



City Research Online

City St George's, University of London

Citation: Ghasemi, S., Rafe, V., Mehrabi, M., Heckel, R. & Al-Azzoni, I. (2025). Test Case Generation from Graph Transformation Systems Using Deep Reinforcement Learning. Paper presented at the 18th International Conference, ICGT 2025, Held as Part of STAF 2025, 11 - 12 Jun 2025, Koblenz, Germany. doi: 10.1007/978-3-031-94706-3_9

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/36739/>

Link to published version: https://doi.org/10.1007/978-3-031-94706-3_9

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](#).

Test Case Generation from Graph Transformation Systems using Deep Reinforcement Learning

Simin Ghasemi¹ , Vahid Rafe² , Mohammad Mehrabi³ , Reiko Heckel⁴ , and
Issam Al-Azzoni⁵ 

¹ Department of Computer Engineering, Faculty of Engineering, Arak University, Arak
38156-8-8349, Iran

`simin-ghasemi@pnu.ac.ir`

² Center for Software Reliability, Department of Computer Engineering, City St
George's, University of London, UK

`vahid.rafe@city.ac.uk`

³ Independent Researcher, Iran `mj.mehrabi96@gmail.com`

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

⁵ College of Engineering, Al Ain University, UAE

`issam.alazzoni@aau.ac.ae`

Abstract. Graph transformations can be used to specify and analyse software systems by modelling operations as rules and generating the labelled transition system (LTS) as a representation of system behaviour. Model-based testing (MBT) often uses model checking over LTS to discover paths that satisfy certain test requirements. Significant challenges include the size of the state space and the complexity of model checking. Meta-heuristic search-based approaches try to cope with this problem by exploring only a small portion of the LTS to produce paths that cover maximal test objectives. Despite acceptable results in small case studies, these approaches also do not scale well.

MBT approaches using graph transformation face the same limitations as LTS-based MBT in general, exacerbated by the more complex nature of graph-based LTS. However, apart from the behavioural information of the LTS, here we are able to use the graph structure of states and rules to define and evaluate test objectives. This enables a new method based on deep reinforcement learning to generate test suites for systems specified through graph transformations.

We use the reward/penalty mechanism of reinforcement learning to optimise the selection of moves within the state space, enabling the generation of test cases based on prior decisions. Our goal is to achieve greater coverage of test objectives while minimising the size of the test cases. The method has been implemented in GROOVE, an open-source toolset for designing and model checking graph transformation systems. Experimental results in well-known case studies demonstrate that our approach generates test cases with improved coverage scores while reducing the cost of testing compared to existing works.

Keywords: Model Checking-based Testing · Test Suite Generation · Graph Transformation Systems · Deep Reinforcement Learning · Neural Network.

1 Introduction

Software testing is an essential but expensive part of software quality assurance. Automated test generation can reduce the cost of testing and improve the effec-

tiveness of tests to discover errors. Model-Based Testing (MBT) [35] is an approach to generate and execute test cases from behavioural models of the system under test. Earlier fault detection reduces the cost of testing and increases the quality of the system under test. Various strategies have been used in MBT to generate more effective tests at a lower cost.

Model checking [25] has been applied to software testing for state-based models. Model checkers [5] were originally designed as formal verification tools capable of producing paths representing counterexamples/witnesses for the violation/satisfaction of a desired property. In Model Checking-based Testing (MCT), a model-checker is employed to take the test model as input and use test objectives as reachability properties to derive paths representing test scenarios. Although MCT ensures high coverage of test objectives for small models, its scalability is limited by state-space explosion, the complexity of model checking, and the generation of redundant test cases. One solution to this problem is to use search and optimisation methods to heuristically explore only a relevant portion of the state space. Genetic Algorithms (GA) [18], Particle Swarm Optimisation (PSO) [9] and Ant Colony Optimisation (ACO) [7] have been used, but there is still room for improvement in terms of eliminating redundant test cases and enhancing accuracy and efficiency. One challenge with these techniques is that they generate test cases in multiple iterations, making it difficult to identify and remove redundant test cases generated at different steps. Additionally, the heuristic functions used for test case selection can sometimes behave randomly, leading to the generation of incorrect or inaccurate test cases (such as a random crossover of test cases). Lastly, additional steps are often required to assess test case quality and eliminate redundancy or inaccuracies, introducing extra post-processing efforts and reducing overall efficiency. Graphs can represent, visualise, and analyse complex software models. Graph Transformation Systems (GTS) [10] use rules to represent pre- and postconditions of operations and model the behaviour of a system as a graph transition system, i.e., labelled transition systems (LTS) with graphs as states and graph transformations as transitions. Established algorithms and tools exist to simulate transformations, generate a graph-based LTS and analyse it through model checking.

However, both graph-based MCT and testing based on heuristic search and optimisation suffer from the same limitations as the generic approaches described above, in particular if they treat the graph transition system as a generic LTS without exploiting the graphical structure.

In this paper, we propose a testing approach for GTS that uses Deep Reinforcement Learning (DRL) to explore state space, finding test cases with the best coverage. The search begins at the initial state and selects actions randomly at first, then explores the state space based on the reward/penalty received for previous actions, ultimately generating semi-optimal test cases with high coverage and low cost. DRL is also able to improve over time by learning from the environment. The novel contributions of this paper are as follows:

- We develop a DRL-based approach to generate test cases for systems specified through GTS that cover more test objectives.
- The proposed approach is capable of storing network weights, allowing them to be reloaded in subsequent executions to reduce computational costs.
- The approach performs efficiently and is more scalable in terms of the time taken to generate test cases.
- The generated test suite can be executed at lower cost because it is smaller and contains shorter test cases.

The remainder of this paper is organised as follows. Section 2 describes the basic concepts of testing with GTS and DRL. Section 3 introduces the state of the art in MBT and reviews the existing work on GTS-based testing. Section 4 describes our approach in detail. The experimental results and the discussions are presented in Section 5. Finally, Section 6 concludes the paper and suggests future work.

2 Background

This section reviews the concepts of GTS testing and deep reinforcement learning.

2.1 Testing Graph Transformation Systems

Software testing is crucial today due to the increasing complexity of software systems and the need for reliability in critical applications. Automation testing is a good alternative to manual testing, but the problems of cost and uncertainty in the environment still remain [35]. Thus, to fully address these issues, MBT is used. MBT uses models to describe the environment and its behaviours, which can be easily extended when a feature is added or changed in the system. This approach leads to more efficient testing practices.

Model-checking [5] is a powerful procedure that can be utilised to automatically generate test cases from the model by exploring the state space to identify faults. In the MCT [1] procedure, a model of the system under test and a set of test objectives, expressed as reachability properties, are given to the model-checker. The behaviour of a system modelled as a GTS can be represented by an LTS providing a starting point for MCT. A test case is a finite path in the state space that starts at the start graph and satisfies certain test objectives expressed by reachability properties or more complex temporal formulas. The aim is to generate a test suite as a set of test cases that together satisfy the test objectives [20]. One of the most mature and effective tools for model checking graph transformation systems is GROOVE [28]. To revisit the concepts of GTS using GROOVE, interested readers are referred to the appendix.

