City Research Online

City St George's, University of London

Citation: Ghasemi, S., Asgari Araghi, M., Rafe, V. & Heckel, R. (2026). MoTDeReL: Model-based testing through deep reinforcement learning for software systems specified through graph transformation. Automated Software Engineering, 33(2), 68. doi: 10.1007/s10515-026-00610-3

This is the published 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/37175/

Link to published version: https://doi.org/10.1007/s10515-026-00610-3

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.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Automated Software Engineering (2026) 33:68
https://doi.org/10.1007/s10515-026-00610-3

MoTDeReL: Model-based testing through deep reinforcement learning for software systems specified through graph transformation

Simin Ghasemi[1] · Maryam Asgari Araghi[2] · Vahid Rafe[3] · Reiko Heckel[4]

Received: 14 September 2025 / Accepted: 21 February 2026
© The Author(s) 2026

Abstract

Effective test case generation is crucial for ensuring software correctness, whereas generating high-coverage test suites efficiently remains a challenge. Graph transformations provide a formal way to specify and analyse software systems by modeling system operations as transformation rules and constructing a state-based representation of system behavior. Model-based testing (MBT) often uses model checking over this representation to discover execution paths that satisfy certain test requirements. However, such approaches suffer from severe scalability issues due to the rapid growth of the state space and the high computational cost of exhaustive exploration. While optimization-based approaches mitigate these issues by exploring a reduced portion of the state space, they still struggle to scale effectively. MBT approaches using graph transformation faces the same scalability and often face additional challenges due to the richer structural complexity of graph-based models. However, apart from the behavioral information derived from state transitions, graph transformation systems also encode explicit structural relationships between states and transformation rules. These structural characteristics can be used to define and evaluate test objectives. To exploit this, we propose a novel approach 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 optimize 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 minimizing 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 on well-known case studies demonstrate that our approach achieves higher coverage with reduced computational cost compared to state-of-the-art techniques.

Keywords Model checking-based testing · Test suite generation · Graph transformation systems · Deep reinforcement learning · Neural network

Extended author information available on the last page of the article

Published online: 24 March 2026

1 Introduction

Software testing is a fundamental aspect of the software development life-cycle. It involves assessing a software application to identify defects, errors, or bugs while ensuring it meets the required specifications and functions as intended (Leloudas 2023). Software testing is a resource-intensive process that can take up a substantial portion of both time and budget in the software development life-cycle (Offutt and Ammann 2008). Automation in software testing is gaining popularity due to its ability to save time, reduce costs, and enhance accuracy (Kumar and Mishra 2016). A key aspect of this process is the automation of test case generation, which significantly lowers overall testing expenses and improves the effectiveness of tests to discover errors (Sahoo et al. 2016). The utilization of software models to automate the testing process and minimize associated costs has been a well-established practice for several years (Dias Neto et al. 2007). Model-Based Testing (MBT) (Utting et al. 2016) is an approach that generates and executes test cases from behavioral models of the System Under Test (SUT). By systematically deriving test cases from formal models, MBT enables early detection of faults before implementation, reducing the cost of testing and improving the quality of the SUT. Various strategies have been employed in MBT to generate more effective tests at a lower cost.

Model checking (Mohalik et al. 2014) has been applied to software testing for state-based models. Originally designed as formal verification tools, model checkers (Baier and Katoen 2008) are capable of generating paths that represent counterexamples/witnesses for the violation/satisfaction of a desired property. These paths, which start from the initial state and lead to a state where the property is either violated or verified, can then be utilized as test cases (Gargantini and Heitmeyer 1999; Rayadurgam and Heimdahl 2001). In Model-Checking-Based Testing (MCT), a model checker is employed to take the system model as input and use test objectives as reachability properties to derive paths representing test scenarios. While MCT ensures high coverage of test objectives for small models, its scalability is limited by state-space explosion (Baier and Katoen 2008), the inherent complexity of model checking, and the potential generation of redundant test cases (Fraser et al. 2009b; Villani et al. 2019).

One approach to addressing the state space explosion problem is to use search and optimization methods that heuristically explore only a relevant portion of the state space. Techniques such as Genetic Algorithms (GA) (Haupt and Haupt 2004), Particle Swarm Optimization (PSO) (Eberhart and Kennedy 1995), and Ant Colony Optimization (ACO) (Dorigo et al. 2006) have been utilized in this context. However, there remains potential for further improvement, particularly in terms of eliminating redundant test cases and enhancing both accuracy and efficiency. A key challenge with these techniques is that they generate test cases over multiple iterations, which complicates the identification and removal of redundant test cases produced at different stages. Furthermore, the heuristic functions used for test case selection can exhibit random behavior, occasionally leading to the generation of incorrect or inaccurate test cases. Additionally, further steps are often needed to evaluate test case quality and eliminate redundancy or inaccuracies, introducing extra post-processing efforts that ultimately reduce overall efficiency.

Graphs are powerful tools for representing, visualizing, and analyzing complex software models. Graph Transformation Systems (GTS) (Ehrig et al. 1999) use rules to represent pre-conditions and post-conditions of operations, modeling a system's behavior as a graph transition system, i.e., labelled transition systems (LTS), where graphs serve as states, and graph transformations act as transitions. GTS (Heckel 2006) provide a formal framework for modeling the dynamics of complex systems. They facilitate a step-by-step simulation of a system's behavior, beginning from an initial state and progressing toward a state that meets specified objectives, following a well-defined path. This type of analysis is commonly applied in GTS and offers valuable insights into the operation of complex systems. Within a GTS, there are interdependencies among the rules in each sequence of states. The order of rule execution is critical to maintaining consistency and correctness within the system. Rules may have preconditions and dependencies that must be satisfied before they can be executed. These interdependencies can influence the system's overall behavior and performance. 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 optimization 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.

Recently, the application of deep reinforcement learning (DRL) has been proposed as a solution to mitigate the problem of state space explosion, particularly for searching reachability properties in systems specified through GTS (Mehrabi and Rafe 2022). In this paper, we propose a testing approach for GTS that leverages DRL to explore the state space and identify test cases with the optimal coverage. The process starts at the initial state and selects actions, where actions are selected randomly. As the exploration progresses, the system adapts its strategy based on the reward/penalty received from previous actions, ultimately generating semi-optimal test cases with high coverage and low cost. Additionally, DRL improves over time by continuously learning from the environment. We aim to apply this technique to enhance the efficiency and effectiveness of test case generation in GTS.

Figure 1 shows the proposed conceptual framework. This framework consists of an agent, an environment, and a repository containing a collection of the agent's previous experiences, along with the corresponding rewards or penalties associated with those experiences. Initially, the agent aims to explore an environment that represents a state space. After various explorations, assume the agent currently resides in state s_2 must choose the next action from the available rules: r_1, r_3, or r_4 (Step ❶). To make this decision, the agent refers to its experience repository and, based on the rewards and penalties encountered previously, estimates the value of each action (Step ❷). Consequently, the agent selects action r_1, which has the highest estimated value. Upon executing this action, the agent transitions to state s_6 (Step ❸). This new state, along with the associated reward or penalty, is then stored in the experience repository for future reference (Step ❹).

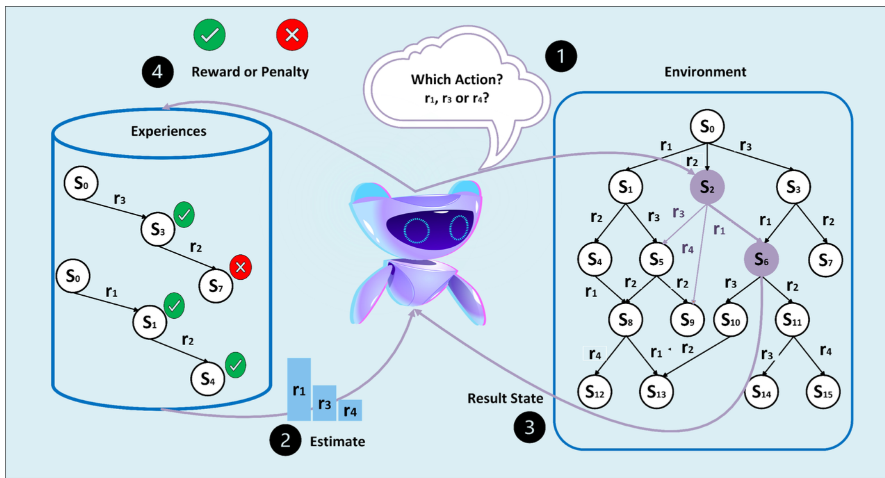


Fig. 1 The proposed conceptual framework

The novel contributions of this paper are as follows:

- We develop a DRL-based approach to generate test cases for systems specified through GTS.
- We devise a search strategy to cover more test objectives.
- The proposed approach enables the storage of network weights, allowing them to be reloaded in subsequent executions, thereby reducing computational costs.
- The approach demonstrates improved efficiently and scalablity, significantly reducing the time required to generates test cases.
- The generated test set is smaller and contains shorter test cases, resulting in lower execution cost.

This paper is an extended version of our ICG2025 conference article (Ghasemi et al. 2025b). While the conference version introduced the core idea of applying DRL to test case generation in GTS, this paper substantially extends both the methodology description and the empirical evaluation. In particular, we provide more detailed explanations of the methodology, incorporate additional recent studies and comparisons, and present a deeper analysis of the results. In particular, we introduce a statistically rigorous analysis of the experimental results, including non-parametric hypothesis testing (Mann–Whitney U test) and effect size analysis using Vargha–Delaney's A_{12}, to assess both statistical significance and practical relevance of the observed differences. Due to the strict space limitations of the conference format, such statistical validation and extended analysis were not included in the earlier version. Additionally, unlike the conference version, this paper provides a complete algorithmic formalization of MoTDeReL, enabling reproducibility and precise analysis. These extensions substantially strengthen the evaluation and demonstrate the scalability and effectiveness of the proposed approach.

The rest of this paper is organized as follows. Section 2 provides some background on different concepts used in this paper. 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 results and the discussions are presented in Section 5. Finally, Section 6 concludes the paper and suggests future work.

2 Background

In this section, we provide an overview of essential concepts that form the foundation of the proposed methodology.

2.1 Graph transformation systems

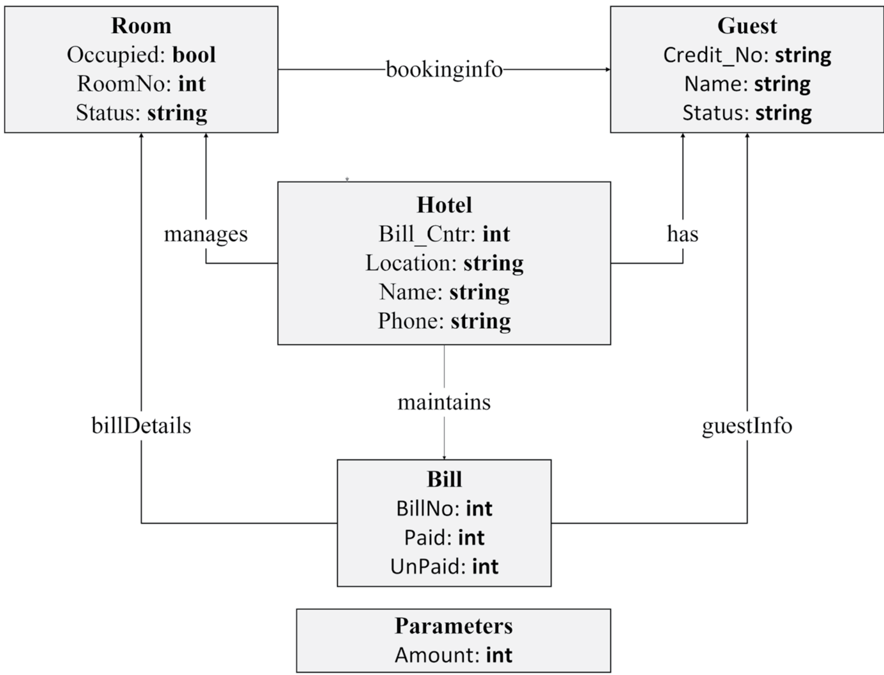
GTS (Ehrig et al. 2004) offer a formal and graphical approach to modeling systems by capturing both their states and behaviors. A GTS is represented by the tuple (TG, HG, R), where:

- TG (Type Graph) represents the system's abstract schema. It is composed of different node types (TGN) and edge types (TGE), with two key functions, src: TGN $\rightarrow$ TGE and trg: TGE $\rightarrow$ TGN, which link nodes to edges.
- HG (Host Graph) represents the initial state of the system and must conform to the structure defined in the type graph.
- R (Rules) defines the transformation rules of the system, where each rule p is denoted by a triple (LHS, RHS, NAC). LHS refers to the left-hand side graph (precondition), RHS to the right-hand side graph (post-condition), and NAC is the negative application condition, specifying configurations where the rule should not apply.

