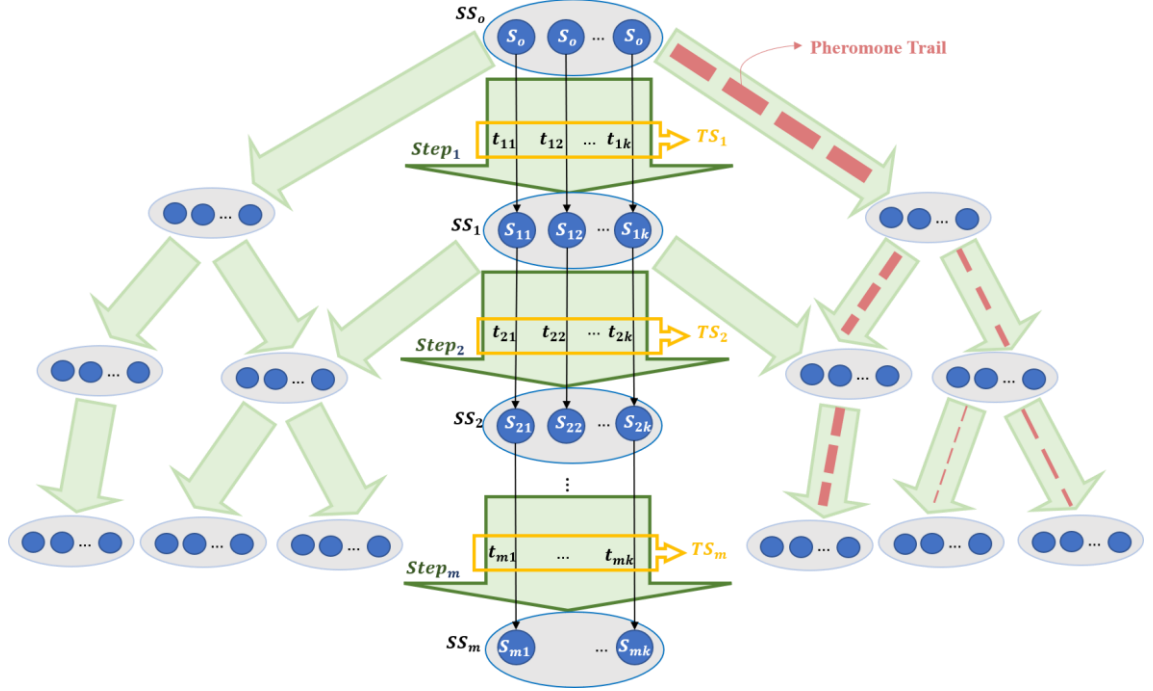


Fig. 6: Construction graph in WholeAcoT ($m=MaxTCLen$ and $TS=TestStep$)

► **Example:** In this section, we continue the HMS example to illustrate the explained phase. According to the WholeAcoT flowchart, the user needs to configure the parameter values. Table 2 represents the selected values, which are kept as small as possible to provide a clear and illustrative case study.

First, the construction graph is initialized by creating SS_0 as the primitive Super State. Since the Maximum Test Suite Length ($MTSLen$) is 3, three instances of S_0 are created, each initiating a separate test path. These initial states represent the host graph shown in Fig. 1. Additionally, two ants are waiting to be initialized and move along the graph. As a result, the construction graph in this phase appears as depicted in Fig. 7.

Table 2: selected values of parameters for HMS example

Parameter	Value	Parameter	Value
Ant Numbers	2	Beta	4
Max Test Suite Length	3	Evaporation Rate	0.4
Max Test Case Length	5	Movement Iterations	30
Alpha	2		

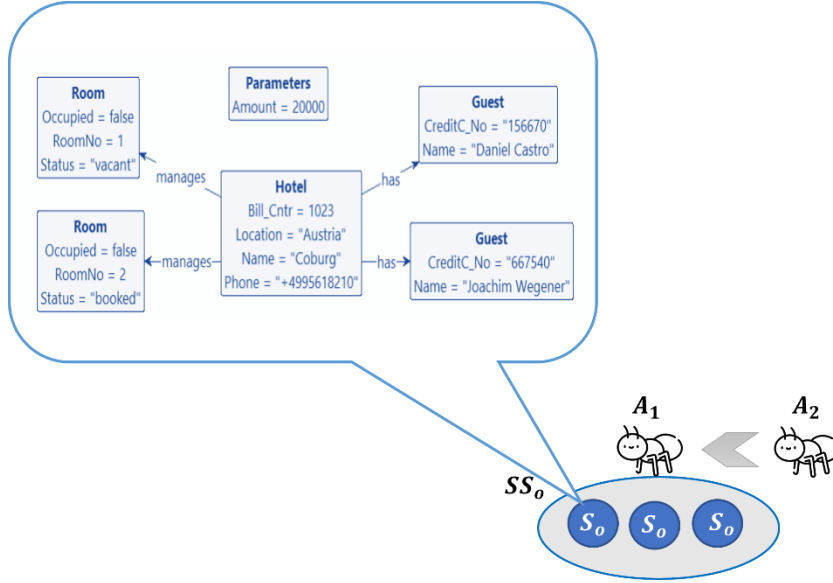


Fig. 7: Initialization of construction graph for the HMS example

4.2 Ants Initialization

According to the WholeAcoT flowchart, the user determines the number of ants and their maximum allowed lengths for test suites and test cases. Since this maximum length is crucial for test effectiveness, it should be selected based on the model under consideration. At first, we initialize the construction graph with a new Super State (SS_0) as the starting point. For each test path of the ant, a new graph state is added to SS_0 , which is then set to the LTS start state. In the initial state, one of the outgoing transitions is randomly selected as the first transition for each test path. These initial transitions form the first test step for the ant. If the construction graph already contains this test step, the ant proceeds to the next Super State; otherwise, the new test step is added to the construction graph before the ant proceeds. With each movement of the ant, another test step is passed. In other words, during an ant's movement, all transitions of the selected test step are applied and added to the corresponding test paths. This movement operator guides the ant to progress simultaneously across all its test paths, enabling the generation of the whole test suite.

To provide further clarification in WholeAcoT, we denote the set of ants as $AntSet$, and each individual ant as $AntPaths$. The traversed test path, which is interpreted as a test case, is represented by TP . The number of ants, maximum test suite length, and maximum test case length are denoted as $AntNum$, $MaxTSLen$, and $MaxTCLen$, respectively. $TestStep$ refers to the set of transitions at a particular step in the ant's movement, with one transition per TP in an ant. The following relations outline the initial setup of the proposed approach.

$$AntSet = \{A_i : Ant \mid i: 1..AntNum\} \quad (5)$$

$$AntPaths = \{TP_j : TestPath \mid j: 1..MaxTSLen\} \quad (6)$$

$$TestStep_i = \{t_{ij} \in Transitions \mid \text{the } i\text{th transition of the } j\text{th test path, } j: 1..MaxTSLen\} \quad (7)$$

► **Example:** The HMS example has 2 ants: $AntSet = \{A_1, A_2\}$. As the $MaxTSLen$ is 3, each ant will be described by $AntPaths = \{TP_1, TP_2, TP_3\}$. Also, for each step that the ant traverses, there are some selected transitions which are listed in $TestStep$. The movement of ant in the first step is randomly, so the considered ant chooses some applicable transitions which are: $TestStep_1 = \{t_{11}, t_{12}, t_{13}\}$. The status of construction graph is shown in Fig. 8. According to this figure, BookRoom (1, "Joachim Wegener") is the first selection of TP_1 , BookRoom (2, "Joachim Wegener") is for TP_2 and BookRoom (1, "Daniel Castro") for TP_3 .