2.2 Coverage Criteria

In MBT approaches that use GTS, criteria such as state coverage, rule-dependency coverage, transition coverage, path coverage, and rule coverage serve as test objectives to direct the testing process. As the rule-dependency coverage [19] gives an estimate of the def-use relations (as used in code-based testing) and could lead to the detection of potential bugs, many studies use it as a coverage criterion.

In GTS, the rules have some data sharing and affect each other. For example, according to Fig. 6, the application of the BookRoom rule appoints a vacant room to the guest by creating an edge labelled 'bookingInfo', which could be read later by the production rule CheckOut. In other words, the application of the rule CheckOut depends on the successful use of BookRoom to the system state. This is a type of rule dependency, where one rule creates/updates an entity (node/edge) and another rule uses (i.e. read/delete) it. The following is the definition of rule dependency: (For a more accurate definition, refer to [19])

Definition 1 (Rule-Dependency). *Given two production rules R_i and R_j , we say R_j is dependent on R_i if there are transformation steps $t_i = (S \xrightarrow{R_i} S_i)$ and s $t_j = (S_i \xrightarrow{R_j} S_j)$ such that t_i enables t_j and S is the a state graph.*

Data-flow relations between rules are considered where one rule can enable another, e.g., when it creates a node or edge or updates an attribute and another rule uses, reads, or deletes it, or when the first rule deletes entities that would violate the negative application conditions of the second. Hence, possible relationships between rules 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). In C_R a rule creates an entity that another rule reads, in C_D a rule creates an entity that another rule deletes, etc.

2.3 Deep Reinforcement Learning

Reinforcement learning is a method that aims at learning to control a set of actions based on feedback from a given environment. Formally, this can be described as a Markov Decision Process. In this framework, machines that learn and make decisions are referred to as agents, and everything external interacting with the agents represents the environment. This interaction forms the basis for decision making and learning, where the agent aims to maximise cumulative rewards. At first, the agent's knowledge is zero, but through interaction with the environment, it acquires information and learns how to achieve the defined goals [33]. At each step of reinforcement learning, the agent receives a state from the environment and performs an available action to transition to the next state. Based on the agent's action, it will receive a reward or penalty from the environment.

Deep reinforcement learning (DRL) uses particular neural networks, such as the Double Deep Q-Network (DDQN) [17]. In DDQNs, one neural network chooses the best action while the other one calculates the value of the selected action used to update the network's weights.

Some important terminology: A *step* denotes the execution of an action by an agent and the corresponding transition to a new state. An *episode* is a sequence of steps from the start to a final state. The *experience replay memory* is used to store the agent's past experiences. An *ϵ -greedy algorithm* is an exploration policy where the agent chooses the current best option most of the time and occasionally chooses a random option with low probability ϵ .

3 Related Work

Based on different techniques that existing approaches have used, we organise them in the following:

3.1 MBT in General

Model-based testing [35] is an effective method for validating the relationship between a model and its implementation where test generation is based on the model rather than the code of the system under test [15].

Most approaches to MBT rely on the mainstream Unified Modelling Language (UML). For example, [26] proposes an MBT approach for web applications using UML and Web diagrams, and [22] uses finite-state machines to reduce redundancy and improve efficiency. Some tools, such as Alloy [21], generate exhaustive test cases in bounded search spaces using SAT solvers. Alloy is a formal modelling language that allows the user to produce models of software systems by abstracting their key characteristics. Although our approach uses MBT techniques, it differs from these methods because it utilises GTS and optimisation, making it a search-based technique.

3.2 Model-checking and search-based techniques

Others have approached the expansion of MBT in different ways. For example, [36] provides a framework based on bounded model checking with coverage-guided fuzzing to generate test cases. They combined model checking with fuzzing for test case generation to generate initial seeds for fuzzers in order to give more interesting and effective seeds than manually created ones. The authors in [32] suggest a method called ISR-MCF to generate test cases in MBT. This method uses three search techniques: NSGA-II-LLH, SPEA2-LLH, and PSO-LLH. This method uses reinforcement learning to check how each operator performs and then picks the best one to create the test case. The study shows that ISR-MCF, when used with NSGA-II-LLH, performs better than the other methods. In [13], the authors study various methods of model checking-based testing and review the state of the art. They also investigated the MCT issues in [12] such as the quality and usability of the generated tests and proposed some techniques to overcome the problems. Our approach also employs model checking and optimisation-based search techniques; however, it is primarily based on GTS.

3.3 GTS and optimisation-based techniques

Some approaches utilise GTS for MBT. In [16], the authors present a methodology for testing components described in a high-level language using GTS. They employed model checking to find adequate test sequences for a given requirement.

Another challenge for MCT is state-space explosion. Some research uses search-based optimisation and AI methods to overcome this, such as Bee Colony, Genetic Algorithms (GA), Particle Swarm Optimisation (PSO), Bat Algorithms and Gravitational Search Algorithms [3,23,27,14,2]. These solutions partially explore the state space to extract paths with high coverage. [23] proposes a search-based hybrid GA and PSO algorithm (HGAPSO) to generate test cases from models specified through GTS. They defined test generation as an optimisation problem that aims to maximise the static data dependency of test paths. They used Groove to implement HGAPSO, and the experimental results show that the approach covers most of the test objectives, but there are challenges when defining the test objectives statically and HGAPSO cannot be applied in complex systems. Due to its limitations, [4] extend HGAPSO to a memetic Algorithm (MA) to perform robustness testing. Their main goals were to maximise coverage and minimise testing costs. The effectiveness of the strategy is evaluated through a type of mutation analysis at the model level. The experimental results demonstrate that the approach outperforms existing ones in fault detection. They also propose another approach in [3] in order to test software models specified through GTS. In this work, a GA has been used to maximise coverage of rule, state, transition, and data flow criteria. The results show that data-dependency coverage works better than other methods in complicated systems with many possible states and related rules. This work has a few drawbacks, like the computation complexity of the approach and also the length of the generated tests. In [14], the authors use Ant Colony Optimisation (ACO) to generate entire test suites instead of individual test cases. The results show that it creates better test suites in terms of size and coverage, but its main disadvantage is the cost of generating the state space and finding the test paths. In a recent work on MCT, the authors in [2] propose an approach that uses two data mining algorithms, Apriori and FP-Growth, to avoid exhaustive state-space exploration. They first explore a portion of the state space using a BFS strategy and then apply data mining algorithms to extract frequent patterns guiding the exploration process. However, this

method has two main drawbacks: its reliance on static dependencies to calculate the test case fitness and the need for an additional reduction phase to minimise the test suite, which significantly increases computational complexity.

Among the related approaches, we have selected HGAPSO [23], GA [3], and WholeAcoT [14] for comparison in different case studies. These approaches were chosen because they share the same objectives as our study, and their code is accessible.