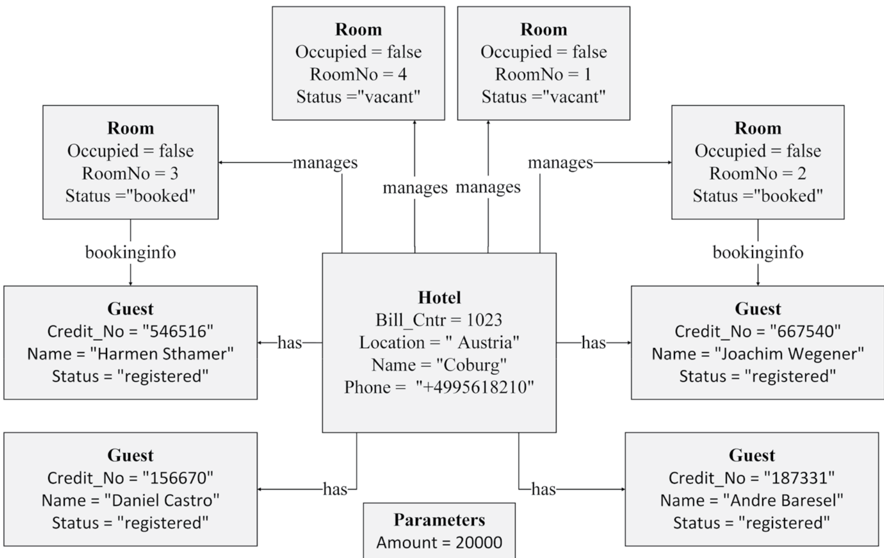
Several popular graph transformation tools, such as ATOM3 (Lara and Vangheluwe 2002), VIATRA2 (Varro and Balogh 2007), AGG (Taentzer 2004), and GROOVE (Rensink 2003), are widely used for modeling and analyzing systems specified with Graph Transformation Systems (GTS). Although each of these tools offers distinct advantages, we selected GROOVE for our approach. This is due to its integrated model checking capabilities, which make it particularly suitable for the automatic exploration and analysis of the state space, as well as for verification and validation purposes (Kastenberg and Rensink 2006).

2.2 Graph transformation system in GROOVE: A running example

The GROOVE toolset is used as the foundation in this paper for modeling and analyzing systems specified by GTS. To illustrate its application, in this section, we present a Hotel Management System (HMS), originally introduced in Heckel et al. (2011). The revised HMS includes multiple rooms and registered guests within a hotel, where each guest has the option to reserve at most one available room. The



(a) Type Graph



(b) Host Graph

Fig. 2 The **a** type graph and **b** host graph for the HMS

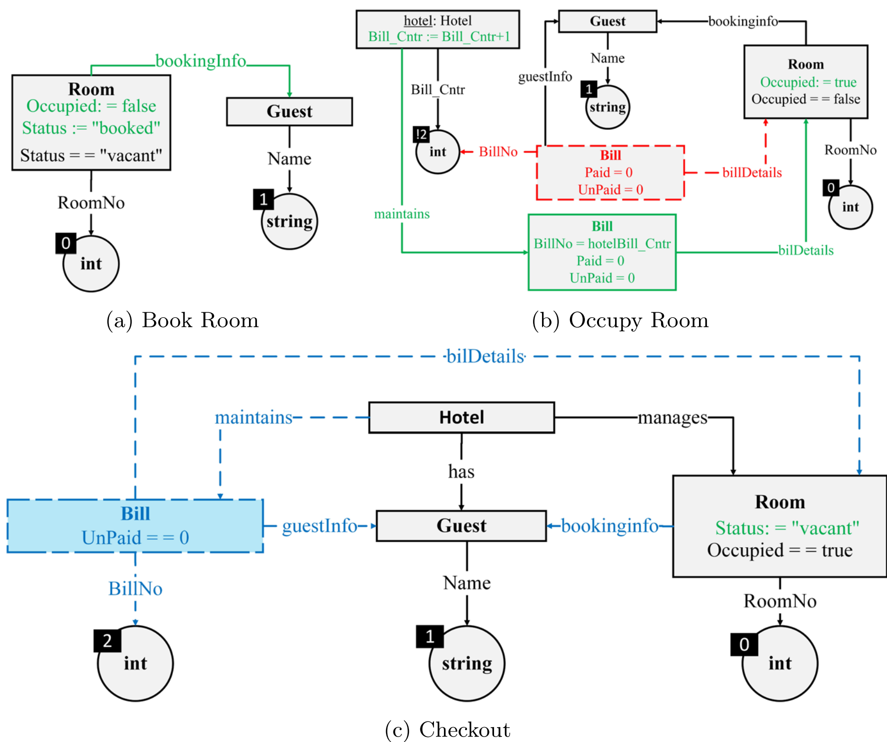


Fig. 3 A subset of the graph transformation rules in HMS **a** Book Room, **b** Occupy Room, **c** Checkout

system automatically generates the corresponding bill, which must be settled before the guest proceeds to checkout.

In this case, the Type Graph and Host Graph of the system are illustrated in Fig. 2 and the corresponding graph transformation rules are presented in Fig. 3. The GROOVE framework provides a visual integration of LHS, RHS, and NAC graphs, with distinct color-coding to differentiate elements (Rensink et al. 2010). Specifically:

- Black highlights common nodes and edges between LHS and RHS.
- Blue represents elements removed from LHS after rule execution.
- Green is used to show newly created elements.
- Red (bold double-border) represents NAC elements.

In practical terms, to apply a transformation rule to a graph, the matching instances of the LHS graph within the host graph are identified, and one instance is replaced by the RHS, provided no instances of the NAC graph exist in the host graph. Figure 4 illustrates a portion of the state space, generated by iteratively applying transformation rules to the initial graph configuration.

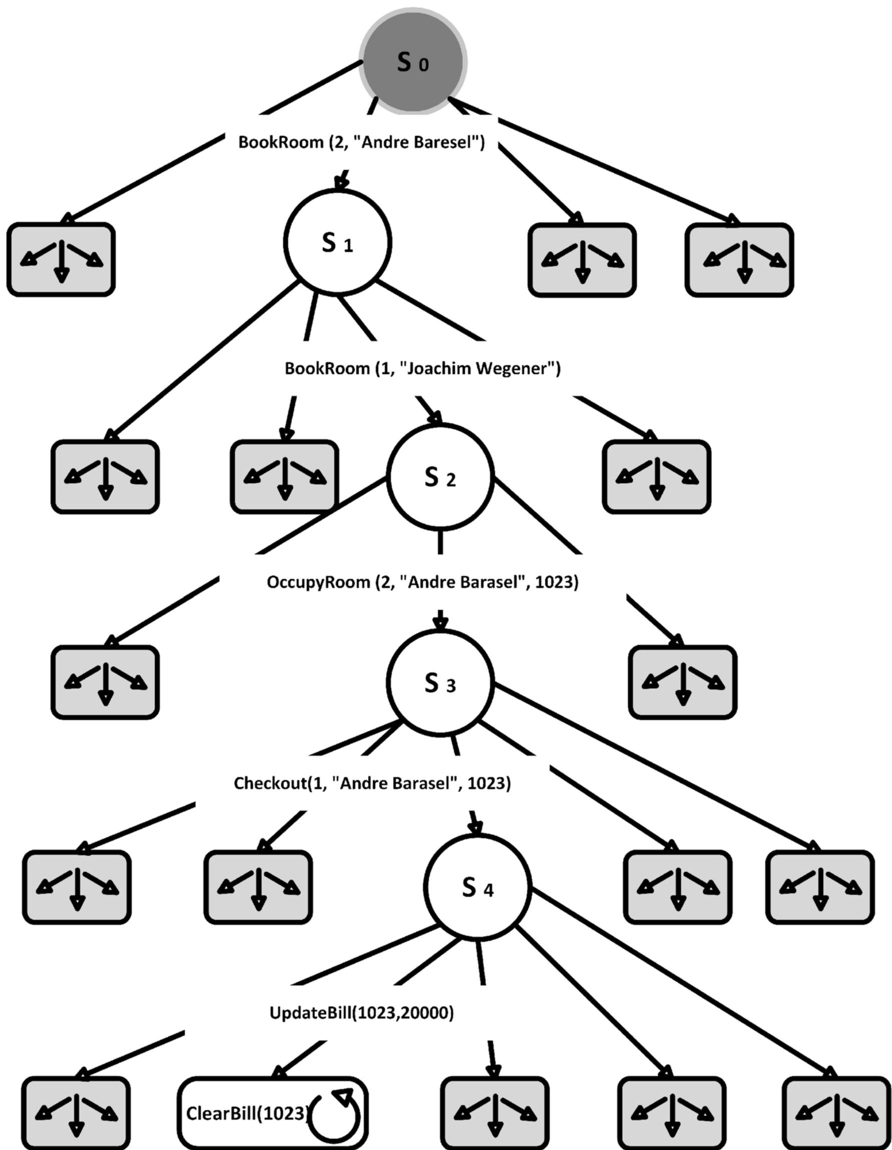


Fig. 4 A segment of the HMS's state space

Figure 3 shows three rules of the HMS, which include Book Room, Occupy Room, and CheckOut. In this figure, a solid black line denotes readers, blue dashed lines symbolize erasers, green solid lines represent creators, and forbidden elements, constituting negative application conditions (NAC), are indicated by red dashed lines.

2.3 Data-flow testing and dependency analysis in GTS

In MBT approaches using GTS, criteria such as state coverage, data-dependency coverage, transition coverage, path coverage, and rule coverage serve as test objectives and direct the testing process (Bahrampour and Rafe 2020). Many studies use data-dependency coverage where 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 (NAC) of the second. Data-flow testing (DFT) is a software testing methodology that focuses on examining how data moves through a program. This approach, first introduced by Herman (1976), selects specific execution paths based on the interaction between data definitions (where a variable is assigned a value) and data uses (where that variable is accessed).

One of the fundamental aspects of DFT is the concept of def-use pairs, which track how a particular variable is defined at one program location and subsequently used at another. The integrity of this flow is maintained as long as no intermediate redefinition disrupts the sequence. Such uninterrupted transitions are often referred to as def-clear paths, as defined in Definition 1.

Definition 1 (Def-Use Pair) A def-use pair, expressed as $\mathtt{du}(\ell_d, \ell_u, v)$, represents a program segment where variable 'v' is defined at location ℓ_d and later utilized at ℓ_u without any redefinition occurring in between. Ensuring these pairs remain intact is essential for validating the correctness of data flow.

To verify def-use relationships in a program, data-flow testing is employed. This method involves designing test inputs that trigger execution paths covering specific def-use pairs. A test case successfully verifies a def-use pair when it enables the program to transition from a variable's definition point to its use without interference.

Definition 2 (Data-Flow Testing) For a given def-use pair $\mathtt{du}(\ell_d, \ell_u, v)$ in program P, data-flow testing aims to construct an input t that forces execution along a path p starting at ℓ_d, where 'v' is defined, and terminating at ℓ_u, where 'v' is used. No redefinition of 'v' must occur along this path to ensure the test properly examines the variable's lifecycle.