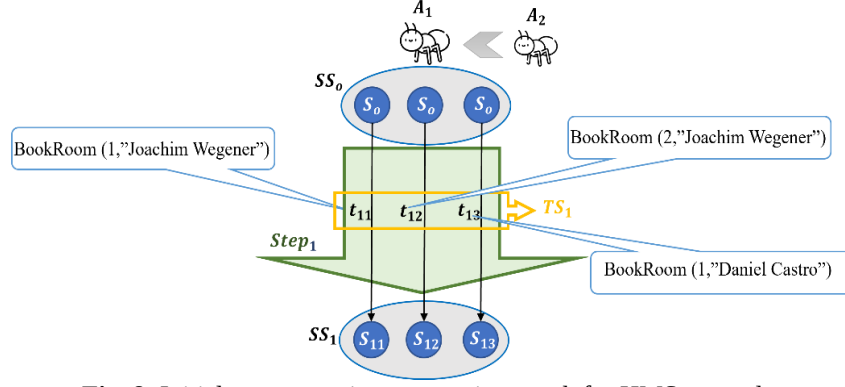


Fig. 8: Initial movement in construction graph for HMS example

4.3 Movement of the Ants

In this section, we will precisely illustrate the main phase of WholeAcoT. The goal of the proposed approach is to distribute ants across different paths to gather information and increase the probability of selecting better transitions, thereby covering more data dependencies in the entire test suite. In a super state SS_i of the construction graph, let T_{ij} be the set of outgoing transitions from S_{ij} . Thus, the next super states of SS_i are defined by relation 8.

$$NextSteps(SS_i) = T_{i1} \times T_{i2} \times \dots \times T_{in} = \{(t_{i1}, t_{i2}, \dots, t_{in}) : t_{i1} \in T_{i1}, t_{i2} \in T_{i2}, \dots, t_{in} \in T_{in}\} \quad (8)$$

There is a practical challenge with this relation. Since an ant can traverse multiple test paths and each T_{ik} can have several transitions, the set $NextSteps(SS_i)$ may become too large to be considered comprehensively. To address these challenges in WholeAcoT, we employed two methods.

First, we randomly select at most two transitions per T_{ik} , based on the coverage score of the outgoing transitions in their test paths. Specifically, we compute the coverage score for the outgoing transitions of each T_{ik} , generate a random number, and then select the first two transitions whose coverage scores are greater than the random number. This method ensures diverse transitions with relatively higher coverage scores.

On the other hand, given the limited number of ants searching within a large state space, the construction graph will inevitably be sparse, with many construction steps that have never been encountered before. As a result, the search process often encounters new test steps without any pheromone, leading to a race condition with existing siblings in the construction graph.

To address this issue in WholeAcoT, we employ the concept of similarity between new test steps and existing siblings. Since two test steps may share some transitions, the similarity between two test steps, $TestStep_i$ and $TestStep_j$, is defined by relation 9. To initialize the pheromone of a new test step for a super state SS , we identify the most similar sibling test step in the construction graph and initialize the pheromone of the new test step according to relation 10. In relation 9, $|TestStep_i \cap TestStep_j|$ is the number of common transitions of these two TestSteps and $|paths_{ant}|$ refers to the count of all transitions of them.

$$sim(TestStep_i, TestStep_j) = \frac{|TestStep_i \cap TestStep_j|}{|paths_{ant}|} \quad (9)$$

$$pheromone_{TestStep_i} = \max\{1, sim(TestStep_i, mostSimilarStep_i) * pheromone_{mostSimilarStep_i}\} \quad (10)$$

As illustrated in Fig. 6, each ant in an inner super state SS of the construction graph must choose its next test step among newly constructed steps and the existing outgoing steps of SS . The pheromone levels and the promising coverage of each outgoing test step guide this selection process. The choice of the admissible step is determined using Equation 11. According to this equation, a probability value is calculated for each step that originates from the current state. These steps and their associated probabilities are denoted by t and $prob[t]$, respectively. The probability is influenced by the prior pheromone ($Pheromone_t$) and the coverage of that step ($Coverage_t$). A higher coverage score from the transitions in a step leads to a higher probability. The parameters $Alpha$ and $beta$ are user-defined and determine the relative importance of pheromone and promising coverage, respectively.

$$\forall t \in Trans: Prob[t] = \frac{(Pheromone_t)^{\alpha} + (Coverage_t)^{\beta}}{\sum_{t \in Trans} (Pheromone_t)^{\alpha} + (Coverage_t)^{\beta}} \quad (11)$$

In each step of the movement, each TP will select an outgoing transition of the next step and move to its next state. Finally, the traversed step should be added to the construction graph.

► **Example:** Continuing with the HMS example, Fig. 9 illustrates the next *TestStep* for the ant A_1 . The possible next steps for SS_1 could be a subset of the following set:

$$NextStep(SS_1) = T_{11} \times T_{12} \times T_{13}$$

where

$$T_{11} = \{OccupyRoom(1, "Joachim Wegener", 1024), BookRoom(2, "Daniel Castro")\}$$

$$T_{12} = \{OccupyRoom(2, "Joachim Wegener", 1024), BookRoom(1, "Daniel Castro")\}$$

$$T_{13} = \{OccupyRoom(1, "Daniel Castro", 1023), BookRoom(2, "Joachim Wegener")\}$$

There are $2 \times 2 \times 2 = 8$ possible next step for this super state, SS_1 .

Table 3: Probability calculation for HMS example

$t \in T_{11}$	<i>OccupyRoom</i> (1, "Joachim Wegener", 1024)	<i>BookRoom</i> (2, "Daniel Castro")
<i>Coverage_t</i>	1	0
<i>Pheromone_t</i>	3	3
<i>Prob[t]</i>	$\frac{(3)^2 + 1^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.53$	$\frac{(3)^2 + 0^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.47$
$t \in T_{12}$	<i>OccupyRoom</i> (2, "Joachim Wegener", 1024)	<i>BookRoom</i> (1, "Daniel Castro")
<i>Coverage_t</i>	1	0
<i>Pheromone_t</i>	3	3
<i>Prob[t]</i>	$\frac{(3)^2 + 1^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.53$	$\frac{(3)^2 + 0^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.47$
$t \in T_{13}$	<i>OccupyRoom</i> (1, "Daniel Castro", 1023)	<i>BookRoom</i> (2, "Joachim Wegener")
<i>Coverage_t</i>	1	0
<i>Pheromone_t</i>	3	3
<i>Prob[t]</i>	$\frac{(3)^2 + 1^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.53$	$\frac{(3)^2 + 0^4}{((3)^2 + 1^4) + (3)^2 + 0^4}$ $\cong 0.47$

We assume there is a *TestStep* with a pheromone value of 6 that has a similarity rate of 0.5 with each *TestStep*. For simplicity, this *TestStep* is considered to have an equal similarity to all explored *TestSteps*. Consequently, the pheromone value for each *TestStep* will be $0.5 \times 6 = 3$. Table 3 shows the *Prob[t]* values for each transition in each *TestStep*, calculated using Equation 11. The coverage values represent the number of data-dependency relationships between rules. For example, when Joachim Wegener books room 1 in the first step and then occupies it in the next step, there is only a C_R dependency.

According to the probability values in the Table 3, we suppose that the selected *TestStep* is as follows:

$$\begin{aligned}
TestStep_2 &= \{t_{21}, t_{22}, t_{23}\} \\
&= \left\{ \begin{array}{l} OccupyRoom(1, "Joachim Wegener", 1024), BookRoom(1, "Daniel Castro"), \\ OccupyRoom(1, "Daniel Castro", 1023) \end{array} \right\}
\end{aligned}$$

The status of the construction graph is depicted in Fig. 9. The remainder of the example follows the same pattern as this step.