4 Test Generation through Deep Reinforcement Learning

The main challenge of MBT is the quality of the generated test cases and test suites in terms of coverage and test size. The key objectives are to achieve high coverage, minimise test size, and develop an executable algorithm that avoids state-space explosion and excessive testing costs. Several approaches in the literature utilise AI methods to explore state space heuristically and choose promising paths. To the best of our knowledge, although most of these methods have improved MBT and provided useful results, the desired criteria are not satisfactorily covered. An effective method for exploring the state space is one that actively acquires knowledge during traversal and utilises this knowledge to guide the exploration of subsequent states. One possible way to design this method is a mechanism that calculates reward and penalty values to guide it toward better paths, a principle that lies at the core of reinforcement learning algorithms. In this paper, we propose MoTDeReL as an approach to create and optimise test cases for GTS, based on DRL. MoTDeReL stands for Model-based testing through DEep REinforcement Learning. Our aim is to cover more test objectives at each step based on accumulated experience while also reducing the costs associated with testing and test generation.

Fig. 1 illustrates our approach using two flowcharts, which we divided to simplify the explanation. According to Fig. 1(a), MoTDeReL initially receives the input model along with the appropriate parameter values. The process begins with the start graph of the input model as the initial state (later referred to as the current state). To facilitate a move (action, rule) within the state space, the approach employs the steps outlined in the second flowchart depicted in Fig. 1(b). In this flowchart (as the second flowchart), action selection is performed using the ϵ -greedy algorithm. Initially, a random number is generated; If this number is less than ϵ (an input parameter), the action is selected randomly; otherwise, it is chosen based on the recommendations of the neural network. The neural network can either be a pre-trained model or one with newly loaded weights. After applying the chosen action, its reward is calculated using the fitness function, and the action is stored in the experience replay memory. If the memory size reaches a predefined limit, a random batch from memory will be selected for training the network. Upon completion of the steps in the second flowchart (Fig. 1(b)), the approach resumes with the steps in the first flowchart (Fig. 1(a)). After the movement is executed, the length of the traversed path is checked. If the length of test cases reaches the maximum allowed for a test case or attains a final state, the path terminates. Additionally, the maximum length of the test suite restricts the number of generated test cases. If the fitness of a generated test case is repetitive up to a maximum number of allowed repeated rewards, that test case is disregarded. Finally, the generated test cases are printed as part of the final test suite. In the following subsections, we will explain each part of the proposed approach in more detail.

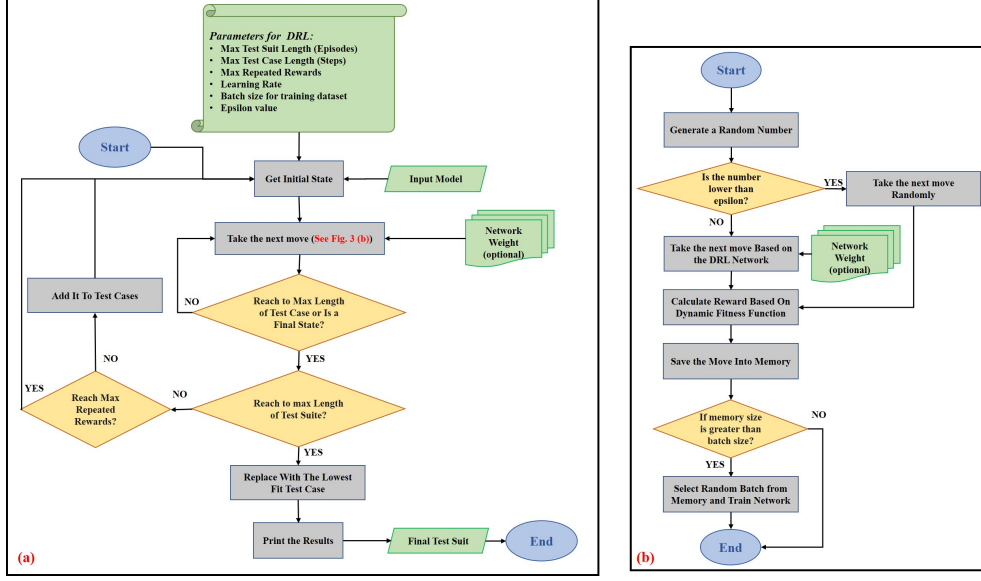


Fig. 1. The general flowcharts of the proposed approach (a) Main flowchart (b) Second flowchart

4.1 Neural Network Configuration

The core components of DRL are its neural networks. Neural networks consist of many simple processing nodes that are arranged in layers to learn weights between connections through several iterations of training data. In MoTDeReL we use a Double Deep Q-Network (DDQN) employing a multi-layer perceptron with three layers: an input layer, a hidden layer, and an output layer. The activation functions for the hidden layer and the output layer are the rectified linear unit and IDENTITY, respectively. The number of neurons in the hidden layer is set to two-thirds of the number of neurons in the input layer, as this configuration has shown optimal results in many case studies [24]. The number of neurons in the output layer is a variable parameter and will match the maximum number of applicable rules of the running model.

4.2 Parameter Analysis

There are different parameters in the DRL that should be analysed and initialised to determine the proper values for them. In this subsection, the most important parameters are explained:

The parameter ϵ is used for the ϵ -greedy algorithm. Its value is a number between 0 and 1. The higher its value, the closer the agent's behaviour is to random selection of actions. A low value for ϵ causes the agent to rely more on the recommendations of the neural network (exploitation) rather than exploring random actions. This means that the agent behaves more purposefully according to its learned policy.

In DRL, the value of ϵ is decreased in each step to balance exploration and exploitation. As ϵ decreases, the agent explores less and takes more advantages of learned experiences. Decreasing ϵ at each step ensures sufficient exploration early on, while gradually shifting to exploitation later in the process. There is another

parameter that determines the rate at which ϵ is reduced over time. This parameter controls the reduction in ϵ in each step, gradually changing the agent’s behaviour from exploration to exploitation. The larger this parameter, the earlier ϵ reaches zero. Small values are usually used for this parameter.

The learning rate directly affects the speed of convergence. Values of 0.1, 0.001 and 0.00001 have been obtained and validated in various experiments. With high learning rates, the algorithm may converge to a local minimum, while low learning rates can result in slow convergence. To achieve optimal results requires the use of an adaptive optimiser. One commonly used parameter for this purpose is Root Mean Square Propagation (RMSProp).

The number of actions (rules) is another important parameter to consider, as it defines the maximum number of output transitions a state can have in determining the number of neurons in the output layer of the neural network. It should be defined according to the problem setting. In many large case studies, more than 200 actions may be required.

When an agent relies solely on its previous experiences to learn, it can lead to an unstable system where the agent forgets its past experiences. To prevent this, the agent’s experiences should be stored in its experience-replay memory. Its size must be carefully chosen: A larger size increases storage usage but allows the agent to retain more past experiences, while a smaller size may cause the agent to forget older experiences and rely more heavily on recent ones.