Among various data-flow testing strategies, the all-def-use-path criterion (Rapps and Weyuker 1985) is widely regarded as one of the most comprehensive. This criterion ensures that all possible execution paths linking definitions to uses are exercised, validating whether data is correctly propagated throughout a program (Offutt and Ammann 2008). Data-flow coverage is particularly valuable because it targets faults that arise from incorrect interactions between variable definitions and uses, bugs that often remain undetected by control-flow-based techniques such as statement or branch coverage. The proposed approach integrates this coverage metric to enhance test adequacy.

A dependency graph (DG) (Albanese 2019) is a structural representation that captures dependencies among different components in a software system. It visually illustrates how various operations interact, making it a useful tool for identifying

potential conflicts, redundancies, or areas that require refactoring. In GTS, computations are represented as rule-based transformations applied to graphs as illustrated by the HMS rules in Fig. 3.

Since these transformation rules may read, create, or delete graph elements, they often exhibit dependencies, ensuring proper execution order is critical. Some rules may require prior execution of others to function correctly, leading to interdependencies that must be analyzed to determine appropriate coverage criteria. Such interdependencies are illustrated by the HMS running example, where rule applications over graphs conforming to the type graph and initial host graph in Fig. 2 give rise to execution paths in the state space (Fig. 4). To capture these relationships, a DG can be derived from the SUT by examining dependencies between transformation rules. Let R_1 and R_2 be two transformation rules in a GTS. Their dependency or interference relationships can be defined formally based on conditions outlined in Definitions 3 and 4, following the approach in Heckel et al. (2011). These relations are then represented structurally in the dependency graph introduced in Definition 5.

Definition 3 (Dependency) A rule R_1 is considered dependent on rule R_2, represented as $R_1 \prec R_2$, if any of the following conditions hold:

- An edge or node appearing in the left-hand side (LHS) of R_1 is introduced by the right-hand side (RHS) of R_2.
- An edge or node appearing in the negative application condition (NAC) of R_1 is deleted by the RHS of R_2.

Definition 4 (Interference) A rule R_1 is said to interfere with rule R_2, represented as $R_2 \nearrow R_1$, when either of the following conditions is met:

- An edge or node in the LHS of R_1 is deleted by the RHS of R_2.
- An edge or node in the NAC of R_1 is added by the RHS of R_2.

Definition 5 (Dependency Graph) A dependency graph (DG) is a formal structure represented as DG = <G, OP, op, lab>, where:

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

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. For example, in the HMS there is a C_D relation between the BookRoom and CheckOut rules, as shown in Fig. 3, because BookRoom creates bookingInfo edge which Checkout deletes it.

2.4 Reinforcement learning and deep reinforcement learning

Reinforcement learning (RL) is a type of machine learning concerned with how agents learn optimal behaviors through interaction with an environment (Sutton 2018). Unlike supervised learning, where agents learn from labeled examples, RL agents must discover strategies autonomously by trial and error, guided by rewards and penalties. Formally, RL problems are modeled as Markov Decision Processes (MDPs), where an agent observes a *state*, selects an *action*, transitions to a *new state*, and receives a corresponding *reward*. 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 maximize cumulative rewards. Initially, the agent's knowledge about the environment is zero and must explore various actions to identify those leading to the highest long-term rewards. Through interaction with the environment, it acquires information and learns how to achieve the defined goals (Sutton 2018).

Classic RL algorithms, such as Q-learning and SARSA, rely on tabular representations like Q-tables to store the expected utility of state-action pairs (Sutton 2018). While effective in small and discrete environments, these approaches become infeasible as the size or complexity of the state space grows, a problem often referred to as the *curse of dimensionality*. In high-dimensional or continuous spaces, maintaining and updating exhaustive Q-tables is computationally prohibitive, and generalizing learning across unseen states becomes impractical.

Supervised Deep Learning (DL) focuses on training models, particularly deep neural networks, to approximate complex input-output mappings by learning from labeled datasets. A deep learning model receives inputs, processes them through multiple layers of representations, and predicts the corresponding output. The success of deep learning in approximating highly nonlinear functions motivated its integration into reinforcement learning frameworks, enabling agents to handle large and continuous state spaces more effectively.

Deep Reinforcement Learning (DRL) was introduced to overcome these limitations by combining RL frameworks with deep learning techniques. Specifically, DRL leverages deep neural networks, a core component of deep learning (DL), to approximate complex functions, such as value functions or policies, without requiring explicit tabular storage. Instead of mapping each state-action pair manually, the neural network learns to predict Q-values directly from input states, enabling the agent to handle environments with very large or continuous state spaces. Figure 5 illustrates how Deep Reinforcement Learning integrates the decision-making principles of RL with the representational power of DL, enabling agents to operate effectively in environments with large and continuous state spaces.

One of the prominent DRL methods is the Double Deep Q-Network (DDQN) (Hasselt et al. 2016), which extends the original Deep Q-Network (DQN) approach. In standard DQN, the same network is used to both select and evaluate the max Q-value, which can lead to overestimating the true value of actions. DDQN addresses this bias by decoupling the selection and evaluation of actions across two separate neural

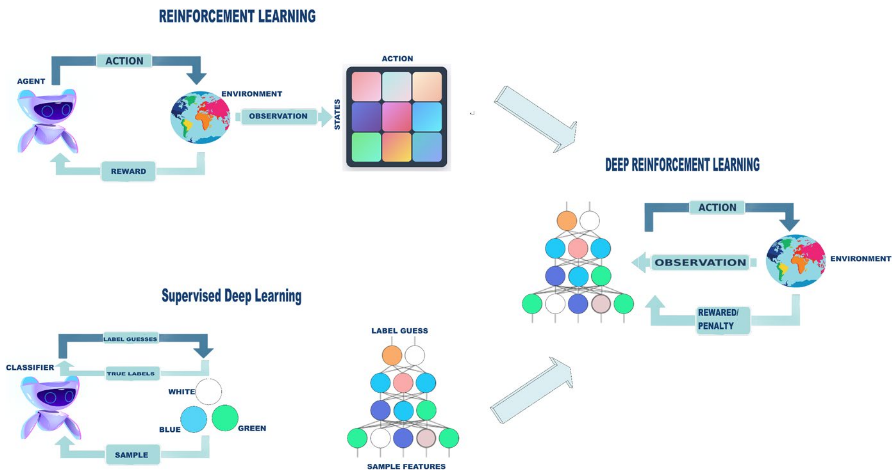


Fig. 5 Relationship between Reinforcement Learning (RL), Supervised Deep Learning (DL), and Deep Reinforcement Learning (DRL)

networks: a primary (online) network and a target network. The primary (online) network selects the action a^* with the maximum predicted Q-value for the next state s' as shown in (1), while the target network then evaluates the value of the selected action, as defined in (2).

$$a^* = \arg\max_a Q_{\text{online}}(s', a) \tag{1}$$

$$Q_{\text{target}}(s', a^*) \tag{2}$$

This separation ensures that the policy updates are based on more accurate value estimates, resulting in more stable learning.

Some important terminology relevant to reinforcement learning includes the following concepts. A *step* refers to a move of an agent involving the execution of an action and the subsequent state change. An *episode* is a sequence of steps from the start to a final state. The *experience replay memory* is a mechanism used to store the agent's past experiences, allowing for randomized sampling during training to improve learning stability. Finally, the *ϵ-greedy algorithm* is an exploration policy where the agent selects the current best-known action most of the time but occasionally chooses a random action with a small probability ϵ.

Deep Reinforcement Learning, by integrating the decision-making capabilities of RL with the generalization power of deep neural networks, offers a scalable and effective framework for solving complex problems. Given the intricate, high-dimensional nature of the state spaces encountered in our work, DRL provides a natural and powerful foundation for the proposed approach.

3 Related work

MBT has been widely explored in the literature, with various approaches leveraging different techniques (Li et al. 2018). To better structure and analyze existing work, we categorize related approaches into the following sections.

3.1 MBT in general

MBT is a software testing technique which involves creating test cases based on a model that represents the system under test and its environment (Utting et al. 2016). Instead of deriving test cases directly from the source code, MBT uses abstract representations such as state machines, flowcharts, or formal specifications to create and execute tests (González et al. 2018). This approach helps ensure systematic testing by using the model to guide test design and execution.

Most MBT approaches are built upon the widely adopted Unified Modelling Language (UML) (Mustafa et al. 2021). For example, Nabuco and Paiva (2014) present an MBT approach for web applications that incorporates UML and Web diagrams, while Jadhav et al. (2023) use finite-state machines to minimize redundancy and enhance efficiency. Rocha et al. (2021) propose a methodology for generating test cases from UML sequence diagrams by transforming them into Extended Finite State Machines (EFSMs). The process involves systematically converting sequence diagrams into EFSMs, from which test cases can be automatically generated and executed using ModelJUnit and JUnit libraries. Their approach facilitates automation and formalization in test case generation, addressing the ambiguity inherent in UML sequence diagrams by assigning precise semantics to their elements. However, the transformation process is limited to UML-based models, which may not generalize well to more complex software representations such as graph transformation systems. While our approach also uses MBT techniques, it distinguishes itself by integrating GTS and optimization, making it a search-based technique rather than a traditional UML-driven method.

Recently, Gómez-Abajo et al. (2025) proposed Wodel-Test, a model-based framework for engineering mutation testing tools tailored to domain-specific languages. While this work focuses on systematic mutation generation to assess test adequacy, MoTDeReL addresses test generation itself by learning optimal exploration strategies in systems with large state spaces. The two approaches are complementary: mutation-based frameworks such as Wodel-Test could benefit from DRL-guided test generation to improve scalability, while mutation analysis could be integrated into the reward function of MoTDeReL to further strengthen fault detection.

3.2 MBT using model-checking techniques

Model Checking has emerged as a powerful technique for automated test case generation in MBT. By systematically exploring system models, it verifies whether certain properties hold generate test cases that ensure high structural and logical coverage. In this approach, testing requirements are typically specified as reachability properties, and counterexamples from violated properties are used as test paths. The integration

of MBT with Model Checking provides a systematic and exhaustive approach to test generation by verifying system properties through complete state space exploration. Fraser et al. (2009a) present a survey of various model-checking-based testing approaches, analyzing their effectiveness across different contexts. In Gönczy et al. (2007), 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. Yang (2023) proposes a bounded model checking framework combined with coverage-guided fuzzing to enhance test case generation. This approach leverages Bounded Model Checking (BMC) to generate targeted test cases within a predefined execution depth, improving fault detection efficiency. BMC has gained prominence as an alternative to Binary Decision Diagram (BDD)-based symbolic model checking, addressing the state space explosion problem by encoding the verification problem as a propositional satisfiability (SAT) problem (Biere 2021). Unlike traditional symbolic model checking, which explores the full system state space, BMC systematically searches for counterexamples within a predefined bound. If a counterexample is found, it serves as a test case revealing potential system violations. The technique has been successfully integrated into various verification tools and has demonstrated efficiency in hardware verification and software model checking. However, BMC does not fully eliminate state space explosion, as increasing the bound leads to exponential growth in the generated SAT clauses. Additionally, Model Checking techniques have been successfully applied to Function Block Diagram (FBD) programs, commonly used in safety-critical systems (Enoiu et al. 2014). The research focused on transforming FBD programs into formal representations suitable for Model Checking, allowing for the automatic generation of test cases that satisfy structural coverage criteria, including decision coverage (DC) and modified condition/decision coverage (MC/DC). The approach demonstrated that using Model Checking, specifically with tools like Uppaal, facilitates the systematic generation of test sequences, ensuring that logic-based test obligations are met efficiently. While these studies highlight the effectiveness of Model Checking in test case generation, MCT methods often face scalability limitations, especially when applied to complex systems. The practical application of this technique to industrial software systems often faces scalability issues due to the large state space of real-world programs. Since model checkers were originally designed for verification rather than test generation (Fraser et al. 2009a). To address this, heuristic search techniques and optimizations have been explored to improve the efficiency of test generation, which is discussed in the following section.