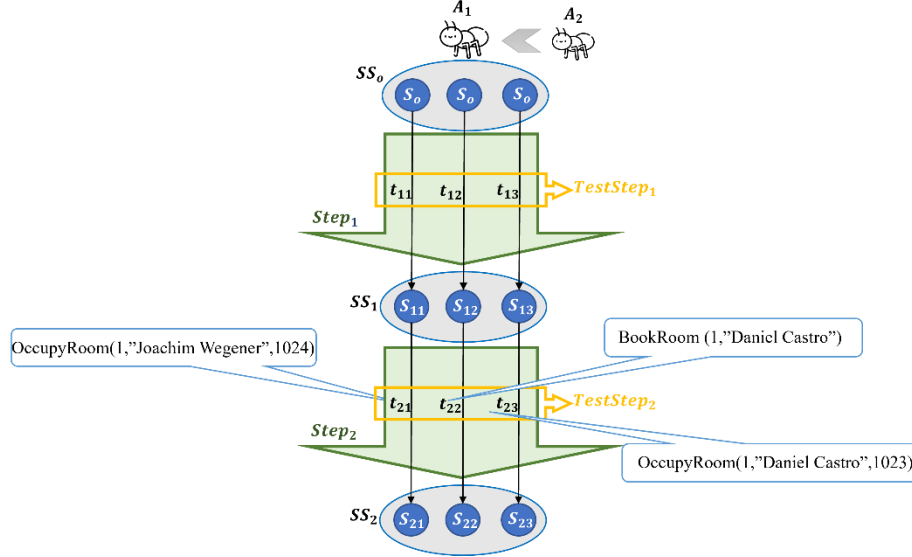


Fig. 9: Second movement in construction graph for HMS example

4.4 Update Pheromone Trails

After all ants have moved, several procedures will be carried out to update the pheromone values, construction graph, and output test suite:

Pheromone evaporation: In ACO, there is an evaporation rate that determines the amount of pheromone to be retained in each iteration. This value ranges between 0 and 1, and it is recommended to set it below 0.5 to reduce the convergence time for the paths of the ants. The equation 12 shows this formula:

$$\forall_{step \in Ant}: Pheromone_{step} = Pheromone_{step} * evaporateRate \quad (12)$$

Fitness calculation: The fitness/coverage of each test case and test suite is determined by the number of test objectives it covers. As explained in section 2.2.2, the test objectives are data-dependency relations between rules. During the fitness calculation process, these values are computed. In WholeAcoT, to update the pheromone to appropriate levels, we need to identify newly covered test objectives at different stages. To achieve this, we propose the following concepts, which will be calculated in this section and are prerequisites for the pheromone update process.

- Coverage Set of the i th transition in the j th test path ($CS_{t_{ij}}$) is the set of covered test objectives by transition t_{ij} for the first time in its TP.
- Coverage Set of the test step TS (CS_{TS}) is the union of covered test objectives by its transitions.

$$CS_{TS} = \bigcup_{t \in TS} CS_t \quad (13)$$

- Coverage Set of the ant A (CS_A) is the union of covered test objectives by its test steps.

$$CS_A = \bigcup_{TS \in A} CS_{TS} \quad (14)$$

► **Example:** To calculate the coverage set of ant A_1 up to the second step of the HMS example, we need to determine the coverage set of each TestStep and each of its transitions using Equations 12 and 13, respectively. Transitions t_{11} , t_{12} and t_{13} are the first transitions of the ant in TestStep₁, so they don't cover any test objectives.

$CS_{t_{11}} = \{\}, CS_{t_{12}} = \{\}, CS_{t_{13}} = \{\}$
Hence,

$$CS_{TS_1} = \cup CS_{t_{11}}, CS_{t_{12}}, CS_{t_{13}} = \{\}$$

In the second TestStep, there are three transitions: t_{21} , t_{22} and t_{23} . Regarding the coverage set $CS_{t_{21}}$ the application of BookRoom(1, "Joachim Wegener") in the first movement creates an edge labeled with "bookingInfo" between the room "Room 1" and the guest "Joachim Wegener" in the host graph. This edge is required to apply the next step, such as OccupyRoom(1, "Joachim Wegener", 1024). Therefore, a C_R relation exists between BookRoom and OccupyRoom. The coverage set for this transition is as follows:

$$CS_{t_{21}} = \{C_R_{BookRoom(1, "Joachim Wegener"), OccupyRoom(1, "Joachim Wegener", 1024)}\}$$

Since there is no data-dependency relation between BookRoom and BookRoom for different rooms, thus $CS_{t_{22}}$ is empty:

$$CS_{t_{22}} = \{\}$$

Following the same explanation as for $CS_{t_{21}}$, the coverage set for the third transition of this TestStep is as follows:

$$CS_{t_{23}} = \{C_R_{BookRoom(1, "Daniel Castro"), OccupyRoom(1, "Daniel Castro", 1023)}\}$$

Coverage Set of TestStep₂ is:

$$CS_{TS_1} = \cup CS_{t_{11}}, CS_{t_{12}}, CS_{t_{13}} = \{C_R_{BookRoom(1, "Joachim Wegener"), OccupyRoom(1, "Joachim Wegener", 1024)}, C_R_{BookRoom(1, "Daniel Castro"), OccupyRoom(1, "Daniel Castro", 1023)}\}$$

Therefore, CS_{A_1} is the set of the calculated coverage sets:

$$CS_{A_1} = \cup CS_{TS_1}, CS_{TS_2} = \{C_R_{BookRoom(1, "Joachim Wegener"), OccupyRoom(1, "Joachim Wegener", 1024)}, C_R_{BookRoom(1, "Daniel Castro"), OccupyRoom(1, "Daniel Castro", 1023)}\}$$

Pheromone Update: The pheromone value of each test step for an ant should be updated in each iteration. In WholeAcoT, we use Equation 15 to update the pheromone for all test steps of ant A. In this equation, the size of coverage set in each test step is multiplied by a user-defined coefficient η and then is summed. The ratio of this value to the sum of all other coverage sets determines the new pheromone value (Ph_A), which is then added to the previous amount.

$$Ph_A = Ph_A + \frac{\sum_{s=1}^{MaxTCLen} \eta^s |CS_{TS_s}|}{|\cup_{A \in Ants} CS_A|} \quad (15)$$

► **Example:** To update the pheromone of ant A_1 in HMS example using Equation 15, the Ph_A is zero and we set coefficient η to 0.7. According to the previous sections, $|CS_{TS_1}| = 0$ and $|CS_{TS_2}| = 2$. We suppose that the number of elements of coverage set for remaining test steps is as follows:

$$|CS_{TS_3}| = 3, |CS_{TS_4}| = 4, |CS_{TS_5}| = 4$$

Also, we suppose that the number of elements in the union of coverage set for the ants is:

$$|\cup_{A \in Ants} CS_A| = 10$$

Therefore, the new pheromone value for A_1 is:

$$Ph_{A_1} = 0 + \frac{\sum_{s=1}^5 0.7^s |CS_{TS_s}|}{|\cup_{A \in Ants} CS_A|} = \frac{0 + 2 + 3 + 4 + 4}{10} = 1.3$$

Update Best Ant: In this phase, one of the ants will be selected as the best ant based on the fitness of its test suite. Although the best ant is chosen through heuristic exploration on the construction graph, it may contain redundant transitions or even test paths. Therefore, the best ant will be minimized using a novel

procedure. Specifically, by traversing the ant’s test paths, any test path that does not cover a new test objective will be removed. Additionally, any final transition of a test path will be deleted if it has no new coverage set. By doing this, the best ant is updated, and ultimately, the output of WholeAcoT is the test suite of the best ant

► **Example:** Suppose that the best ant from an execution for HMS example is as follows. It has two TPs:

BestAnt: $A_x = \{$

```
{BookRoom(1,"J W"), OccupyRoom(1,"J W", 1023), UpdateBill(1023,20000),
  UpdateBill(1023,20000), ClearBill(1023)},
{BookRoom(1,"D C"), OccupyRoom(1,"D C", 1024), UpdateBill(1024,20000), ClearBill(1024)}
}
```

The ant has a duplicate UpdateBill transition in the first test path that does not affect the coverage size. Therefore, this transition will be removed during the minimization process. Additionally, the second test path has the same transitions as the first test path and does not cover any new test objectives, so the entire test path will be removed during the minimization process.