The batch size of the training dataset determines how many samples are processed in each neural network training iteration. A larger batch size can enhance prediction accuracy, but requires more time and storage, while a smaller batch size may lead to poor future predictions. Therefore, balancing speed and accuracy is crucial.

4.3 Reward Engineering

Rewards management is crucial in reinforcement learning framework, as they guide the agent’s decision-making process by providing feedback on the consequences of its actions. To design a more accurate reward function, we should assign rewards to each action (rule) based on the relative worthiness or impact of that action in achieving the desired goal.

In our MBT, the goal is to generate a test suite containing test cases with the highest coverage and shortest length while minimising time and effort. For an accurate reward function, these parameters must be considered; however, the nature of DRL allows us to eliminate some of them. For the coverage score of generated test cases, in each step we count the data dependencies between the rules and return this number as a reward. The more data dependencies a test case covers, the higher its reward. Based on this reward function, the DRL agent aims to select rules that maximise dependency coverage.

For the length of the generated test cases, we initialise the discount factor (γ) value accordingly. The discount factor is an important parameter in reinforcement learning that influences the agent’s valuation of future rewards, the balance between exploration and exploitation, and the stability of the learning process. It determines the degree to which agents prioritise long-term rewards over short-term ones. If $\gamma = 0$, the agent will be completely myopic and learn about actions that produce an immediate reward. And if $\gamma = 1$, the agent will evaluate each of its actions based on the total sum of all future rewards. If we choose a value near 1 for γ , the behaviour

of the neural network will lead to a longer test case, and if the value was near zero, the length of the test case will be very short. According to these explanations, we choose the value for γ as 0.95 to prevent the agent from generating very short test cases with low coverage scores.

Algorithm 1 The Reward Function algorithm

- 1: **Input:** Current state of the model = sequence of selected rules in the state space
 - 2: **Output:** R = reward of the current state
 - 3: **Initialise** $dependencies = \{\}$ // The pair of dependent rules as C_R , C_D , C_U , D_N , U_R , and U_U
 - 4: **Extract** dependencies of rules
 - 5: **Delete** redundant pairs from dependencies
 - 6: **Set** R as size of dependencies
 - 7: **Return** R
-

For the time effort involved in the test-case generation, there is no need to parametrise the reward function, as the nature of DRL enables the agent to quickly learn the test case generation process and produce suitable test cases with minimal time effort. Although most rule selections in the early stages of DRL are random, after several steps, the neural network is appropriately weighted, and rule selection becomes more deliberate.

According to the above explanation, we set the reward function to calculate the data-dependency values of the rules. The reward function used for MoTDeReL is presented in Algorithm 1. The input of the algorithm is the sequence of selected rules in the state space as current state, and the output is the reward value for the current state of the selections. In line 3 of the algorithm, the set of dependencies is initialised. Then, in line 4, we extract the data-dependency relation of selected rules. $dependencies$ is a set of pairs of rules which have one type of data-dependency relation as C_R , C_D , C_U , D_N , U_R and U_U . Then in line 5, we delete repetitive pairs and in line 6 count the number of pairs in $dependencies$ and set it as the value for the reward R . In other words, this reward value is the fitness function of the current state of the agent in state space.

4.4 Feature Selection

Feature selection is the process of designing a model that is easier to understand and interpret by humans [8]. In large state spaces with numerous features, identifying the most relevant features serves to reduce the dimensionality of the feature space and improve model efficiency.

In MoTDeReL, the inputs to the neural network correspond to the sequence of rules applied, from the initial state to the current state. The number of inputs is also equal to the maximum number of steps the agent can take in the environment. For example, if the agent is allowed a maximum of 4 steps per episode, the neural network will have 4 inputs, all initialised to 0 at the beginning. The maximum number of steps allowed for the agent is determined by the maximum length of the test case, which is specified in the input parameters by the user. The user usually specifies it according to the input model and the maximum length to cover more dependencies.

An example is shown in Fig. 2 where the agent is allowed to take 4 steps, resulting in 4 inputs for the neural network. Suppose that the agent selects the rules in the

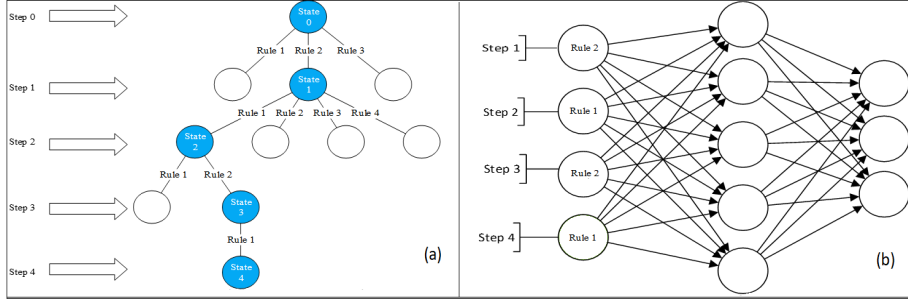


Fig. 2. An example for feature selection in MoTDeReL

following sequence: 2, 1, 2, and 1 (Fig. 2(a)). Consequently, the first input to the neural network becomes 2 in the first step. In the second step, the second input is updated to 1. The third input changes to 2 in the following step. In the final step, the fourth input changes to 1, after which the agent cannot proceed further. Therefore, the final inputs to the neural network are 2, 1, 2, and 1, which match the order of selection of the rules shown in Fig. 2(b). As explained previously, the number of neurons in the output layer is determined by the user to cover all applicable rules of the model. In this example, the output layer has 3 neurons, which means that the network can estimate one of three rules for the next movement.

4.5 Handling Invalid Actions

In MoTDeReL the agent in each step selects one rule according to the recommendation of the neural network. Sometimes, this rule is not applicable to the current state. These types of actions are referred to as invalid actions. To address this issue, a large output layer is created. After filtering the outputs using applicable rules, the highest Q-value is selected.

4.6 Double Deep Q-Network Implementation

The idea of MoTDeReL is illustrated in Algorithm 2 and Algorithm 3. In Algorithm 2, the next move execution on state space is presented. At first, in line 1, the *epsilon* parameter, the maximum steps for updating the target network (*MSUTN*), the maximum number of repetitions in a test case (*MFR*), the current state of the model and *allRewardsUntilNow* array are received. The user specifies the initial value for *epsilon* parameter of DRL. the *MSUTN* parameter specifies the number of steps required to regularly update the target network. The *MFR* is used to finalize the test case if the fitness value does not improve after a series of identical fitness evaluations. The *allRewardsUntilNow* array is an array which contains all gathered rewards until now in prior episodes. It will be updated in this algorithm. The algorithm finally will return the updated current state i.e., the next traversed state. In line 3, the experience replay memory $\mathcal{D}$ with capacity N , the main neural network Q_θ , the target neural network $Q_{\theta'}$ with weight 0 and batch size β are initialised. The algorithm starts with the extracting the applicable rules of the current state in line 4 denoted as *rules*. Then, according to the ϵ -greedy algorithm, a random number between 0 and 1 is generated as *rand*. Based on the value of *rand* and its comparison to the *epsilon* an action (rule) is selected. If the *rand* value is lower