3.3 MBT using optimization and search-based techniques

MBT has been extensively enhanced using search-based techniques, which leverage heuristic search algorithms to explore large state spaces and generate effective test cases. Search-Based Testing formulates test generation as an optimization problem, where the goal is to find a set of test cases that achieve high coverage while minimizing computational cost.

In Sulaiman et al. (2023), the authors propose ISR-MCF, which employs three search algorithms—NSGA-II-LLH, SPEA2-LLH, and PSO-LLH—along with reinforcement learning to optimize the selection of test cases. The study highlights that NSGA-II-LLH outperforms other methods in generating cost-effective and high-coverage test cases.

Several approaches utilize GTS for MBT. A variety of evolutionary, heuristic, and AI-based algorithms have been used in MBT, such as Bee Colony, Genetic Algorithms (GA), Particle Swarm Optimization (PSO), Bat Algorithms (BA), Gravitational Search Algorithms (GS), data mining, Bayesian optimization Algorithm (BOA), Beam search Algorithm (Bahrampour and Rafe 2020; Kalaee and Rafe 2019; Ghasemi et al. 2025a; Asgari Araghi et al. 2024; Rafe et al. 2022; Asgariaraghi et al. 2019). These techniques explore the state space partially to extract high-coverage test paths. In Kalaee and Rafe (2019), the authors use search-based algorithms, including Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Bat Algorithm (BA), Gravitational Search Algorithm (GSA), and a hybrid algorithm combining GA and PSO (HGAPSO) to generate test cases from models specified through GTS. They defined the test generation as an optimization problem, aiming to maximize the static data dependency of test paths. The implementation, carried out using Groove, demonstrates that HGAPSO achieves highest test coverage. However, the approach faces challenges in defining test objectives statically and is not suitable for complex systems. To overcome these limitations, Bahrampour and Rafe (2021) extend HGAPSO to a Memetic Algorithm (MA) to enhance robustness testing. Their primary goals of this approach are to maximize test coverage while minimizing testing costs. The effectiveness of the strategy is evaluated using mutation analysis at model level. The experimental results demonstrate that the approach outperforms the existing ones in fault detection. Further, in Bahrampour and Rafe (2020), the authors introduce another approach for testing software models specified through GTS. Their methodology employs GA to maximize coverage across rule, state, transition, and data-flow criteria. Results suggest that data-dependency coverage outperforms other criteria, particularly in complex systems with numerous states and interrelated rules. However, this approach also has drawbacks, including high computational complexity and excessive length of generated tests. In Ghasemi et al. (2025a), the authors explore the use of ACO to generate whole test suites rather than individual test cases. Their findings show that this method produces superior test suites in terms of size and coverage. However, its primary drawback is the high computational cost associated with generating the state space and finding optimal test paths. Another contribution in this domain is the use of the Bayesian Optimization Algorithm (BOA) for model-based integration testing (Rafe et al. 2022). This approach tackles the challenges of state space explosion and redundant test case generation by leveraging Bayesian Optimization to guide test suite generation. The authors define test cases as executable sequences of system functions, ensuring that the paths cover essential data flow dependencies. A recent study (Asgari Araghi et al. 2024) proposes an approach that uses two data mining algorithms, Apriori and FP-Growth, to avoid exhaustive state space exploration. This method initially employs a Breadth-First Search (BFS) strategy to explore a portion of the state space, followed by the application of data mining algorithms to extract frequent patterns that guide the exploration process. Despite its

advantages, this approach has two notable limitations: it relies on static dependencies to evaluate test case fitness and requires an additional reduction phase to minimize the test suite, significantly increasing computational complexity.

For our comparison, we have selected HGAPSO (Kalaee and Rafe 2019), GA (Bahrampour and Rafe 2020), WholeAcoT (Ghasemi et al. 2025a), ISR-MCF (Sulaiman et al. 2023), and the FP-Growth approach proposed in Asgari Araghi et al. (2024). These approaches were chosen not only because they align with our study's objectives and their implementations are accessible but also because they represent more recent advancements in the field.

We have excluded BOA (Rafe et al. 2022) from direct comparison, as Asgari Araghi et al. (2024) has already demonstrated superior performance over this method. Given that Asgari Araghi et al. (2024) outperforms BOA, and is a more recent approach, comparing against it ensures a more relevant and up-to-date evaluation.

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.

Figure 6 illustrates our approach using two flowcharts, which we divided to simplify the explanation. According to Fig. 6(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. 6(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

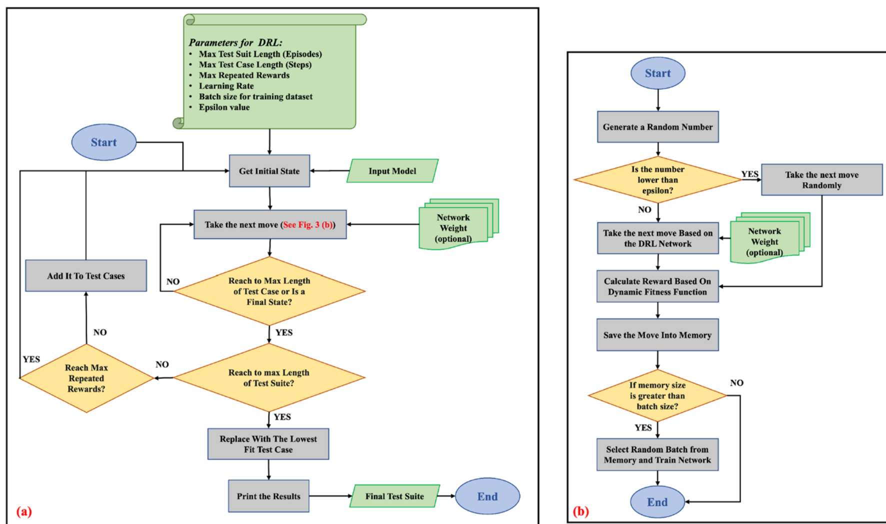


Fig. 6 The general flowcharts of the proposed approach **a** Main flowchart **b** Second flowchart

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. 6(b)), the approach resumes with the steps in the first flowchart (Fig. 6(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.

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 (Mehrabi and Rafe 2022). 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. These dependencies instantiate edges of the dependency graph defined in Definition 5, restricted to the executed transformation path, which corresponds to a sequence of rule applications in the graph transition system, as illustrated by the HMS state-space fragment in Fig. 4. 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 2 computes data-flow coverage for a given transformation path by extracting valid def-use pairs with respect to Definition 2. The input is a path $p = \langle t_0, \ldots, t_k \rangle$ of rule application; the output is the set of dependencies *Deps*, each represented as a triple (r_d, r_u, rel), where r_d is the defining rule, r_u the using rule, and *rel* the dependency type (e.g., C_R, C_D, U_R), as specified in Definition 5.

To enable efficient computation, we first derive, once for the entire grammar (each GTS rule), a global map of *Event signatures*. These signatures describe, abstractly, which elements the rule reads, creates, updates, or deletes and are computed by Algorithm 1. An event signature is an abstract description of the operations a rule performs on its pattern elements, expressed as a triple (*op*, *pe*, *locus*), where:

- *op* $\in \{C, U, R, D\}$ denotes the type of operation: *Create*, *Update*, *Read*, or *Delete*.
- *pe* represents a pattern element of the rule (a node or edge in the graph transformation rule).
- $locus \in \{\mathrm{LHS}, \mathrm{RHS}, \mathrm{NAC}\}$ indicates the structural context in which the element occurs.

For every rule r, the Algorithm 1 enumerates its pattern elements and classifies them according to structural position: (i) every element in the LHS contributes a read access; (ii) elements appearing only in the RHS are marked as creations; (iii) elements present in both sides but structurally modified are updates; (iv) elements present in the LHS but absent from the RHS are deletions. Additionally, negative application conditions (NACs) contribute read accesses. The resulting mapping Events is fixed throughout test generation and is implicitly referenced by Algorithm 2.

Algorithm 1 PrecomputeRuleEventSignatures.

```
1: Input: Set of rules 𝓡 from the GTS model
2: Output: Events: mapping from each rule r ∈ 𝓡 to a set of triples (op, pe, locus)
3: for all r ∈ 𝓡 do
4:     Events[r] ← ∅
5:     for all pe ∈ r.LHS do
6:         Events[r] ← Events[r] ∪ {(R, pe, LHS)}          ▷ Read access on LHS
7:     end for
8:     for all pe ∈ r.NACs do
9:         Events[r] ← Events[r] ∪ {(R, pe, NAC)}          ▷ NAC-read access
10:    end for
11:    for all pe ∈ r.RHS do
12:        if pe ∉ r.LHS then
13:            Events[r] ← Events[r] ∪ {(C, pe, RHS)}      ▷ New element: Create
14:        else if pe is modified then
15:            Events[r] ← Events[r] ∪ {(U, pe, RHS)}      ▷ Updated element
16:        end if
17:    end for
18:    for all pe ∈ r.LHS \ r.RHS do
19:        Events[r] ← Events[r] ∪ {(D, pe, LHS)}          ▷ Deleted element
20:    end for
21: end for
22: return Events
```

Algorithm 2 begins by initializing *Deps* as empty and setting up a mapping *lastDef* to store the most recent definition for each runtime element. The algorithm iterates over all transformations in the path (lines 5–7). For each rule application, its pattern events are lifted to runtime by binding pattern elements to matched elements (lines 8–11), producing U_i. Next, each runtime event is analyzed to identify possible uses (lines 13–26). If an element has a prior definition in *lastDef*, the algorithm checks the *Def-Clear* property, ensuring no redefinition occurs between the original definition and the current use. If this property holds, the pair of defining and using rules is classified into one of relation types: C_R, C_D, C_U, U_R, U_U, or D_N (lines 28–43). After processing uses, any Create, Update, or Delete in the current rule updates *lastDef* (lines 47–50). Finally, the set *Deps* of all valid dependencies is returned. This method ensures precise identification of definition–use relations for data-flow-based coverage in model based testing.

Algorithm 2 ComputeDataFlowCoverage.

```
1: Input: Path prefix p = ⟨t_0, ..., t_k⟩; precomputed map Events (Algorithm 1)
2: Output: Deps = set of data-flow pairs (r_d, r_u, rel) with rel ∈ {C_R, C_D, C_U, U_R, U_U, D_N}
3: Deps ← ∅
4: lastDef ← ∅                                   ▷ map runtime element e ↦ (j, defOp, defRule)
5: for i ← 0 to k do
6:    r_u ← t_i.rule; m_i ← t_i.match
7:    U_i ← ∅                                     ▷ lift pattern events to runtime for t_i
8:    for all (op, pe, locus) ∈ Events[r_u] do
9:       if m_i binds pe to runtime element e then
10:         U_i ← U_i ∪ {(op, e, locus)}
11:      end if
12:   end for      ▷ Uses: R, U, D may close a def–use pair with the last def of the same element
13:   for all (op, e, locus) ∈ U_i do
14:      if op ∈ {R, U, D} and e ∈ lastDef then
15:         (j, defOp, r_d) ← lastDef[e]
16:         // Def-clear: ensure no C/U/D on e between j and i
17:         defClear ← true
18:         for k′ ← j+1 to i−1 do
19:            r′ ← t_k′.rule; m′ ← t_k′.match
20:            for all (op′, pe′, _) ∈ Events[r′] do
21:               if m′ binds pe′ to e and op′ ∈ {C, U, D} then
22:                  defClear ← false; break
23:               end if
24:            end for
25:            if not defClear then break
26:            end if
27:         end for
28:         if defClear then
29:            usedNAC ← (op=R ∧ locus=NAC)
30:            // Relation mapping inline
31:            if defOp=C and op=R and not usedNAC then
32:               Deps ← Deps ∪ {(r_d, r_u, C_R)}
33:            else if defOp=C and op=D then
34:               Deps ← Deps ∪ {(r_d, r_u, C_D)}
35:            else if defOp=C and op=U then
36:               Deps ← Deps ∪ {(r_d, r_u, C_U)}
37:            else if defOp=U and op=R and not usedNAC then
38:               Deps ← Deps ∪ {(r_d, r_u, U_R)}
39:            else if defOp=U and op=U then
40:               Deps ← Deps ∪ {(r_d, r_u, U_U)}
41:            else if defOp=D and usedNAC then
42:               Deps ← Deps ∪ {(r_d, r_u, D_N)}
43:            end if
44:         end if
45:      end if
46:   end for                                     ▷ Definitions: C/U/D at t_i (re)define e
47:   for all (op, e, _) ∈ U_i do
48:      if op ∈ {C, U, D} then
49:         lastDef[e] ← (i, op, r_u)
50:      end if
51:   end for
52: end for
53: return Deps
```

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 description, we define the reward function to measure the data-flow coverage achieved by the current sequence of applied rules in the model transformation process. The reward function implemented in MoTDeReL is shown in Algorithm 3. The algorithm takes as input a GTS model of the system under test, the current state S_t in the state space, and a list of previously computed rewards (*allRewardsUntilNow*). The goal is to compute the reward *R* for the current state.