5 Evaluation

In this paper, we introduce a testing approach based on the whole test suite generation technique and the ant colony optimization algorithm for software models specified through graph transformation systems. The proposed approach, WholeAcoT, has four primary objectives. The first is to generate compact test suites that maximize coverage of test objectives for software modeled using GTS. The second is to develop a cost-aware test generation method to address the state space explosion problem in complex systems. **The third and fourth goals are to generate tests that rapidly converge to the best coverage score and present reliable coverage scores in different iterations. To evaluate these goals, we implemented WholeAcoT in the GROOVE toolset and assessed the related metrics. To this end, we consider four research questions, compared to state-of-the-art methods:**

- Q1: How much has WholeAcoT improved the coverage score?
- Q2: How much has WholeAcoT reduced the cost of the testing process?
- Q3: How much more rapidly does WholeAcoT converge to the best coverage score?
- Q4: What is the distribution of coverage values in different iterations of WholeAcoT?

Four each of the research questions, we use a metric from the literature. For the first question, we employ the coverage metric that is the most well-known indicator for evaluating testing approaches [40]. We count the number of covered test objectives to get the coverage score. The details of test objectives are provided in the Background section. For the second question, we utilize the cost metric to emphasize the proportionality of size and coverage for a generated test suite. In this context, cost is a type of ROI (Return On Investment) which computes the ratio of overhead and benefit of automatically generated tests [40]. For the third research question, according to common metrics for evolutionary algorithms [41], we use a convergence metric to investigate the speed of convergence after several iterations. Finally, to answer the fourth question we use a distribution metric which shows the dispersion of the coverage score across different iterations relative to the mean value. The rationale is that lower dispersion indicates that the coverage values are closer to the target value across different iterations, with fewer fluctuations.

We evaluate the performance of a range of approaches across different models. According to the literature, the methods most closely related to our model-based test generation approach are [5] and [6], both based on GTS. Therefore, we compare our proposed approach with these strategies. Additionally, we select the approach of [8], a more recent work, and implement its concept within our GTS framework to compare it with WholeAcoT. In the following, we first describe the experimental settings and selected case studies. We then evaluate the performance and cost of the proposed approach in comparison with similar methods, and provide answers to the research questions.

5.1 Experimental Setting

We implemented our WholeAcoT approach using the GROOVE toolset. GROOVE is a model checker for object-oriented systems specified by GTS models. Given sufficient memory, GROOVE can generate the state space of a finite-state GTS. In WholeAcoT, we made significant modifications to GROOVE to create effective test cases and test suites. **Our implementation is based on Groove 5.7.2. The newly added WholeAcoT package includes classes related to test generation, such as Ant, TestPath, TestStep,**

ACOSearch and ACOTestGenerator. We modified existing classes such as Closing Strategy. Additionally, we added new window to the GROOVE GUI to trigger the test suite generation. All code and case studies are available at <https://github.com/SGhasemi1987/WholeAcoT>.

In this experiment, we use data dependencies C_R, C_D, C_U, and D_N as our coverage criteria. The initial parameters and their values for executing WholeAcoT are listed in Table 4. These values were determined through multiple executions of the approach to achieve optimal results. The *MaxTCLen* and *MaxTSLen* values vary for each case study. *MaxTCLen* refers to the maximum length which test case can have and *MaxTSLen* is the maximum size of a test suite.

The value of *Beta* is set higher than *Alpha* because we found that when the selection probability (Equation 7) is more influenced by the fitness value than by prior pheromone, it yields better results. A smaller or larger evaporation rate can lead to later or earlier convergence, respectively, so the evaporation rate is set to 0.5. The number of iterations is set to 30, though in many cases, the algorithm converges earlier. The experiments were conducted on a PC with an Intel Core (TM) i7-8565U CPU at 1.99 GHz and 16 GB of memory.

Table 4: Parameters of WholeAcoT

Ant Numbers	MaxTCLen	MaxTSLen	Alpha	Beta	Evaporation Rate	Iterations
30	Variable	Variable	2	4	0.5	30

5.2 Case Studies

To evaluate our test suite generation, we use six case studies from the literature for assessing GTS-based test generation algorithms [33], [42], [43], [44], [45]. The first case study is the Dining Philosophers problem (DPs) [43]. The second case study is a specification of the Hotel Management Service (HMS), described in Section 2.1. In this specification, the rule dependencies include a higher number of D_N relations in addition to C_R, C_D, and C_U. The next three case studies are the Online Shopping System (OSS) [44], the Bug Tracker System (BTS) [42], and the Travel Agency System [45]. The final case study is the Scanflow Cash Register Protocol (SCRp) [33], which models an automatic Scanflow cash register in a store. This case study is larger and more realistic than the others. However, it is important to note that in MBT, only the critical portions of the SUT are modeled. After the evaluation, code-level implementation and testing are performed.

Table 5 presents the primary metric for the selected case studies, represented by rule numbers. This metric serves as an approximate indicator of the size of each case study. The first two case studies are relatively small, but due to the use of a fairly large start graph, they encounter the state space explosion problem. As a result, they serve as appropriate cases for comparing our proposed approach with other strategies. The next three case studies are sufficiently large to demonstrate the effectiveness of heuristic search in generating acceptable test suites. These cases are well-known in the literature for evaluating GTS-based test generation algorithms. The last case study is substantial enough to be considered an industrial model. The host graphs of all the case studies are designed to accurately model real-world behaviors, meaning even small host graphs can lead to state space explosion.

Table 5: Main metric of selected case studies

Case Study	DPs	HMS	BTS	OSS	TAS	SCRp
Rule Numbers	6	7	34	20	43	63

5.3 Experimental Results

To fully compare the performance of WholeAcoT against other strategies (GA [5], HGAPSO [6], and ISR-MCF [8]), we executed each method under similar conditions for 10 iterations, then calculated the average results. The experimental results are shown in Table 6. The first column of Table 6 lists the case studies, with their settings denoted as *MaxTCLen* (|TC|) and *MaxTSLen* (|TS|). These two parameters significantly influence the results. Larger *MaxTCLen* and *MaxTSLen* values result in larger generated test sizes, thereby increasing the cost of test execution. However, to achieve high coverage, we must select relatively large and balanced values for *MaxTCLen* and *MaxTSLen*.

The different parameter values for each case study reflect the nature of the individual cases. For instance, in the DPS case study, the primitive decisions for each test case are similar, meaning a single test case in

the test suite can cover the necessary test objectives, eliminating the need for multiple test cases. The same reasoning applies to the HMS case study. On the other hand, BTS, OSS, TAS, and SCRP are more complex case studies where generating long test cases is impractical. Therefore, we executed them with multiple short test cases to cover as many test objectives as possible.

In the third, fourth, fifth, and sixth columns, each cell contains two values. The first value represents the coverage score of the corresponding approach for the respective case study, while the second value indicates the size of the generated test. It is important to note that test size differs from |TS|: test size refers to the overall size of the generated test, whereas |TS| indicates the number of test cases. The maximum possible test size is |TC| * |TS|, although in most cases, it is smaller.

For example, the cell $\frac{17.5 \pm 0.5}{17.3 \pm 2}$ indicates that, after 10 executions of the WholeAcoT approach for the DPS case study, the test covers an average of 17.5 test objectives with a standard deviation of 0.5, and its size is 17.3 transactions with a standard deviation of 2. In Table 6, bold values highlight the best results in each row. These values allow us to address the research questions based on the comparison results.