than the *epsilon*, the action (rule) will be selected randomly (line 7). Otherwise, it is predicted using the main neural network (line 9). In line 11, the selected action is performed, the next state is explored. The reward of applied action on the current state will be calculated through Algorithm 1 in line 12. In line 13, the agent stores the last step as an experience in the experience replay memory $\mathcal{D}$. In line 14, if the size of the $\mathcal{D}$ is equal or greater than the batch size β , a random batch of experiences is retrieved from $\mathcal{D}$ and used to train the main neural network. As stated in lines 17 to 21, if the next state is a terminal state, the target value will be equal to the reward amount. Otherwise, the main neural network selects the best action for the next state, and the target neural network is used to estimate its Q-value, which is then added to the reward value. Here, the target value refers to the value used to update the Q-value of a state-action pair during training. For each value of the loop, the algorithm performs a gradient descent with respect to the network parameters. In line 24, if the steps value equals *MSUTN*, the weights of the two neural networks (main and target) are synchronized. In lines 28 and 30, the agent takes one step and moves to the next state and reduces the value of *epsilon*. In line 33, it counts the number of repeated rewards (e.g., *fitness*) in the path (e.g., *allRewardsUntilNow*) and if this number is greater than the MFR, the loop is broken. This is done to return the test case if the fitness does not improve after a number of repetitions. Finally, the Algorithm returns the updated current state at line 36.

Algorithm 3 presents the algorithm of MoTDeReL approach. In this algorithm, first, the GTS model of the system under test is provided as the input model. In addition, the number of algorithm episodes (M), the number of steps of the agent per episode (N), the maximum number of selected test cases (MST) as input parameters is received from the user. For each episode, the algorithm generates a test case and, for each step of the episode, it traverses a transition of the state space or performs an action. The *MST* determines the total number of test cases that are ultimately generated. In line 2, the output of the algorithm is determined which is the finally generated test cases. In line 3, the selected test case array is initialised. The first loop of the algorithm in line 4 is used to run multiple episodes. For each episode, the initial state of the input model is taken as s_t and *allRewardsUntilNow* is initialised. *allRewardsUntilNow* array will be used to store all the rewards (fitness) achieved for different test cases. In line 7 the *for* loop is used to run multiple steps. At each step, a movement on the state space will occur, representing an application of a rule which changes the current state to the next one. This is done using Algorithm 2 (line 8). The execution of Algorithm 2 leads to a sequence of states as a TC. After finishing the episodes or breaking out of the loop in line 10, the proposed approach checks if the optimal number of test cases has been reached. If so, the test case with the lowest fitness in the *selectedTestCases* array is replaced; otherwise, the new test case is added to *selectedTestCases*. Finally, in line 16, after termination of all episodes, the *selectedTestCases* is returned as the output.

5 Evaluation¹

In this paper, we propose an approach based on deep reinforcement learning to test the software models specified using GTS. The MoTDeReL approach has two main objectives: first, to generate a test suite that maximises the coverage of the test objectives, and second, to develop a cost-aware test generation methodology that

¹ The Groove toolset version designed for this research along with the necessary case studies are available at <https://github.com/SGhasemi1987/MoTDeReL>

Algorithm 2 The Test Case Generation Algorithm

```

1: Input:  $\epsilon$  as epsilon,  $MSUTN$  as Maximum steps for update target network,  $MFR$ 
   as Maximum number of fitness repetition for a test case,  $s_t$  as current state,
    $allRewardsUntilNow$  array
2: Output: updated  $s_t$ 
3: Initialise experience replay memory  $\mathcal{D}$  to capacity  $\mathcal{N}$ , primary network  $Q_\theta$  and target
   network  $Q_{\theta'}$  with weights 0, batch size  $\beta$ 
4: Extract rules as the list of applicable rules for  $s_t$ 
5: Generate a random number as  $rand$ 
6: if  $rand$  is lower than  $\epsilon$  then
7:   Select a random action  $a_t$  from 0 to  $\text{count}(\text{rules})$ 
8: else
9:   Select  $a_t = \arg \max Q_\theta(s_t)$  to count of rules
10: end if
11: Execute  $a_t$ , explore the next state  $s_{t+1}$ 
12: Set reward  $r_t = R(s_t, a_t)$  // Algorithm 1
13: Store  $(s_t, a_t, r_t, s_{t+1})$  in experience replay memory  $\mathcal{D}$ 
14: if  $\text{size}(\mathcal{D}) \geq \beta$  then
15:   Select batch = Sample random batch from  $\mathcal{D}$ 
16:   for each  $(s_j, a_j, r_j, s_{j+1})$  in batch do
17:     if  $s_{t+1}$  is a Terminal State then
18:        $Target = r_j$ 
19:     else
20:        $Target = r_j + \gamma Q_{\theta'}(s_{j+1}, \arg \max_a Q_\theta(s_{j+1}, a))$ 
21:     end if
22:     Perform a gradient descent step on  $(Target - Q_\theta(s_{j+1}, a))^2$ 
23:   end for
24:   if  $steps$  equals  $MSUTN$  then
25:     Reset  $Q_{\theta'}$  weights =  $Q_\theta$ 
26:   end if
27: end if
28: Set  $s_t = s_{t+1}$ 
29: if  $\epsilon > threshold$  then
30:   Reduce  $\epsilon$ 
31: end if
32: Store fitness in  $allRewardsUntilNow$ 
33: if  $\text{count}(r_t \text{ in } allRewardsUntilNow) > MFR$  then
34:   Break
35: end if
36: Return  $s_t$ 

```

Algorithm 3 The MoTDeReL Approach

```

1: Input: Model as A GTS model of the system under test,  $N = \text{Steps}$ ,  $MST$  as Maximum number of selected test case
2: Output: SelectedTestCases
3: Initialise SelectedTestCases array
4: for  $episodes = 1$  to  $M$  do
5:    $s_t = \text{initial state of the Model state space}$ 
6:   Initialise allRewardsUntilNow array
7:   for  $steps = 1$  to  $N$  do
8:      $s_t = \text{NextMoveExecution}(s_t)$  // Algorithm 2
9:   end for
10:  if  $\text{count}(\text{selectedTestCases}) > MST$  then
11:    Replace  $s_t$  with the lowest fitness testcase in selectedTestCases
12:  else
13:    Add it to SelectedTestCases
14:  end if
15: end for
16: Return SelectedTestCases

```

addresses the state space explosion problem in complex systems. To evaluate our solution against these objectives, we implemented MoTDeReL using the Groove toolset and assessed its coverage score and cost-effectiveness. In the following, we consider three research questions and answer them based on our findings.

There are several approaches in the literature that use MBT for graph transformations. The most relevant are HGAPSO [23], GA [3], and WholeAcoT [14]. Hence, we compared our proposed approach with these approaches.

We first describe the experimental setup. Then we evaluate the performance and cost of MoTDeReL in comparison with similar approaches and provide answers to the research questions.