First, the algorithm retrieves the path P_t from the initial state S_0 to the current state S_t, which corresponds to the sequence of applied transformation rules. Next, it computes the data-flow dependencies among the applied rules using ComputeDataFlowCoverage Algorithm 2, resulting in a set of dependent rule pairs. Redundant pairs are then removed to avoid counting duplicates. The size of the remaining dependency set is considered as the coverage value.

To discourage repetitive exploration, the algorithm checks if the current coverage value has already appeared in *allRewardsUntilNow*. If so, a penalty is applied by subtracting the number of previous occurrences from the coverage value. Otherwise, the reward equals the current coverage. Finally, the computed reward *R* is returned.

Algorithm 3 The reward function.

1: **Input:** Model as a GTS model of the system under test, S_t as the current state of the Model, *allRewardsUntilNow* as list of previously retrieved rewards
2: **Output:** *R* as the reward of S_t
3: P_t ← ExtractPathFromState(S_t) ▷ P_t is the sequence of rules applied from S_0 to S_t
4: *dependencies* ← ComputeDataFlowCoverage(P_t) ▷ Extract all pair of dependent rules from P_t (Algorithm 2)
5: *dependencies* ← DeleteRedundant(*dependencies*) ▷ Delete repetitive pairs from dependencies
6: *coverage* ← Size(*dependencies*)
7: **if** *coverage* is in *allRewardsUntilNow* **then**
8: *R* ← *coverage* − count(*coverage in allRewardsUntilNow*) ▷ The number of times the coverage value is repeated in *allRewardsUntilNow* deducted from coverage as penalty
9: **else**
10: *R* ← *coverage*
11: **end if**
12: **return** *R*

The algorithm first computes all definition–use dependencies along the path by invoking COMPUTEDATAFLOWCOVERAGE(p) (line 4). These dependencies correspond to pairs of rules connected by one of the data-flow relation types: C_R, C_D, C_U, U_R, U_U, or D_N. Since this helper already ensures uniqueness and enforces the *Def-Clear* property, no further processing is typically required. Finally, in line 4, the reward value *R* is computed as the cardinality of the dependency set. This value directly represents the degree of data-flow coverage achieved by the current state and serves as the fitness measure guiding the reinforcement learning process.

4.4 Feature selection

Feature selection is the process of designing a model that is easier to understand and interpret by humans (Duboue 2020). 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 initialized 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. 7 where the agent is allowed to take 4 steps, resulting in 4 inputs for the neural network. Suppose the agent selects the rules in the following sequence: 2, 1, 2, and 1 (Fig. 7a). 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, matching the order of rule selection shown in Fig. 7b. 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.

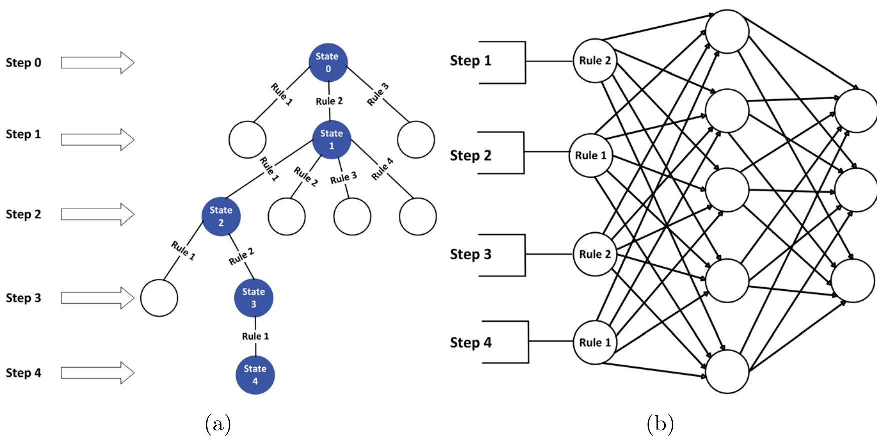


Fig. 7 An example for feature selection in MoTDeReL

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 MoTDeReL approach leverages a Double Deep Q-Network (DDQN) for adaptive test case generation. The entire process is outlined in Algorithm 4. This algorithm operates on a Graph Transformation System (GTS) model of the system under test and iteratively explores the state space to generate diverse and high-coverage test cases. For each episode, the agent starts from the initial state and performs a series of steps, each corresponding to applying a rule (transition) on the state space. The exploration is guided by Q-learning with experience replay, target network updates, and an ϵ-greedy policy for balancing exploration and exploitation.

Algorithm 4 describes the overall MoTDeReL approach for generating test cases using a DDQN-based exploration strategy. The algorithm receives as input the GTS model of the system under test, the number of episodes (*M*), the number of steps per episode (*N*), the maximum number of selected test cases (*MST*), and several reinforcement learning parameters such as ϵ (exploration rate), γ (learning rate), maximum steps for target network update (*MSUTN*), maximum repetition threshold (*MFR*), replay memory capacity, batch size (β), and an ϵ reduction threshold. The output is a set of selected test cases with the highest coverage or fitness, each represented as a sequence of applied rules. As shown in line 3, the algorithm begins by initializing the required data structures, including the electedTestCases, replay memory $\mathcal{D}$ and the two Q-networks (Q_θ and $Q_{\theta'}$). For each episode (line 4), the initial state of the GTS model is selected (s_t), and an empty array *allRewardsUntilNow* is created to track accumulated rewards (achieved for different test cases) for stagnation detection. Within each episode, the algorithm performs up to *N* steps (line 7). for each step of the episode, it traverses a transition of the state space or performs an action. At every step, the procedure *NextMoveExecution* (Algorithm 5) is invoked to determine the next state and update the Q-network based on the selected action (line 10). If the returned state is null or terminal, the episode ends early, otherwise a movement on the state space will occur, representing an application of a rule which changes the current state to the next one. After completing the steps of an episode, the algorithm extracts the path (P_t) from the initial state to the final state (line 15). This path represents a candidate test case. The *MST* determines the total number of test cases that are ultimately generated. To maintain best test cases and limit the size of the final test set, the algorithm checks if the maximum allowed number of test cases (*MST*) is reached (line 18). If so, the weakest test case in the *selectedTestCases* set is replaced by the new candidate; otherwise, the new test case is added to *selectedTestCases* (lines 19–21). After completing all episodes, the algorithm returns the set of selected test cases (line 24).

Algorithm 4 The MoTDeReL approach.

1: **Input:** Model as a GTS model of the system under test, N as Steps, M s Episodes, MST as Maximum number of selected test case, ϵ, $MSUTN$ as Maximum steps for update target network, MFR as Maximum number of fitness repetition for a test case, γ as learning rate, capacity of memory N, β as batch size, threshold for epsilon reduction
2: **Output:** SelectedTestCases
3: Initialize SelectedTestCases = {}, experience replay memory $\mathcal{D}$ to capacity N, primary network Q_θ, and target network $Q_{\theta'}$ with weights 0
4: **for** episodes = 1 to M **do**
5: s_t = initial state of Model
6: Initialize allRewardsUntilNow = { }
7: **for** step = 1 to N **do**
8: **if** s_t is null **then**
9: **go** to the next episode
10: **end if**
11: **if** s_t is a Terminal state **then**
12: **break**
13: **end if**
14: s_{t+1} = **NextMoveExecution**(Model, ϵ, $MSUTN$, MFR, allRewardsUntilNow, step, γ, $\mathcal{D}$, β, Q_θ, $Q_{\theta'}$, threshold, s_t) // ▷ Algorithm 5
15: $s_t = s_{t+1}$
16: **end for**
17: P_t = ExtractPathFromState(s_t) ▷ P_t is the sequence of rules applied from S_0 to S_t
18: **if** Size(SelectedTestCases) > MST **then**
19: Replace lowest-coverage test case with P_t
20: **else**
21: Add P_t to SelectedTestCases as a new testcase
22: **end if**
23: **end for**
24: **return** SelectedTestCases

As seen in line 14 of Algorithm 4, the procedure *NextMoveExecution* is invoked at each step to determine the next state and update the Q-network. This procedure is detailed in Algorithm 5. It applies an ϵ-greedy strategy to select the next action (rule), executes the action to explore the next state, calculates its reward, and performs DDQN updates based on experience replay and target network synchronization. In Algorithm 5, the next move execution on state space is presented. At first, the input includes the GTS model of the system under test, *epsilon* parameter (ϵ), 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, the *allRewardsUntilNow* array, the learning rate (γ), experience replay memory $\mathcal{D}$, batch size β, and two Q-networks (Q_θ and $Q_{\theta'}$).

The user specifies the initial value for *epsilon* parameter of DRL. The *MSUTN* parameter specifies the number of steps required to update the target network, while *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 or null if termination conditions are met.

The algorithm begins by extracting the applicable rules of the current state (line 3) denoted as *rules*. Then, according to the ϵ-greedy policy, 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. It either selects a random rule or predicts the best action using the primary Q-network (lines 4–9). If the rand value is lower than the *epsilon*, the action (rule) will be selected randomly (line 6). Otherwise, it is predicted using the main neural network (line 8). The selected action is executed, leading to the next state (line 10), and the reward of applied action on the current state is computed via a reward function (Algorithm 3) (line 11). Experience tuples (s_t, a_t, r_t, s_{t+1}) are stored in the replay memory $\mathcal{D}$(line 13).

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. If enough samples are available, a batch is sampled to update the Q-network using gradient descent with respect to the network parameters (lines 14–19). Here, the target value refers to the value used to update the Q-value of a state-action pair during training. The target Q-network is synchronized periodically based on *MSUTN* (lines 20–22). If the steps value equals *MSUTN*, the weights of the two neural networks (main and target) are synchronized. Finally, ϵ is decayed (line 25). In line 27, 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 30.

Algorithm 5 NextMoveExecution.