Table 6: Comparison of average and standard deviation for coverage-score and test-size for different case studies

Case Study (TS , TC)	Approaches:	WholeAcoT	GA	HGAPSO	ISR-MCF
DPS (1,20)	Coverage	17.5 ± 0.5	17.1 ± 0.5	2 ± 0	13.6 ± 1.1
	Test Size	17.3 ± 2	29.3 ± 3.4	19.3 ± 0.9	20 ± 0
HMS (1,20)	Coverage	34 ± 0.4	33.9 ± 0.7	8 ± 0	27.19 ± 4.6
	Test Size	18 ± 1.5	32.8 ± 3.5	18.7 ± 0.9	20 ± 0
BTS (7,15)	Coverage	85.9 ± 4.3	74.5 ± 22.5	49.2 ± 5.49	84 ± 12.4
	Test Size	29.2 ± 0.97	79 ± 17.67	105 ± 0	105 ± 0
OSS (4,30)	Coverage	207.8 ± 1.24	122.9 ± 38	46 ± 1.19	166.3 ± 24.2
	Test Size	113.6 ± 4.47	74.3 ± 16.1	103.6 ± 12.49	120 ± 0
TAS (7,15)	Coverage	286.7 ± 5.02	203 ± 76	59.7 ± 6.48	230 ± 23.3
	Test Size	99.9 ± 3.03	75 ± 21	104.4 ± 1.8	105 ± 0
SCRP (10,30)	Coverage	457.1 ± 10.6	390.9 ± 53.5	108 ± 2.1	454.5 ± 39.4
	Test Size	207.5 ± 1.5	193.9 ± 31.3	292.3 ± 6.5	300 ± 0

Q1: How much has WholeAcoT improved the coverage score?

A comparison of the values in the first rows of each cell for every case study in Table 6 shows that the WholeAcoT approach consistently covers more test objectives than GA, HGAPSO, and ISR-MCF across all case studies. In the smaller, simpler case studies like DPS and HMS, the difference in coverage scores between the WholeAcoT and GA approaches is not significant. This is likely because, in most runs, the maximum possible coverage score was achieved by both the WholeAcoT and GA approaches. However, these two approaches diverge significantly in other case studies, with WholeAcoT demonstrating superior performance. The HGAPSO approach generally yields the lowest coverage scores in most cases. While the ISR-MCF approach achieves acceptable coverage across all case studies, its large test size and associated high cost diminish this advantage.

A general overview of Table 6 reveals that as test size increases, coverage of test objectives also tends to increase, with this relationship being more consistent and accurate for WholeAcoT. In contrast, for the

other approaches, there are instances where the test size triples compared to other case studies, but the corresponding coverage does not even double.

Fig. 10 shows the coverage score for each case study using test suites generated by WholeAcoT, GA, HGAPSO, and ISR-MCF. This comparison demonstrates that the coverage score of WholeAcoT consistently exceeds that of HGAPSO across all case studies. Furthermore, in more complex case studies such as BTS, OSS, and TAS, WholeAcoT significantly outperforms GA [5], HGAPSO [6], and ISR-MCF [8]. WholeAcoT's utilization of local search by each ant to select more appropriate states contributes to its high test objective coverage score. In the final case study, the coverage score of ISR-MCF is close to that of WholeAcoT, but the two approaches differ in terms of test size. These results prove that WholeAcoT achieves high coverage and is well-suited for solving real-world complex problems.

Q2: How much has WholeAcoT reduced the cost of testing process?

When evaluating any test suite generation solution, the coverage score is the most important metric. However, when the test suite is excessively large, the coverage value loses its significance. The applicability of a test suite is closely related to its size, as lengthy test suites are generally undesirable. According to Table 6, WholeAcoT generates test suites that not only achieve a higher level of test objective coverage but are also much smaller in size. The bold values in the second row of each case study indicate the smallest generated test suite sizes. In most cases, WholeAcoT produces the most compact test suites, except in the cases of OSS, TAS, and SCRP, where the superior coverage score justifies the larger size. In general, smaller test suites are preferable for real-world problems, which typically involve large and complex models.

Fig. 11 represents a comparison of the testing costs associated with WholeAcoT, GA [5], HGAPSO [6], and ISR-MCF [8] across various case studies. In this figure, testing cost is calculated as the ratio of test suite size to its coverage value. A smaller test suite combined with a higher coverage value results in a lower cost, which is desirable. According to this figure, our approach generates test suites with appropriate costs in all case studies, making it suitable for real-world complex problems. Additionally, WholeAcoT is designed to produce multiple small test cases rather than lengthy ones, which are more challenging to manage in complex scenarios

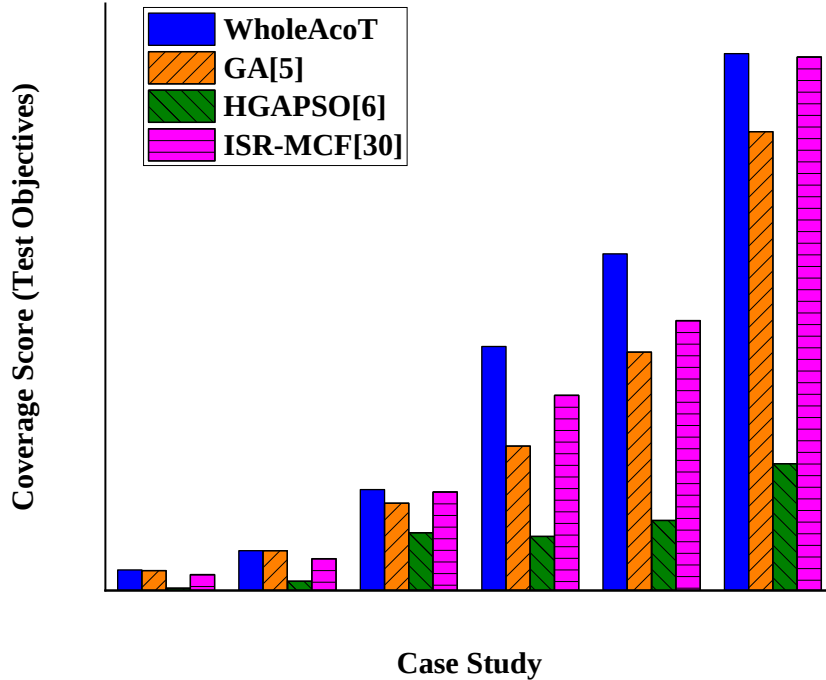


Fig. 10: Coverage score of WholeAcoT vs GA[5], HGAPSO[6] and ISR-MCF[8] in different case studies