5.1 Experimental Setup

We implemented MoTDeReL using Groove. Although primarily a model checker designed for verifying graph transformation systems, we adapted Groove to generate effective test cases and test suites. The experiments were carried out on a PC with an Intel Core (TM) i7-8565U CPU at 1.80 GHz and 16 GB of memory. In this experiment, we used data dependencies including C_R, C_D, C_U, D_N, U_R and U_U as our coverage criteria. As stated in Subsection 2.2, data-dependency gives a suitable estimation of the def-use relations among rules, and this is the motivation to reveal the bugs of the system under test. In order to specify suitable values for these parameters, we ran the experiment multiple times and chose the optimal values. The MST and step values that determine the maximal number and length of test cases, respectively, are variable for each case study.

The parameter ϵ is initially set to 0.2, allowing the agent to take some random actions. To gradually reduce randomness over time, we apply a decay factor of 0.995, ensuring that the epsilon decreases at each step. This approach progressively shifts the agent from exploration to exploitation.

The learning rate, a key parameter in neural network training, is set to 0.001, a commonly used value that facilitates a fast convergence to a global optimum. Additionally, we chose RMSProp as the optimiser to effectively regulate the learning rate.

After evaluating various case studies, we determined that a maximum action output parameter of 400 is optimal. This value is sufficiently large to accommodate most models while maintaining efficiency.

According to the explanation in Subsection 4.3, we set the discount factor to 0.95. The agent utilises this value as part of its decision-making process. Additionally, we set the memory size for the experience replay to 1000 to achieve a balanced trade-off between memory usage and learning stability. In addition, the batch size is set to 8 to achieve a balance between speed and accuracy. A very high batch size increases computational time and storage requirements, so we chose this value to effectively manage these trade-offs. In addition, we set the *MSUTN* value to 5 to update the target network as often as possible. The summary of the parameter setting for the experiments is listed in Table 1.

Table 1. Parameters of MoTDeReL

Parameter	Value
episode	10
step	variable
MST	variable
MFR	5
epsilon	0.2
decrease value of epsilon	0.995
learning rate	0.001
adaptive optimiser	RMSProp
Max action output	400
Max steps for update target network	5
discount factor	0.95
experience replay memory size	1000
batch size	8

Table 2. Rule Numbers, Step and MST for each case study

Case Study	Rule Numbers	MST	Step
DPs	6	1	20
HMS	7	1	20
OSS	20	4	30
BTS	34	7	15
TAS	43	7	15
SCRP	63	10	30

For the purpose of evaluation, some well-known case studies are selected from the literature, including the Dining Philosophers problem (DPs) [31], the Hotel Management System (HMS) [19], the Bug Tracker System (BTS) [29], the Online Shopping System (OSS) [11], the Travel Agency System (TAS) [34] and the Scanflow Cash Register Protocol (SCRP) [6]. The case studies have different sizes but all of them can encounter state explosion due to the use of a fairly large start graph. Table 2 presents the number of rules for each case study, serving as a metric of their relative sizes. Although the DPs and HMS case studies are relatively small, they

are fundamental examples commonly used in the literature to compare different strategies. To incorporate more realistic scenarios, we included BTS, TAS, OSS, and SCRP as real-world case studies. Also, We have determined different values for step and MST parameters with respect to each case study. These values were chosen to generate applicable test suites. In Table 2, the step and MST values are listed for each case study.

5.2 Experimental Results

To compare the performance of MoTDeReL against other strategies [3,23,14] we executed each method under similar conditions ten times and then calculated the average results. The experimental results are shown in Table 3. The first column lists the case studies, while the second, third, and fourth columns show the results for each approach, including the coverage score and test size. The coverage score is the total number of rule dependencies covered in the generated test cases, while the test size is the sum of the length of each test case in the resulting test suite.

Based on the values in Table 3, in the rest of this section, we analyse the data and address the research questions.

Q1: How much has MoTDeReL improved the coverage score compared to the state-of-the-art? The blue numbers in Table 3 are the results of the coverage values. The highlighted values in the table indicate the best results for each case study. A comparison of these values shows that the MoTDeReL approach achieves a higher coverage of the test objectives compared to GA, HGAPSO, and WholeAcoT. As the size of the case study increases, the difference in coverage scores achieved by the MoTDeReL approach becomes more significant. The HGAPSO approach produces the lowest coverage scores in most cases, whereas the GA approach achieves better coverage than HGAPSO, although still lower than both WholeAcoT and MoTDeReL. Although the coverage score of the WholeAcoT approach is higher than that of GA and HGAPSO, MoTDeReL outperforms all other approaches in terms of coverage.

Table 3. Comparison of average results for coverage and test size for different approaches

Approaches → Case Studies ↓		HGAPSO[23]	GA[3]	MoTDeReL	WholeAcoT[14]
DPS	Coverage	2.0	17.1	18.2	17.5
	Test Size	19.3	29.3	19.9	17.3
HMS	Coverage	8.0	33.9	54.3	34
	Test Size	18.7	32.8	19.5	18
BTS	Coverage	49.2	74.5	95.5	85.9
	Test Size	105.1	79.0	88.4	29.2
OSS	Coverage	46.0	122.9	223.2	207.8
	Test Size	103.6	74.3	119.0	113.6
TAS	Coverage	55.7	203.0	370.7	286.7
	Test Size	103.2	57.5	83.0	99.9
SCRP	Coverage	108.0	390.8	981.6	457.1
	Test Size	292.3	193.0	256.5	207.5

Q2: How much has MoTDeReL reduced testing costs compared to the state-of-the-art? In software testing, cost-effective tests are essential to ensure an efficient

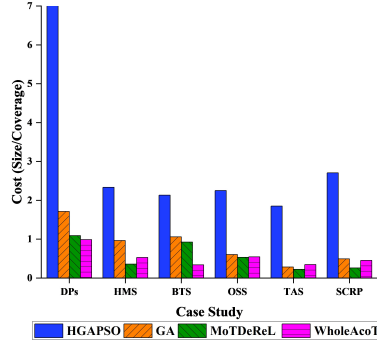


Fig. 3. Comparison of testing Cost for MoTDeReL, HGAPSO [23], GA [3] and WholeAcoT [14] in different case studies

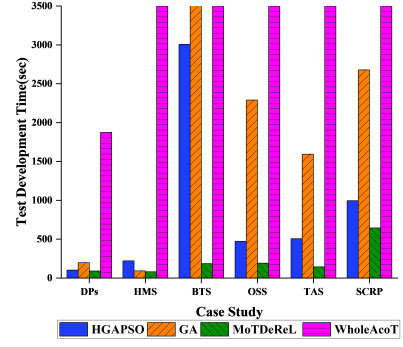


Fig. 4. Comparison of test development time for MoTDeReL, HGAPSO [23], GA [3] and WholeAcoT [14] in different case studies

testing process and developing high-quality software. When a smaller test case reveals more defects, its cost becomes significantly lower compared to larger tests with the same output. We aim for tests to achieve high objective coverage while remaining compact in size.