1: **Input:** Model as a GTS model of the system under test, ϵ, $MSUTN$ as Maximum steps for update target network, MFR as Maximum number of coverage repetition for a test case, allRewardsUntilNow array, step as current step of exploration, γ as learning rate, experience replay memory $\mathcal{D}$, batch size β, primary network Q_θ, target network $Q_{\theta'}$, threshold for epsilon reduction, s_t as current state
2: **Output:** Next state s_{t+1} or null
3: Extract rules as the list of applicable rules for s_t
4: Generate a random number as $rand$
5: **if** $rand < \epsilon$ **then**
6: Select random action a_t from 0 to count (rules)
7: **else**
8: Select $a_t = \arg\max Q_\theta(s_t)$
9: **end if**
10: Execute a_t, explore next state s_{t+1}
11: Calculate r_t = RewardFunction(Model, s_{t+1}, allRewardsUntilNow) ▷ (Algorithm 3
12: Store r_t in allRewardsUntilNow
13: Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
14: **if** $|\mathcal{D}| \geq \beta$ **then**
15: batch = Sample random batch from $\mathcal{D}$
16: **for** each (s_j, a_j, r_j, s_{j+1}) in batch **do**
17: $Target = r_j + \gamma Q_{\theta'}(s_{j+1}, \arg\max_a Q_\theta(s_{j+1}, a_j)$
18: Perform a gradient descent step on $(Target - Q_\theta(s_{j+1}, a_j))^2$
19: **end for**
20: **if** step = $MSUTN$ **then**
21: Reset $Q_{\theta'}$ weights = Q_θ
22: **end if**
23: **end if**
24: **if** $\epsilon > threshold$ **then**
25: Reduce ϵ
26: **end if**
27: **if** count(r_t in allRewardsUntilNow) > MFR **then**
28: **return** null
29: **else**
30: **return** s_{t+1}
31: **end if**

5 Evaluation

In this paper, we propose an approach based on deep reinforcement learning to test software models specified using GTS. The MoTDeReL approach has two main objectives: first, to generate a TS that maximizes the coverage of the test objectives, and second, to develop a cost-aware test generation methodology that addresses the state space explosion problem in complex systems. To evaluate our solution against these objectives, we implemented MoTDeReL using the Groove toolset [1] 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 FP-Growth (Asgari Araghi et al. 2024), HGAPSO (Kalaee and Rafe 2019), GA (Bahrampour and Rafe 2020), WholeAcoT (Ghasemi et al. 2025a), and ISR-MCF (Sulaiman et al. 2023). Hence, we compared our proposed approach with these approaches.

In addition to comparing against state-of-the-art techniques, it is also standard practice to evaluate a test generation method against a baseline, such as Random Testing (RT). RT operates by randomly exploring the state space without any strategic guidance, and while simple, it serves as a common reference point for assessing the relative improvements offered by more advanced methods.

Notably, prior studies (Asgari Araghi et al. 2024; Kalaee and Rafe 2019) have shown that their proposed techniques consistently outperform RT. In our evaluation, we benchmarked our approach against the best-performing methods from these works. As demonstrated in the following section, our method achieves superior coverage, indicating not only an advancement over the selected approaches but, by implication, an improvement over RT as well. Due to space constraints, we do not include RT's results explicitly, as its limited performance is already well-documented in the literature.

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

5.1 Case studies

For the evaluation of our proposed approach, we selected six well-established case studies, each recognized for their significant state spaces and the potential to encounter the state explosion problem: the Dining Philosophers Problem (DPs) (Schmidt and Varró 2003), the Hotel Management System (HMS) (Heckel et al. 2011), the Bug Tracker System (BTS) (Runge et al. 2013), the Online Shopping System (OSS) (Engels et al. 2006), the Travel Agency System (TAS) (Thöne 2005), and the Scanflow Cash Register Protocol (SCRP) (Bruijn 2013). The case studies have different sizes but all of them can encounter state explosion due to the use of a fairly large start graph. Although the DPs and HMS case studies are relatively small, they are

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

Table 1 Case studies specifications

Case Study	DPS	HMS	BTS	OSS	TAS	SCRP
#Rule	6	7	34	20	43	63

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. Our objective with these diverse benchmarks is to rigorously assess the effectiveness of our methodology in handling such challenging and complex scenarios.

Specifically, the Dining Philosophers Problem (DPs) serves as a classic concurrency control challenge, modeling how multiple processes (philosophers) share limited resources (forks) without deadlock or starvation. The Hotel Management System (HMS) simulates the intricate operations of a hotel, including booking, check-in/out, and room service, representing a real-world system with numerous interacting components and states. The Online Shopping System (OSS) facilitates customer interactions for online product purchases and credit card payments, encompassing various transactional workflows. The Bug Tracker System (BTS) is designed to manage the lifecycle of software defects throughout development projects, involving states such as submission, assignment, resolution, and verification. The Travel Agency System (TAS) handles flight and hotel bookings for clients, dynamically adapting to their preferences and budgetary constraints. Finally, the Scanflow Cash Register Protocol (SCRP) models the communication and transaction logic of a modern cash register system, highlighting complex protocol interactions.

Table 1 presents the specifications of these chosen case studies, including the number of rules defining their behavior, serving as a metric of their relative sizes. These models, all based on practical real-world scenarios, are particularly valuable due to their inherent large state spaces, which make them excellent candidates for evaluating the efficacy of our test case generation techniques in navigating and covering vast solution landscapes.

5.2 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.

Given the heuristic nature of the algorithms introduced in this study, each experiment was executed ten times to ensure consistency and reduce the impact of randomness. The average of these runs was then used for further analysis. To assess the performance and reliability of the proposed method, we employed the Mann-Whitney U test, a non-parametric statistical test commonly used for comparing two independent samples. In addition, we utilized the effect size measure $\hat{A}_{12}$, proposed by Vargha and Delaney (Arcuri and Briand 2011), to quantify the magnitude of differences between methods.

The $\hat{A}_{12}$ statistic represents the probability that a randomly selected observation from one method will have a higher value than a randomly selected observation from another. A value below 0.5 suggests that the second method is more likely to produce better results, a value of 0.5 indicates no difference, and a value above 0.5 favors the first method. Alongside this, we evaluated statistical significance using the $P-value$. A $P-value$ less than 0.05 is considered evidence of a statistically significant difference between the two methods. If the $P-value$ exceeds this threshold, further analysis is needed to draw reliable conclusions.

The experiments were conducted on a PC with an Intel Core ™ 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 Section 2.3, 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 which determine the maximal number and length of TCs, respectively, are variable for each case study.

Algorithm 5 comes with its own set of hyperparameters that must be configured. Below, we outline the choices made for these parameters. Unless explicitly specified, we rely on the default settings. A potential avenue for future research could involve a more in-depth investigation into how hyperparameter selection impacts the final learning outcomes and whether these parameters should be specifically tailored to the SUT.

The ϵ parameter 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 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 fast convergence to a global optimum. Additionally, we chose RMSProp as the optimizer to regulate the learning rate effectively.

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 Section 4.3, we set the discount factor to 0.95. This value assists the agent in making decisions. Additionally, we set the experience replay memory size 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. Moreover, 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 2.

We determined different values for step and MST parameters with respect to each case study. These values were chosen to generate applicable TSs. In Table 3, the step and MST values are listed for each case study.

Table 2 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 optimizer	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 3 Step and MST for each case study

Case Study	MST	Step
DPs	1	20
HMS	1	20
BTS	7	15
OSS	4	30
TAS	7	15
SCRP	10	30

5.3 Experimental results

This section presents the empirical evaluation of our proposed approach, MoTDeReL. To assess its effectiveness, we compare it against several existing test generation strategies, including FP-Growth (Asgari Araghi et al. 2024), HGAPSO (Kalaee and Rafe 2019), GA (Bahrampour and Rafe 2020), WholeAcoT (Ghasemi et al. 2025a), and ISR-MCF (Sulaiman et al. 2023). All approaches were executed under consistent experimental settings, and each run was repeated ten times to ensure statistical reliability.

The evaluation is structured around the following research questions:

- **RQ1:** To what extent does MoTDeReL improve coverage compared to the state-of-the-art techniques?
- **RQ2:** How effectively does MoTDeReL reduce test size and overall testing cost?
- **RQ3:** How does MoTDeReL perform in terms of test generation time?

The remainder of this section presents the results and provides a detailed analysis to address each research question.

RQ1: How much has MoTDeReL improved the coverage score compared to the state-of-the-art?

Table 4 Comparison of achieved coverage for MoTDeReL and baseline approaches across benchmark case studies

Case	Metric	FP-Growth	HGAPSO (Kalaee and Rafe 2019)	GA (Bahrampour and Rafe 2020)	WholeAcoT (Ghasemi et al. 2025a)	ISR-MCF	MoT-DeReL
DPS	Mean	12.2	2	17.1	17.5	13.6	**18.2**
	Median	12	2	17	17.5	13.5	18.5
	Variance	2.4	0	0.6	0.5	1.1	2.3
HMS	Mean	32.9	8	33.9	34.0	27.1	**54.3**
	Median	33	8	34	34	28	53.5
	Variance	1.3	0	0.7	0.5	4.7	2.9
BTS	Mean	79.1	49.2	74.5	85.9	84.0	**95.5**
	Median	78.5	49	80	85	83.5	78.5
	Variance	5.5	5.4	22.5	4.3	12.3	37.8
OSS	Mean	45.9	46.0	122.9	207.8	166.3	**223.2**
	Median	46	46	120.5	207	170.5	215.5
	Variance	0.9	1.3	38.3	1.3	24.3	34.3
TAS	Mean	84	55.7	203.0	286.70	230.00	**370.7**
	Median	86	57	184	288	228	360
	Variance	4.7	4.4	75.9	5	23.2	31
SCRP	Mean	351.1	108.0	390.8	457.2	454.5	**981.6**
	Median	345	108.5	376.5	455	442	1014
	Variance	27.7	2.2	52.9	10.1	39.3	343

To answer this question, we analyzed the coverage scores obtained by each approach across all benchmark case studies. As shown in Table 4, coverage results are reported using three statistical measures, mean, median, and variance, computed over ten independent runs. The first column lists the case studies, and the second specifies the type of metric. The subsequent columns present the results for FP-Growth, HGAPSO, GA, WholeAcoT, ISR-MCF, and the proposed MoTDeReL method.

For a clearer comparison of average coverage, Fig. 8 visualizes the mean values from the table. This graphical representation highlights the relative performance of each approach in terms of their average test objective coverage. The bold values in Table 4 indicate the highest mean coverage scores achieved for each case study.

In the DPs case study, MoTDeReL achieves the highest mean coverage (18.2), slightly outperforming WholeAcoT (17.5) and GA (17.1). ISR-MCF (13.6) and FP-Growth (12.2) provide moderate results, while HGAPSO performs the worst with only 2.0 coverage. In HMS case study, MoTDeReL leads significantly with 54.3, a substantial improvement over all other approaches. The next best, WholeAcoT (34.0), is followed closely by GA (33.9) and FP-Growth (32.9). HGAPSO again performs the worst (8.0), while ISR-MCF shows limited effectiveness (27.1). For BTS case study, MoTDeReL again leads with 95.5, followed by WholeAcoT (85.9) and ISR-MCF (84.0). Interestingly, FP-Growth (79.1) outperforms GA (74.5), reversing the trend seen in previous models. HGAPSO ranks last (49.2), continuing its underperformance. In OSS case study, MoTDeReL achieves the highest mean coverage (223.2), with WholeAcoT and ISR-MCF following at 207.8 and 166.3, respectively. GA (122.9) performs moderately, while FP-Growth (45.9) slightly underperforms

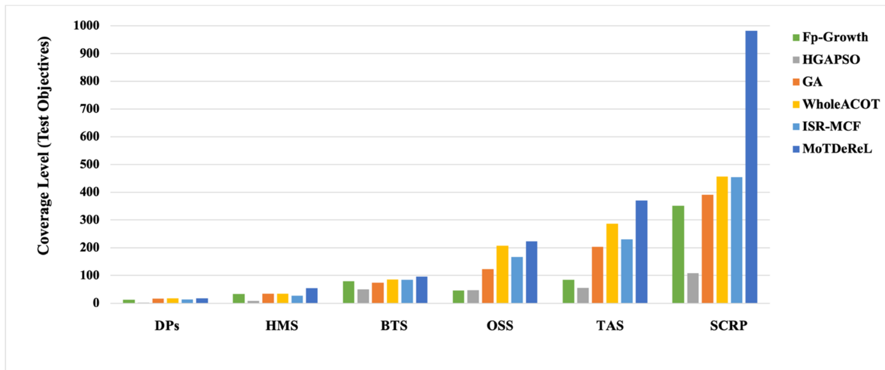


Fig. 8 Mean coverage comparison across all case studies and approaches

HGAPSO (46.0), marking the only case where FP-Growth ranks lowest. In TAS case study, MoTDeReL maintains its lead (370.7), with WholeAcoT (286.7) and ISR-MCF (230.0) trailing. GA (203.0), FP-Growth (84.0), and HGAPSO (55.7) remain consistently behind. In SCRP case study, MoTDeReL achieves a mean coverage of 981.6, more than double that of the next best method, WholeAcoT (457.2). All other approaches perform significantly worse, with HGAPSO (108.0) at the bottom, and FP-Growth (351.1) and GA (390.8) showing only modest results.