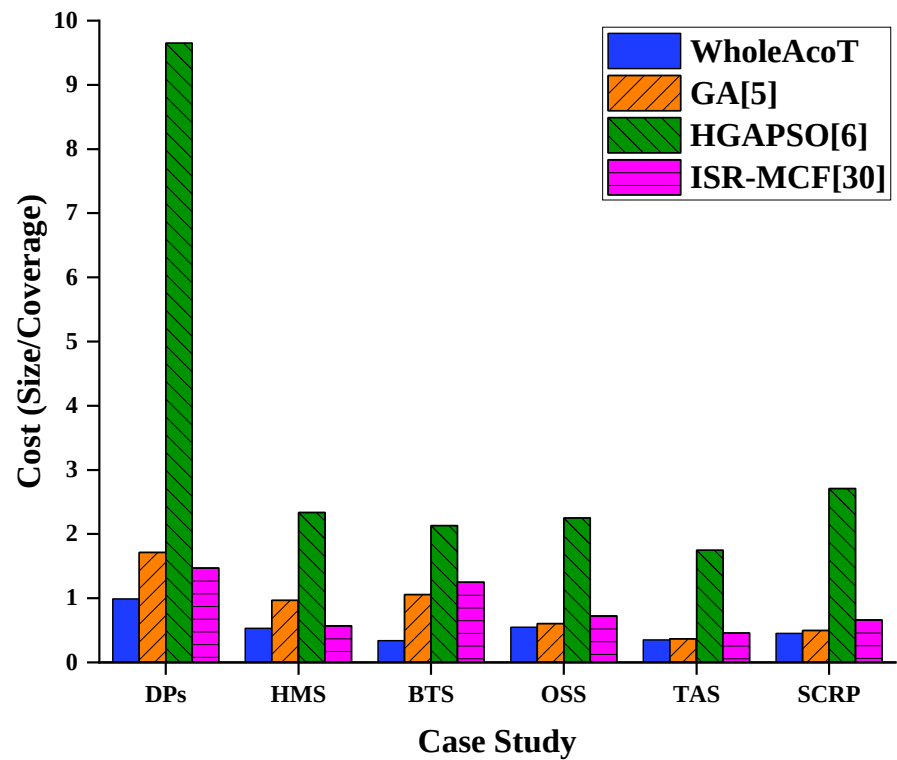
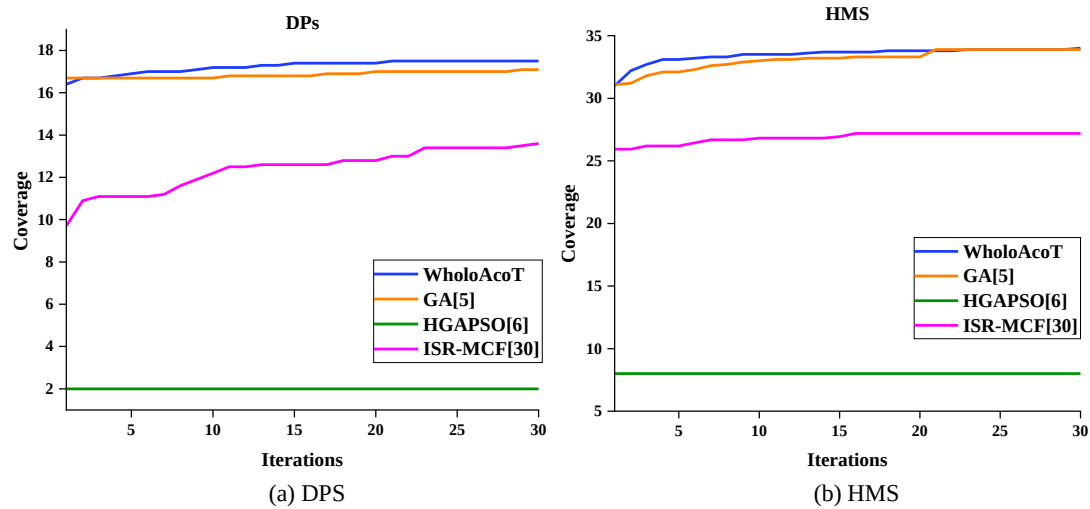


Fig. 11: Comparison of testing Cost for WholeAcoT, GA[5], HGAPSO[6] and ISR-MCF[8] in different case studies



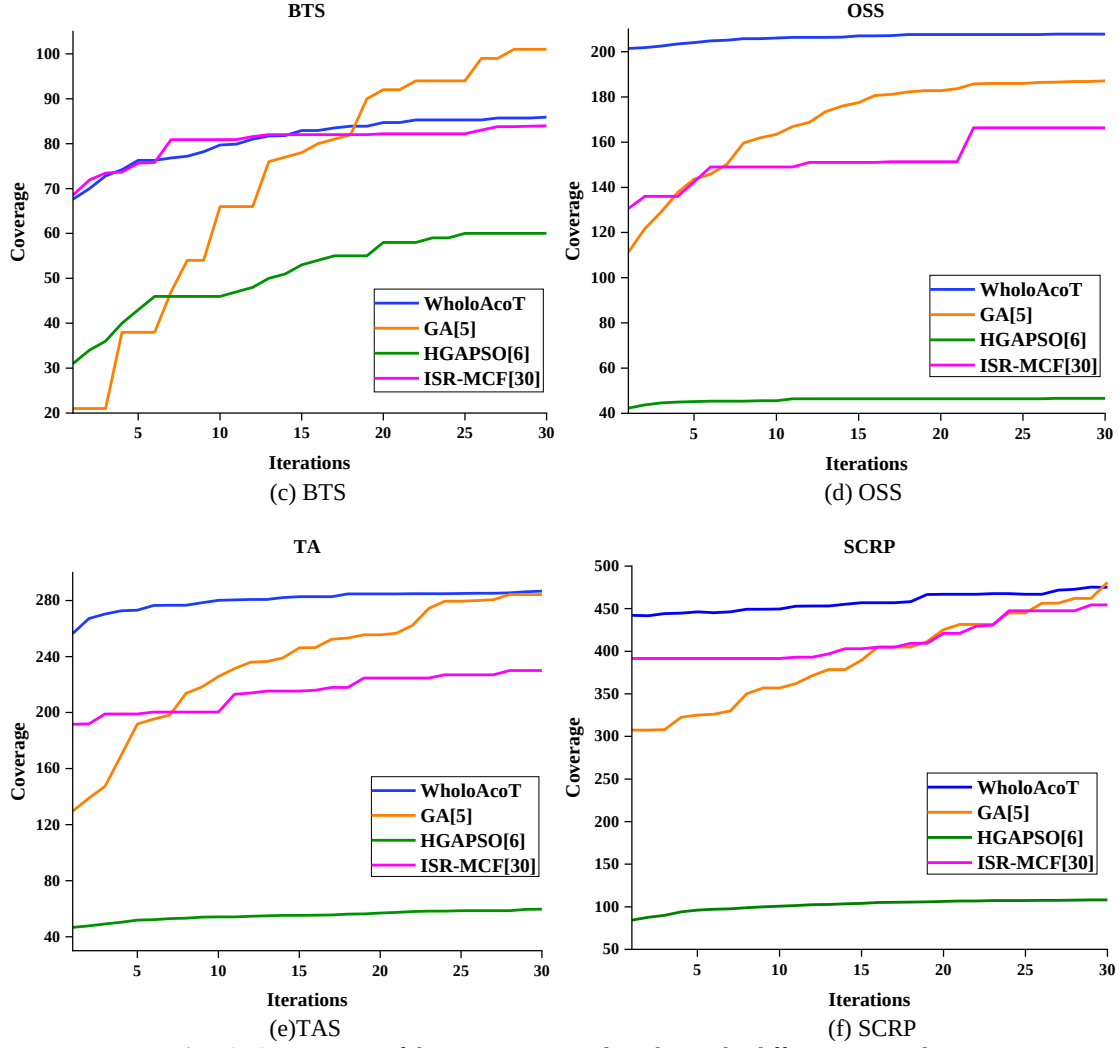


Fig. 12: Convergence of the test generation algorithm in the different case studies

Q3: How much more rapidly does WholeAcoT converge to the best coverage score?

A key criterion for evaluating the efficiency of a new strategy is the convergence speed in reaching the optimal value. To assess this, we executed WholeAcoT, GA [5], HGAPSO [6], and ISR-MCF [8] over 30 iterations (generations) across 10 trials, then averaged the coverage scores to illustrate the convergence patterns. Fig. 12 presents these results, with each chart depicting the data for a specific case study.

In Fig. 12(a) and 12(b), WholeAcoT and GA [5] exhibit similar behavior, both quickly achieving the optimal value. As noted earlier, in these two examples, both GA [5] and WholeAcoT reach maximum coverage, making the difference between these strategies negligible.