One commonly used metric for evaluating approaches is the length of the generated test cases. The black numbers in Table 3 indicate the size of the generated test cases for each approach, with the smallest test size in each case study is highlighted. For test size, we calculated the total size of the final generated test suite, which includes multiple test cases. Based on the black values in Table 3, the test cases generated by the MoTDeReL approach are not consistently shorter in most cases. However, upon closer examination, tests with shorter lengths cover significantly fewer test objectives, making their reduced size less beneficial for these approaches. To illustrate this point, we define the concept of testing cost as an efficiency metric [30] that describes the capacity of the specified test to fulfill the test objectives along with its size which affects the testing process:

Definition 2 testing cost: Given a test suite TS with length of L and coverage value of CV , the testing cost is defined as $\frac{L}{CV}$.

Fig. 3 represents the testing cost of generated tests associated with HGAPSO, GA, WholeAcoT and MoTDeReL. The goal is to have smaller test suites with higher coverage values, resulting in lower testing costs. According to this figure, the proposed approach generates test suites with appropriate costs in most case studies, making it more suitable for complex problems. Although in DPs and BTS, the cost of WholeAcoT is lower than that of MoTDeReL, its low coverage level in these case studies outweighs this advantage.

Q3: How does MoTDeReL works in terms of time compared to state-of-the-art methods? A key criterion for assessing the efficiency of a new strategy is the time required to complete the test design and development process. This time represents the cost of preparing the test, and it is desirable to generate tests in the shortest possible time.

Fig. 4 shows the time required for test generation by each approach for all case studies. As shown in the figure, the MoTDeReL approach is the most time-efficient compared to all other approaches in every case study. Although the time

difference between the MoTDeReL approach and both GA and HGAPSO is not very significant for small test cases, it becomes particularly notable for large and complex models. The time complexity of WholeAcoT is excessively high, making it an inefficient approach for complex case studies.

5.3 Discussion

In this section, we discuss the experimental results to validate the objectives of our test suite generation approach and address some potential weaknesses. Table 3 shows that MoTDeReL generates test cases and test suites that cover more test objectives and can detect more defects in the models. This performance is attributed to the well-trained neural network within the DRL structure. In MoTDeReL, the agent traverses the state space and receives rewards or penalties based on the coverage score of its actions. The neural network leverages this feedback to distinguish between effective and ineffective actions, thereby refining its decision-making process for future moves. This iterative learning mechanism allows the network to progressively improve its performance by prioritising actions that produce favourable outcomes while avoiding those that do not. Since these moves are guided by neural network recommendations, the exploration of the state space is conducted heuristically, enabling the discovery of paths with extensive coverage. This approach ensures that the system efficiently navigates through potential states, prioritising areas of interest while maintaining a broad exploration of possibilities. As a result, the neural network can identify and exploit high-value trajectories, balancing exploration and exploitation to optimise decision making over time.

Table 3 indicates that the tests generated by MoTDeReL are not smaller than those generated with the other approaches. Although generating smaller tests is advantageous, it is not the primary objective of test generation. The ideal scenario is to produce a small test suite that covers more test objectives. To quantify this, we introduced a metric called testing cost, which calculates the ratio of the test coverage score to the test size. For a small test with a high coverage score, this cost becomes smaller. According to Fig. 3, the test cost of MoTDeReL is lower than HGAPSO, GA, and WholeAcoT in most cases. This suggests that MoTDeReL generates an efficient test suite, improving the test objective coverage while maintaining a shorter length.

The data in Fig. 4 show that test generation in MoTDeReL is faster than other approaches. Since HGAPSO, GA and WholeAcoT utilise multiple generations to produce semi-random test cases and test suites, they require more time to produce the final generation. In contrast, MoTDeReL uses a previously trained neural network to select actions in new iterations, leading to improved test cases in terms of coverage score.

From the experimental results and discussions presented, we can conclude several advantages of the MoTDeReL approach:

- Generates acceptable test suites with high coverage compared to related approaches.
- Produces effective test suites with low testing cost relative to other methods.
- The runtime of MoTDeReL is significantly lower than that of other approaches.
- MoTDeReL improves with each iteration as it learns, progressively improving its performance until it achieves full coverage.
- Due to the nature of DRL, the results of each learning cycle can be stored and applied to larger or smaller host graphs, eliminating the cold start issue.

However, there are some weaknesses that should be addressed in future work.

- The length of generated tests could be shorter in certain models or case studies.
- In limited-time systems, a zero-knowledge agent may initially produce suboptimal tests due to the time required for DRL to acquire sufficient knowledge.
- Given MoTDeReL’s reliance on rewards and penalties, even minor changes to the reward function can have significant effects.

6 Conclusion

We proposed an approach that employs DRL to generate test cases and test suites for systems specified by graph transformation. This approach traverses the state space to explore desired test paths that cover a large number of test objectives. It utilises the penalty/reward mechanism of DRL to assess the effectiveness of selected actions and adjusts the neural network’s weights based on this feedback. The next action is then selected using the neural network’s recommendations.

We implemented the approach alongside other related methods in the Groove toolset and evaluated them on the basis of coverage score, testing cost, and test generation time. The experimental results indicate that our approach improves the testing process by generating test cases with a high coverage score at a lower cost and in significantly less time. Despite these improvements, there are areas for future work:

- Instead of randomly retrieving experiences, we can employ more intelligent techniques, such as prioritised experience replay.
- Although double DQN demonstrate strong overall performance in DRL, alternative methods such as the Actor-Critic method could also be explored.

Reward functions are crucial in every reinforcement learning algorithm, so optimising them could yield further enhancements.

7 Appendix: Graph Transformation in Groove

In this section, we present a Hotel Management System (HMS) originally introduced in [19] revised in [3] for MBT to recall concepts of GTS using the Groove tool set. The revised HMS includes multiple rooms and registered guests within a hotel. Each guest has the option to reserve at most one available room, leading to the generation of the relevant bill. Before proceeding to checkout, the guest must settle the bill. Fig. 5 shows a simple start graph of the HMS in Groove. Fig. 6 shows two sample rules of the HMS, which include BookRoom and CheckOut. In this figure, a solid black line denotes readers, blue dashed lines symbolise erasers, green solid lines represent creators, and forbidden elements, constituting negative application conditions, are indicated by red dashed lines. The complete description of this example and its rules is given in [4].