A comparison of these values shows that MoTDeReL consistently achieves the best coverage of the test objectives across all benchmarks, outperforming all counterparts. As the size of the case study increases, the difference in coverage scores achieved by the MoTDeReL approach becomes more significant.

To complement these summary statistics, it is important to examine the distribution of coverage values across the different algorithms for each case study. Figure 9 provides violin plots illustrating the spread, median, and density of coverage results across ten independent runs per algorithm. Each subfigure in Fig. 9 represents a case study, highlighting the variance and median coverage for each approach.

Figure 9 highlights important distributional patterns. While MoTDeReL consistently achieves the highest median coverage across all case studies, its distribution is wider for complex systems (e.g., TAS and SCRP), reflecting its exploratory nature. This variance, however, still lies well above the maximum coverage of competing methods, demonstrating robustness even in stochastic conditions. In contrast, FP-Growth and HGAPSO exhibit low variance but at the cost of limited coverage, indicating weaker adaptability. GA and WholeAcoT provide competitive results for smaller cases but fail to scale effectively, as evident from their plateaued distributions in larger models. A more compact distribution reflects stability but often at the cost of adaptability, as seen in FP-Growth and HGAPSO. In contrast, MoTDeReL's wider distribution in complex cases suggests active exploration, leading to significantly higher best and median coverage values.

While these results offer valuable insights, statistical significance must also be verified. To establish the significance and practical relevance of the findings, we conducted a detailed analysis using $P - value$ calculations and the $\hat{A}_{12}$ effect size measure.

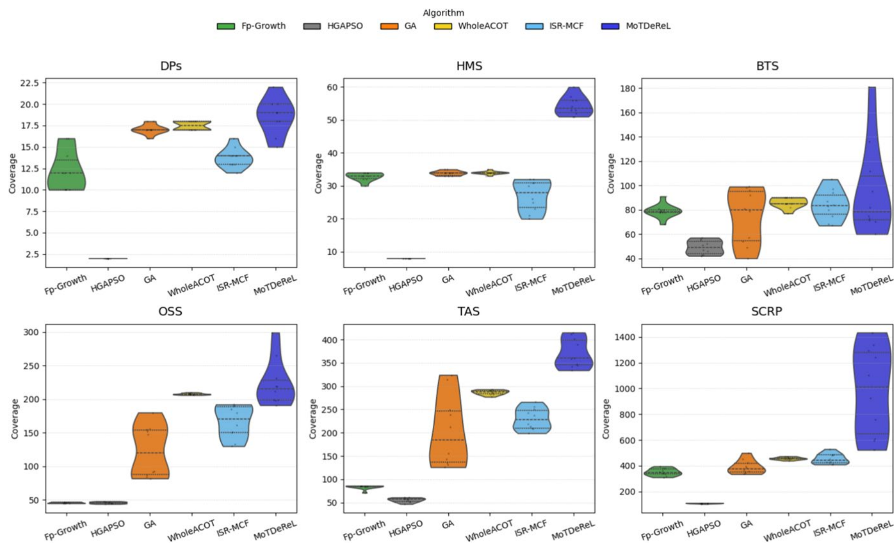


Fig. 9 Coverage distribution across MoTDeReL and baseline approaches for benchmark case studies using violin plots

Table 5 presents the results of pairwise statistical comparisons between MoTDeReL and the other approaches across the six benchmark case studies. For each comparison, the $P - value$ indicates whether the observed difference in mean coverage is statistically significant, while the $\hat{A}12$ effect size represents the probability that MoTDeReL achieves higher coverage than the corresponding method. A value of $\hat{A}12 > 0.5$ suggests superior performance by MoTDeReL, whereas a value below 0.5 indicates lower performance. A value equal to 0.5 denotes no measurable difference. Statistically significant outcomes (P-value < 0.05) are highlighted in bold to emphasize meaningful differences.

Based on the findings presented in Table 5, MoTDeReL demonstrates consistent and significant improvement in coverage across most case studies. In the majority of comparisons, the $\hat{A}_{12}$ values exceed 0.9, often reaching 1, indicating a very high probability that MoTDeReL outperforms the other methods in terms of average coverage.

In the DPs case study, MoTDeReL demonstrates strong effect sizes compared to FP-Growth, HGAPSO, and ISR-MCF (all with $\hat{A}_{12} \geq 0.95$), while showing no statistically significant difference when compared to GA and WholeAcoT, as their respective $P - values$ exceed the 0.05 threshold. Similarly, in BTS, the comparisons with FP-Growth, GA, WholeAcoT, and ISR-MCF also result in non-significant differences, whereas the comparison with HGAPSO reveals a clear and statistically significant advantage in favor of MoTDeReL. In the OSS case study, MoTDeReL achieves the highest average coverage; however, its comparison with WholeAcoT does not yield statistical significance ($P - values > 0.05$).

Across the HMS, TAS, and SCRF case studies, MoTDeReL consistently outperforms all other methods, with statistically significant P-values and large effect sizes ($\hat{A}12 \geq 0.98$), demonstrating a clear performance advantage.

Table 5 Effect size of coverage differences between MoTDeReL and other approaches across benchmark case studies

Case	Metric	FP-Growth	HGAPSO (Kalaee and Rafe 2019)	GA (Bahrampour and Rafe 2020)	WholeAcoT (Ghasemi et al. 2025a)	ISR-MCF
DPS	P-Value	6.94×10^{-4}	6.20×10^{-5}	0.1621	0.2584	3.35×10^{-4}
	$\hat{A}_{12}$	**0.95**	**1**	0.69	0.65	**0.97**
HMS	P-Value	1.65×10^{-4}	6.25×10^{-5}	1.56×10^{-4}	1.09×10^{-4}	1.78×10^{-4}
	$\hat{A}_{12}$	**1**	**1**	**1**	**1**	**1**
BTS	P-Value	0.8495	1.82×10^{-4}	0.3447	0.5421	1
	$\hat{A}_{12}$	0.53	**1**	0.63	0.41	0.51
OSS	P-Value	1.63×10^{-4}	1.61×10^{-4}	1.81×10^{-4}	0.4688	2.81×10^{-4}
	$\hat{A}_{12}$	**1**	**1**	**1**	0.60	**0.98**
TAS	P-Value	1.32×10^{-1}	1.81×10^{-4}	1.83×10^{-4}	1.79×10^{-4}	1.83×10^{-4}
	$\hat{A}_{12}$	**1**	**1**	**1**	**1**	**1**
SCRP	P-Value	1.81×10^{-4}	1.54×10^{-4}	1.83×10^{-4}	1.82×10^{-4}	2.46×10^{-4}
	$\hat{A}_{12}$	**1**	**1**	**1**	**1**	**0.99**

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

In total, 30 pairwise comparisons were conducted between MoTDeReL and competing approaches across all case studies. Among these, 23 comparisons showed statistically significant differences (P-value < 0.05), confirming MoTDeReL's consistent advantage in coverage performance. The remaining 7 comparisons did not reach statistical significance (the $P - values$ exceeded the 0.05 threshold), likely due to marginal performance differences or limited sample variability.

To visually illustrate the effect sizes for average coverage across the case studies, Fig. 10 presents box plots for the 23 comparisons in which MoTDeReL demonstrated statistically significant differences. These visual summaries provide a clearer view of its comparative performance.

In conclusion, MoTDeReL outperforms existing methods across all evaluated scenarios, with no cases of inferior performance. In instances where statistical significance was not established (7 comparisons where P-values exceed 0.05), further analysis using complementary metrics, such as test suite size or generation time, may offer a more nuanced comparison and reinforce the practical strengths of the proposed approach.

Q2: How much has MoTDeReL reduced testing costs compared to the state-of-the-art?

In software testing, achieving cost-effective tests is crucial for an efficient testing process and for delivering high-quality software. When a smaller test case reveals more defects, its cost becomes significantly lower compared to larger tests with the same output. In software testing, it is ideal for a test to achieve higher objective coverage while remaining compact in size.

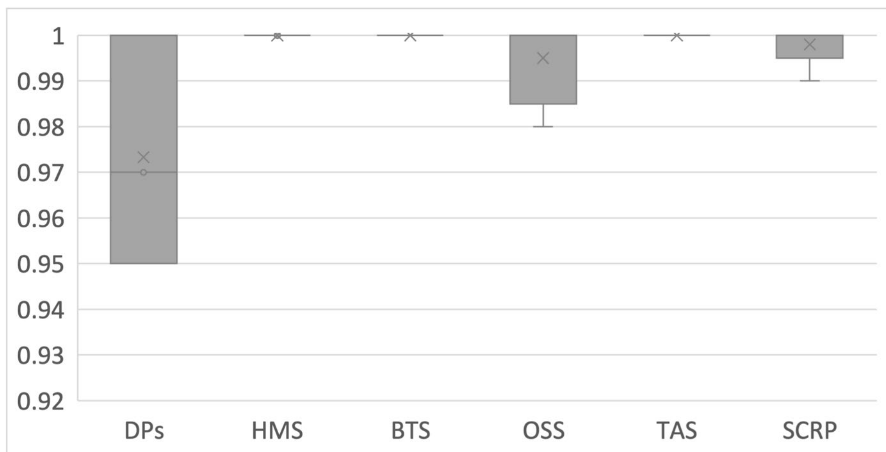


Fig. 10 Box plot visualization of coverage variation across case studies: MoTDeReL consistently outperforms competing approaches in achieved coverage

One commonly used metric for evaluating approaches is the length of the generated test cases. Table 6 indicates the size of the generated test cases for each approach, with the smallest test size in each case study shown in bold. For test size, we calculated the total size of the final generated test suite, which includes multiple test cases. Based on the bold values in Table 6, 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 (Schieferdecker 2012) that describes the capacity of the specified test to fulfill the test objectives along with its size, which affects the testing process. As shown in Definition 6, the testing cost captures the tradeoff between the number of tests and their achieved coverage.

Definition 6 Given a test suite *TS* with length *L* and coverage value *CV*, the testing cost is defined as

$$\mathrm{Cost}(TS) = \frac{L}{CV}.$$

Figure 11 represents the cost of generated tests associated with FP-Growth, HGAPSO, GA, WholeAcoT, ISR-MCF, and MoTDeReL. The goal is to have smaller tests 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.

Table 6 Comparison of the generated test suite size for MoTDeReL and baseline approaches across benchmark case studies

Case	FP-Growth	HGAPSO (Kalaee and Rafe 2019)	GA (Bahrampour and Rafe 2020)	WholeAcoT (Ghasemi et al. 2025a)	ISR-MCF	MoT-DeReL
DPS	28	19.3	29.3	**17.3**	20	19.9
HMS	30	18.7	32.8	**18**	20	19.5
BTS	151.33	105	79	**29.2**	105	88.4
OSS	**60**	103.6	74.3	113.6	120	119
TAS	113.77	103.2	**57.5**	99.9	105	83
SCRP	200	292.3	**193**	207.50	300	256.5

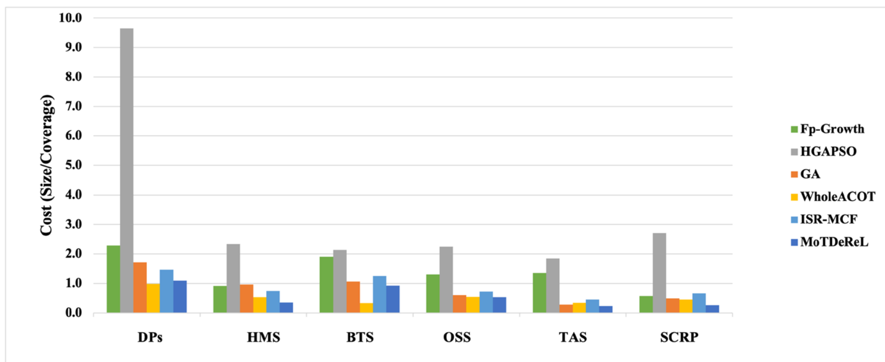


Fig. 11 Comparison of testing cost for MoTDeReL and baseline approaches across benchmark case studies

Q3: How does MoTDeReL work 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.