In Fig. 12(c), 12(d), 12(e), and 12(f), WholeAcoT converges to the best coverage score much earlier than GA [5], HGAPSO [6], and ISR-MCF [8]. In Fig. 12(c), while GA [5] achieves higher maximum coverage scores than WholeAcoT, its average coverage is lower, and the high variance indicates instability in the results. Additionally, GA [5] shows delayed convergence in other large case studies, which is a drawback of this approach. The convergence speed of HGAPSO [6] is acceptable, but its low coverage scores diminish its effectiveness. The convergence behavior of ISR-MCF [8] is somewhat similar to that of WholeAcoT, but in most cases, ISR-MCF [8] converges later, often not until the twentieth iteration.

Q4: What is the distribution of coverage values in different iterations of WholeAcoT?

In Table 6, the data in the first rows of each case study represent the mean and standard deviation of the coverage scores. Another important metric in evaluating coverage scores is the distribution of these values. The distribution of data across different iterations reveals how much the method focuses on the mean value.

A more compact data distribution indicates that the approach is more consistent and predictable. Fig. 13 illustrates the distribution of coverage values for each approach across the case studies.

According to this figure, the distribution of WholeAcoT is more compact than for the other approaches. This is due to its search method, which generates test cases almost simultaneously, achieving high coverage values in the initial iterations. In contrast, GA [5]—despite its high coverage values—exhibits a more widespread distribution. Therefore, in GA [5], a larger number of iterations is crucial for obtaining better results.

For HGAPSO [6], the distribution is centered around the mean, with the best values typically achieved in the early iterations. The data distribution in ISR-MCF [8] is also generally centered around the mean values. However, in some case studies, the distribution varies, which could be attributed to the random starting conditions of the method.

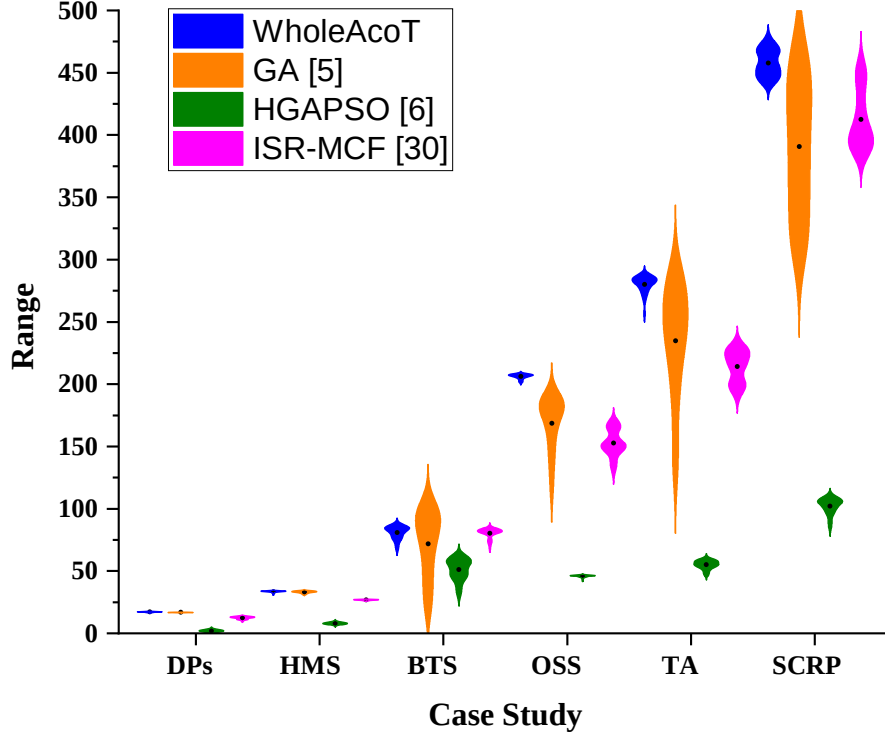


Fig. 13: Distribution of the coverage score for all approaches in the different case studies

5.4 Discussion

In this section, we discuss our findings from the evaluation to support the validity of the proposed test suite generation approach and address some potential weaknesses. As detailed in Section 5.3, our experimental results demonstrate that WholeAcoT outperforms other strategies. This effectiveness can be evaluated from three key aspects: coverage score, test suite/test case (TS/TC) size, and testing cost.

Fig. 10 and Table 6 show that WholeAcoT significantly improves the coverage of test objectives in most case studies. The approach's efficiency is attributed to its simultaneous execution of local and global searches by ants, which enhances coverage scores, particularly in large and complex case studies. Additionally, the results indicate that the test cases generated by WholeAcoT generally have small and acceptable lengths.

It is important to assess the size of test suites and their respective test cases in relation to their coverage scores. While smaller test suites are expected for lower coverage scores, larger test suites are justified when covering a high number of test objectives. For example, the test suites generated for the OSS and TAS case studies by WholeAcoT cover a substantial number of test objectives, making their size reasonable and justifiable.

One of the notable weaknesses of WholeAcoT is the variety of permutations for output transitions of a graph state. This issue leads to numerous TestStep choices and can potentially cause a state space explosion. As discussed in Section 4.3, the outgoing transitions can be filtered based on coverage score, and TestStep similarity is used to predict new values. While these innovations help mitigate the challenge, it remains a significant concern for large models.

6 Conclusion and future work

One of the most efficient methods for assessing the quality of a software system is the software testing. Model-based testing offers a cost-efficient approach to evaluate the software based on the designed model rather than the implemented code. Model-checker based testing is a compelling approach for generating test cases from the model. Model-checkers are originally designed for formal-verification of finite-state systems. Using them for test case generation might result in some challenges such as state space explosion. To address this challenge, several approaches in the literature have employed evolutionary algorithms such as GA and PSO to explore the state space heuristically. While these approaches have improved the results, some operations like crossover or velocity can lead to generate invalid test cases and test suites.

Most of the approaches in the literature generate test cases individually and then use procedures to merge the test cases into a test suite as well as to minimize its size. *Whole test suite generation is a promising technique to mitigate these issues. In whole test suite generation, a test suite is considered as a whole and test cases are produced according to the relations between them.*

In this paper, we designed an ACO-based algorithm, WholeAcoT, to generate efficient and compact test suites. We utilized data-dependency coverage as a reliable measure to test software models specified through GTS. The cost-effectiveness of the proposed approach has been evaluated against related works in the literature. Our results demonstrate that the redesigned ACO, combined with the whole test suite generation strategy, reduces the length of the test suites while maintaining coverage comparable to previous approaches. The size of the test suite is particularly crucial for large models with extensive state space graphs. Similarly, minimizing the cost of the testing process is essential for ensuring practical test execution. In the proposed approach, testing costs have been significantly reduced across various case studies, alongside an increase in coverage scores.

One notable limitation of the proposed approach is the variety of possible permutations for the ants, which can complicate the test generation process. As a future direction, integrating neural networks and reinforcement learning could optimize and filter ant selections. This approach may reduce test generation costs and accelerate the process of achieving maximum coverage scores.

Declarations

Authors' Contribution:

It should be declared that Simin Ghasemi, Vahid Rafe, Anvar Bahrapour and Reiko Heckel:

- reviewed the manuscript.
- agreed with the content and that all gave explicit consent to submit.
- made substantial contributions to the idea, structure and writing.
- revised the work critically for important intellectual content.
- approved on the version and agreed to be accountable for all aspects of the work.

Funding:

The authors (Simin Ghasemi, Vahid Rafe, Anvar Bahrapour and Reiko Heckel) did not receive support from any organization for the submitted work.

Conflicts of interest:

The authors (Simin Ghasemi, Vahid Rafe, Anvar Bahrapour and Reiko Heckel) have no relevant financial or non-financial interests to disclose.