References

1. Anand, S.: An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* **86** (1978)
2. Asgari Araghi, M., Rafe, V., Khendek, F.: Using data mining techniques to generate test cases from graph transformation systems specifications. *Automated Software Engineering* **31**(1), 17 (2024)

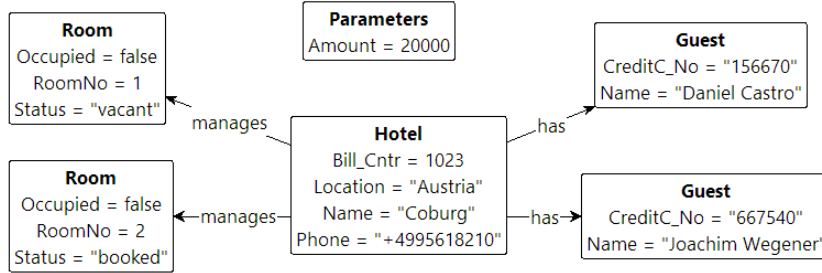


Fig. 5. A start graph for the HMS[4]

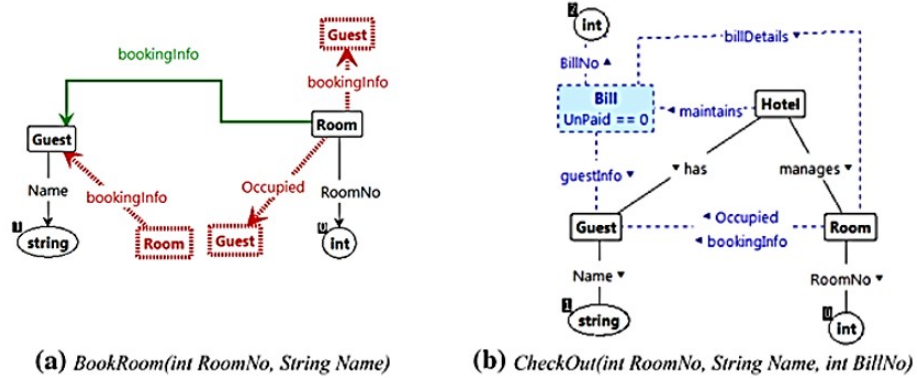


Fig. 6. Two sample rules of the HMS [4]

3. Bahrampour, A., Rafe, V.: Using search-based techniques for testing executable software models specified through graph transformations. *International Journal of Machine Learning and Cybernetics* **11**, 2743–2770 (2020)
4. Bahrampour, A., Rafe, V.: Using memetic algorithm for robustness testing of contract-based software models. *Artif Intell Rev* **54**, 877–915 (2021)
5. Baier, C., Katoen, J.P.: *Principles of model checking* (2008)
6. Bruijn, V.: *Model-based testing with graph grammars*. Ph.D. thesis, University of Twente (2013)
7. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. *IEEE Comput Intell Mag* **1**, 28–39 (2006)
8. Duboue, P.: *The art of feature engineering: essentials for machine learning* (2020)
9. Eberhart, R., Kennedy, J.: Particle swarm optimization. *Proceedings of the IEEE international conference on neural networks* pp. 1942–1948 (1995)
10. Ehrig, H., Rozenberg, G., Kreowski, R.H.J.: *Handbook of graph grammars and computing by graph transformation* **3** (1999)
11. Engels, G., Güldali, B., Lohmann, M.: Towards model-driven unit testing. *Models in Software Engineering: Workshops and Symposia at MoDELS* **9**, 182–192 (10 2006)
12. Fraser, G., Wotawa, F., Ammann, P.: Issues in using model checkers for test case generation. *Journal of Systems and Software* **82**, 1403–1418 (2009)
13. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. *Software Testing, Verification and Reliability* **19**, 215–261 (2009)
14. Ghasemi, S., Rafe, V., Bahrampour, A., Heckel, R.: Whole test suite generation from graph transformation specifications using ant colony optimization. *Software Quality Journal* **33**, 1–37 (2025)
15. González, C., Varmazyar, M., Nejati, S., Briand, L., Isasi, Y.: Enabling model testing of cyber-physical systems. *Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems* pp. 176–186 (2018)
16. Gönczy, L., Heckel, R., Varró, D.: Model-based testing of service infrastructure components. *Testing of Software and Communicating Systems: 19th IFIP TC6/WG6. 1 International Conference* pp. 155–170 (06 2007)
17. Hasselt, H.V., Guez, A., Silver, D.: Deep reinforcement learning with double qlearning. *Proceedings of the AAAI conference on artificial intelligence* (2016)
18. Haupt, R., Haupt, S.: *Practical genetic algorithms* (2004)
19. Heckel, R., Khan, T., Machado, R.: Towards test coverage criteria for visual contracts. *Electronic Communications of the EASST* **41** (2011)
20. Hong, H., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. *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* pp. 327–341 (04 2002)
21. Jackson, D.: *Software Abstractions: logic, language, and analysis*. MIT press (2012)
22. Jadhav, P., Lanke, G., Shrivastava, A., Patil, V., Gupta, S.: Efficient test case generation using model-based testing, and model paradigm approach. *International Conference on Recent Trends in Computing* pp. 817–828 (2023)
23. Kalaei, A., Rafe, V.: Model-based test suite generation for graph transformation system using model simulation and search-based techniques. *Inf Softw Technol* **108**, 1–29 (2019)
24. Mehrabi, M., Rafe, V.: Using deep reinforcement learning to search reachability properties in systems specified through graph transformation. *Soft comput* **26**, 9635–9663 (2022)
25. Mohalik, S., Gadkari, A., Yeolekar, A., Shashidhar, K., Ramesh, S.: Automatic test case generation from simulink/stateflow models using model checking. *Software Testing, Verification and Reliability* **24**, 155–180 (2014)
26. Nabuco, M., Paiva, A.: Model-based test case generation for web applications. *International Conference on Computational Science and Its Applications* pp. 248–262 (2014)

27. Rafe, V., Mohammady, S., Cuevas, E.: Using bayesian optimization algorithm for model-based integration testing. *Soft comput* **26**, 3503–3525 (2022)
28. Rensink, A.: The groove simulator: A tool for state space generation. *Applications of Graph Transformations with Industrial Relevance: Second International Workshop* **2**, 479–485 (09 2003)
29. Runge, O., Khan, T., Heckel, R.: Test case generation using visual contracts. *Electronic Communications of the EASST* **58** (2013)
30. Schieferdecker, I., Hoffmann, A.: Model-based testing. *IEEE software* **29**(1), 14–18 (2012)
31. Schmidt, A., Varró, D.: Checkvml: A tool for model checking visual modeling languages. *International Conference on the Unified Modeling Language* pp. 92–95 (2003)
32. Sulaiman, R., Jawawi, D., Halim, S.: Cost-effective test case generation with the hyper-heuristic for software product line testing. *Advances in Engineering Software* **175**, 103335 (2023)
33. Sutton, R.: *Reinforcement learning: An introduction* (2018)
34. Thöne, S.: *Dynamic Software Architectures: A Style-Based Modeling and Refinement Technique with Graph Transformations*. Ph.D. thesis, University of Paderborn (2005)
35. Utting, M., Legeard, B., Bouquet, F., Fournieret, E., Peureux, F., A, A.V.: Recent advances in model-based testing. *Advances in computers* **101**, 53–120 (2016)
36. Yang, Y.: Improve model testing by integrating bounded model checking and coverage guided fuzzing. *Electronics (Basel)* **12**, 1573 (2023)

Disclosure of Interests

The authors have no competing interests to declare that are relevant to the content of this article.