Data Availability Statement

It should be mentioned that the Groove toolset version designed for this research beside the necessary case studies are available on:

<https://github.com/SGhasemi1987/WholeAcoT>

Any more questions about the data of the research could be sent to the email of corresponding authors.

References

- [1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.
- [2] V. Sumalatha and G. Raju, “Model-based test case optimization of UML activity diagrams using evolutionary algorithms,” *Int J Comput Sci Mob Appl*, vol. 2, no. 11, pp. 131–142, 2014.
- [3] M. Utting, B. Legeard, F. Bouquet, E. Fournieret, F. Peureux, and A. Vernotte, “Recent advances in model-based testing,” *Advances in Computers*, vol. 101, pp. 53–120, 2016.
- [4] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
- [5] A. Bahrampour and V. Rafe, “Using search-based techniques for testing executable software models specified through graph transformations,” *International Journal of Machine Learning and Cybernetics*, vol. 11, pp. 2743–2770, 2020.
- [6] A. Kalaei and V. Rafe, “Model-based test suite generation for graph transformation system using model simulation and search-based techniques,” *Inf Softw Technol*, vol. 108, pp. 1–29, 2019.
- [7] V. Rafe, S. Mohammady, and E. Cuevas, “Using Bayesian optimization algorithm for model-based integration testing,” *Soft comput*, vol. 26, no. 7, pp. 3503–3525, 2022.
- [8] R. A. Sulaiman, D. N. A. Jawawi, and S. A. Halim, “Cost-effective test case generation with the hyper-heuristic for software product line testing,” *Advances in Engineering Software*, vol. 175, p. 103335, 2023.
- [9] G. Fraser and A. Arcuri, “Whole test suite generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.
- [10] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE Comput Intell Mag*, vol. 1, no. 4, pp. 28–39, 2006.
- [11] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural, “A temporal logic based theory of test coverage and generation,” in *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, 2002, pp. 327–341.
- [12] R. Heckel and G. Taentzer, *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer, 2020. doi: 10.1007/978-3-030-43916-3.
- [13] R. Heckel, T. A. Khan, and R. Machado, “Towards test coverage criteria for visual contracts,” *Electronic Communications of the EASST*, vol. 41, 2011.
- [14] H. Kastenbergh and A. Rensink, “Model checking dynamic states in GROOVE,” in *International SPIN workshop on model checking of software*, 2006, pp. 299–305.
- [15] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.
- [16] S. Anand et al., “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.
- [17] V. Rafe, M. Darghayedi, and E. Pira, “MS-ACO: a multi-stage ant colony optimization to refute complex software systems specified through graph transformation,” *Soft comput*, vol. 23, pp. 4531–4556, 2019.
- [18] G. Fraser, F. Wotawa, and P. Ammann, “Issues in using model checkers for test case generation,” *Journal of Systems and Software*, vol. 82, no. 9, pp. 1403–1418, 2009.
- [19] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, “A detailed investigation of the effectiveness of whole test suite generation,” *Empir Softw Eng*, vol. 22, pp. 852–893, 2017.
- [20] O. Sahin, B. Akay, and D. Karaboga, “Archive-based multi-criteria Artificial Bee Colony algorithm for whole test suite generation,” *Engineering Science and Technology, an International Journal*, vol. 24, no. 3, pp. 806–817, 2021.
- [21] G. Fraser, A. Arcuri, and P. McMinn, “A memetic algorithm for whole test suite generation,” *Journal of Systems and Software*, vol. 103, pp. 311–327, 2015.
- [22] G. Fraser and A. Arcuri, “Achieving scalable mutation-based generation of whole test suites,” *Empir Softw Eng*, vol. 20, pp. 783–812, 2015.
- [23] A. Bahrampour and V. Rafe, “Using memetic algorithm for robustness testing of contract-based software models,” *Artif Intell Rev*, vol. 54, no. 2, pp. 877–915, 2021.

- [24] O. Grumberg, E. M. Clarke, and D. Peled, "Model checking," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*; Springer: Berlin/Heidelberg, Germany, 1999.
- [25] J. Callahan, F. Schneider, S. Easterbrook, and others, "Automated software testing using model-checking," in *Proceedings 1996 SPIN workshop*, 1996.
- [26] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.
- [27] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*, Springer, 2011, pp. 1–30.
- [28] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.
- [29] S. Liu, "Validating formal specifications using testing-based specification animation," in *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, 2016, pp. 29–35.
- [30] K. Bender, M. Broy, I. Péter, A. Pretschner, and T. Stauner, "Model based development of hybrid systems: specification, simulation, test case generation," in *Modelling, Analysis, and Design of Hybrid Systems*, Springer, 2002, pp. 37–51.
- [31] R. Heckel, "Graph transformation in a nutshell," *Electron Notes Theor Comput Sci*, vol. 148, no. 1, pp. 187–198, 2006.
- [32] H. Ehrig, G. Rozenberg, and H.-J. rg Kreowski, *Handbook of graph grammars and computing by graph transformation*, vol. 3. world Scientific, 1999.
- [33] V. Bruijn, "Model-based testing with graph grammars," University of Twente, 2013.
- [34] M. AsgariAraghi, V. Rafe, and A. Kalace, "Model-Based Test Case Generation from Graph Transformation Specifications using Beam Search Algorithm," *TABRIZ JOURNAL OF ELECTRICAL ENGINEERING*, vol. 49, no. 1, pp. 343–356, 2019.
- [35] L. Gönczy, R. Heckel, and D. Varró, "Model-based testing of service infrastructure components," in *Testing of Software and Communicating Systems: 19th IFIP TC6/WG6. 1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007. Proceedings*, 2007, pp. 155–170.
- [36] R. Aduni Sulaiman, D. N. A. Jawawi, and S. A. Halim, "Derivation of test cases for model-based testing of software product line with hybrid heuristic approach," in *Emerging Trends in Intelligent Computing and Informatics: Data Science, Intelligent Information Systems and Smart Computing 4*, 2020, pp. 199–208.
- [37] K.-W. Shin and D.-J. Lim, "Model-based automatic test case generation for automotive embedded software testing," *International Journal of Automotive Technology*, vol. 19, pp. 107–119, 2018.
- [38] A. Saeed, S. H. Ab Hamid, and M. B. Mustafa, "The experimental applications of search-based techniques for model-based testing: Taxonomy and systematic literature review," *Appl Soft Comput*, vol. 49, pp. 1094–1117, 2016.
- [39] A. Saeed, S. H. Ab Hamid, and A. A. Sani, "Cost and effectiveness of search-based techniques for model-based testing: an empirical analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 04, pp. 601–622, 2017.
- [40] P. Nirpal and K. Kale, "A Brief Overview Of Software Testing Metrics," *International Journal on Computer Science and Engineering*, vol. 3, Oct. 2011.
- [41] S. Jiang, Y.-S. Ong, J. Zhang, and L. Feng, "Consistencies and contradictions of performance metrics in multiobjective optimization," *IEEE Trans Cybern*, vol. 44, no. 12, pp. 2391–2404, 2014.
- [42] O. Runge, T. A. Khan, and R. Heckel, "Test case generation using visual contracts," *Electronic Communications of the EASST*, vol. 58, 2013.
- [43] A. Schmidt and D. Varró, "CheckVML: A tool for model checking visual modeling languages," in *International Conference on the Unified Modeling Language*, 2003, pp. 92–95.
- [44] G. Engels, B. Güldali, and M. Lohmann, "Towards model-driven unit testing," in *Models in Software Engineering: Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers 9*, 2007, pp. 182–192.
- [45] V. Rafe, "Scenario-driven analysis of systems specified through graph transformations," *J Vis Lang Comput*, vol. 24, no. 2, pp. 136–145, 2013.