Figure 12 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. While 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.

5.4 Discussion

In this section, we discuss the experimental results to validate the objectives of our test suite generation approach and address some potential weaknesses.

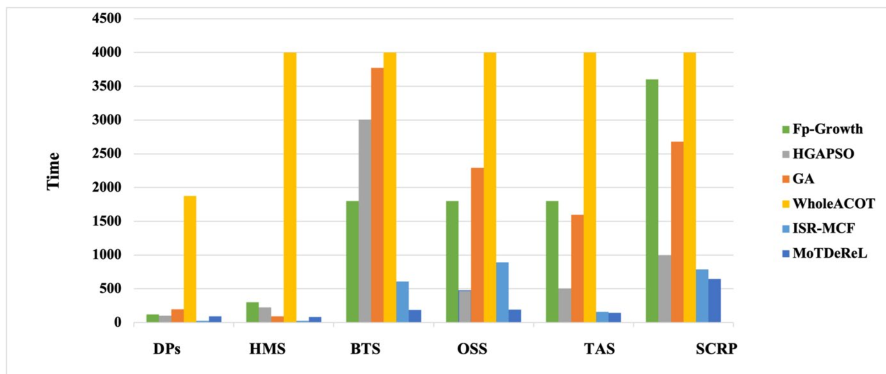


Fig. 12 Comparison of test development time for MoTDeReL and baseline approaches across benchmark case studies

Tables 4 and 6 demonstrate 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 prioritizing actions that yield 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, prioritizing 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 optimize decision-making over time.

Table 6 indicates that the tests generated by MoTDeReL are not smaller than with the other approaches. While 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. 11, the testing cost of MoTDeReL is lower than HGAPSO, GA, and WholeAcoT in most cases. This suggests that MoTDeReL generates an efficient test suite, improving test objective coverage while maintaining a shorter length.

The data in Fig. 12 shows that test generation in MoTDeReL is faster than other approaches. Since HGAPSO, GA, and WholeAcoT utilize multiple generations to produce semi-random TCs and TSs, they require more time to produce final generation. In contrast, MoTDeReL uses a previously trained neural network to select actions in new iterations, leading to improved TCs in terms of coverage score.

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

- It generates acceptable TSs with high coverage compared to related approaches.
- It produces effective TSs with low testing cost relative to other methods.
- The runtime of MoTDeReL is significantly lower than other approaches.
- MoTDeReL improves with each iteration as it learns, progressively enhancing 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.

An additional factor that influences the effectiveness of MoTDeReL is the choice of reinforcement learning hyperparameters, such as the exploration rate (ϵ), discount factor (γ), replay memory size, and network architecture. In this study, a fixed configuration was used across all case studies to ensure fairness, reproducibility, and comparability with existing approaches. While this configuration proved robust across systems of varying size and complexity, different SUT characteristics, such as the number of rules, branching factor, or state-space density, may benefit from alternative parameter settings. A systematic sensitivity analysis or adaptive hyperparameter tuning could further improve learning efficiency and coverage, particularly for large-scale or highly dynamic models.

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 high number of test objectives. It utilizes 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 based on the neural network's recommendations.

We implemented the approach alongside other related methods in the Groove toolset and evaluated them based on coverage score, testing cost, and test generation time. 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 prioritized experience replay.
- While Double DQN demonstrates strong overall performance in DRL, alternative methods such as the Actor-Critic method could also be explored.

- A deeper investigation into the impact of hyperparameter selection on learning outcomes is a promising direction for future work. In particular, adapting hyperparameters to the characteristics of the SUT, such as model size, rule complexity, or state-space structure, could further improve learning efficiency and coverage.
- Since the reward function plays a central role in guiding the learning process, future work will focus on refining and extending reward design. In addition to structural coverage metrics, we will investigate the use of mutation score as an effectiveness measure to assess the fault-detection capability of generated test cases, and explore the integration of mutation analysis into the reward function of MoTDeReL to guide learning toward generating more fault-revealing test cases.

Author Contributions S.Gh implemented the idea, wrote some parts of the paper. M.A.A implemented some parts to compare the results, she wrote some other parts of the paper. V. R provided the main idea of using ML techniques to solve this problem and revised the paper. R.H advised on using GTS, how to generate tests and revised the paper.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing Interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

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

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

Asgari Araghi, M., Rafe, V., Khendek, F.: Using data mining techniques to generate test cases from graph transformation systems specifications. Autom. Softw. Eng. **31**(1), 17 (2024)

AsgariAraghi, M., Rafe, V., Kalaee, A.: Model-based test case generation from graph transformation specifications using beam search algorithm. Tabriz J Electr Eng **49**(1), 343–356 (2019). https://tjee.tabrizu.ac.ir/article_8786.html

Bahrampour, A., Rafe, V.: Using search-based techniques for testing executable software models specified through graph transformations. Int. J. Mach. Learn. Cybern. **11**, 2743–2770 (2020)

Bahrampour, A., Rafe, V.: Using memetic algorithm for robustness testing of contract-based software models. Artif. Intell. Rev. **54**, 877–915 (2021)

Baier, C., Katoen, J.P.: Principles of model checking (2008)

Biere, A.: Bounded model checking. In: Handbook of satisfiability, pp. 739–764. IOS press (2021)

Bruijn, V.: Model-based testing with graph grammers. Ph.D. thesis, University of Twente (2013)

Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007. pp. 31–36. WEASELTech '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1353673.1353681

Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Comput. Intell. Mag. **1**, 28–39 (2006)

Duboue, P.: The art of feature engineering: essentials for machine learning (2020)

Eberhart, R., Kennedy, J.: Particle swarm optimization. In: Proceedings of the IEEE international conference on neural networks pp. 1942–1948 (1995)

Ehrig, H., Rozenberg, G., Kreowski, R.H.J.: Handbook of graph grammars and computing by graph transformation, **3** (1999)

Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G.: Graph Transformations: Second International Conference, vol. 3256. Springer, Rome, Italy (2004)

Engels, G., Güldali, B., Lohmann, M.: Towards model-driven unit testing. Models Softw Eng Workshops Symposia Models **9**, 182–192 (2006)

Enoiu, E., Causevic, A., Ostrand, T., Weyuker, E., Sundmark, D., Pettersson, P.: Automated test generation using model-checking: An industrial evaluation. Int J Softw Tools Technol Trans (2014). https://doi.org/10.1007/s10009-014-0355-9

Fraser, G., Wotawa, F., Ammann, P.: Issues in using model checkers for test case generation. J. Syst. Softw. **82**, 1403–1418 (2009). https://doi.org/10.1016/j.jss.2009.05.016

Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw Testing, Verif Reliab **19**, 215–261 (2009)

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

Ghasemi, S., Rafe, V., Bahrampour, A., Heckel, R.: Whole test suite generation from graph transformation specifications using ant colony optimization. Software Qual. J. **33**, 1–37 (2025)

Ghasemi, S., Rafe, V., Mehrabi, M., Heckel, R., Al-Azzoni, I.: Test case generation from graph transformation systems using deep reinforcement learning. In: Endrullis, J., Tichy, M. (eds.) Graph Transformation, pp. 178–201. Springer Nature Switzerland, Cham (2025)

Gómez-Abajo, P., Guerra, E., de Lara, J.: Wodel-test: A model-based framework for engineering language-specific mutation testing tools. SoftwareX **31**, 102195 (2025). https://doi.org/10.1016/j.softx.2025.102195, https://www.sciencedirect.com/science/article/pii/S2352711025001621

Gönczy, L., Heckel, R., Varró, D.: Model-based testing of service infrastructure components. In: Testing of Software and Communicating Systems: 19th IFIP TC6/WG6. 1 International Conference pp. 155–170 (2007)

González, C., Varmazyar, M., Nejati, S., Briand, L., Isasi, Y.: Enabling model testing of cyber-physical systems. In: Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems pp. 176–186 (2018)

Hasselt, H.V., Guez, A., Silver, D.: Deep reinforcement learning with double qlearning. In: Proceedings of the AAAI conference on artificial intelligence (2016)

Haupt, R., Haupt, S.: Practical genetic algorithms (2004)

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

Heckel, R., Khan, T., Machado, R.: Towards test coverage criteria for visual contracts. Electron. Commun. EASST 41 (2011)

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

Jadhav, P., Lanke, G., Shrivastava, A., Patil, V., Gupta, S.: Efficient test case generation using model-based testing, and model paradigm approach. In: International Conference on Recent Trends in Computing pp. 817–828 (2023)

Kalaee, 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)

Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In: Valmari, A. (ed.) Model Checking Software, pp. 299–305. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)

Kumar, D., Mishra, K.: The impacts of test automation on software's cost, quality and time to market. Proc. Comput. Sci. **79**, 8–15 (2016). https://doi.org/10.1016/j.procs.2016.03.003, https://www.sciencedirect.com/science/article/pii/S1877050916001277, Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016

Lara, J.d., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.D., Weber, H. (eds.) Fundamental Approaches to Software Engineering. pp. 174–188. Springer, Berlin Heidelberg, Berlin, Heidelberg (2002)
Leloudas, P.: Introduction to software testing: A practical guide to testing, design, automation, and execution. Apress, Berkeley, CA, 1 edn. (2023). https://doi.org/10.1007/978-1-4842-9514-4
Li, W., Le Gall, F., Spaseski, N.: A survey on model-based testing tools for test case generation. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) Tools and Methods of Program Analysis, pp. 77–89. Springer International Publishing, Cham (2018)
Mehrabi, M., Rafe, V.: Using deep reinforcement learning to search reachability properties in systems specified through graph transformation. Soft. Comput. **26**, 9635–9663 (2022)
Mohalik, S., Gadkari, A., Yeolekar, A., Shashidhar, K., Ramesh, S.: Automatic test case generation from simulink/stateflow models using model checking. Softw. Test. Verif. Reliab. **24**, 155–180 (2014)
Mustafa, A., Wan-Kadir, W.M., Ibrahim, N., Shah, M.A., Younas, M., Khan, A., Zareei, M., Alanazi, F.: Automated test case generation from requirements: A systematic literature review. Comput. Mater. Continua **67**(2), 1819–1833 (2021)
Nabuco, M., Paiva, A.: Model-based test case generation for web applications. In: International Conference on Computational Science and Its Applications pp. 248–262 (2014)
Offutt, J., Ammann, P.: Introduction to software testing. Cambridge University Press Cambridge (2008)
Rafe, V., Mohammady, S., Cuevas, E.: Using bayesian optimization algorithm for model-based integration testing. Soft. Comput. **26**(7), 3503–3525 (2022). https://doi.org/10.1007/s00500-021-06476-9
Rapps, S., Weyuker, E.: Selecting software test data using data flow information. Softw. Eng. IEEE Trans. **SE–11**, 367–375 (1985). https://doi.org/10.1109/TSE.1985.232226
Rayadurgam, S., Heimdahl, M.: Coverage based test-case generation using model checkers. In: Proceedings eighth annual IEEE international conference and workshop on the engineering of computer-based systems-ECBS 2001. pp. 83–91 (2001). https://doi.org/10.1109/ECBS.2001.922409
Rensink, A.: The groove simulator: A tool for state space generation. Appl Graph Trans Ind Relev Second Intl Workshop **2**, 479–485 (2003)
Rensink, A., Boneva, I., Kastenberg, H., Staijen, T.: User manual for the groove tool set. University of Twente, The Netherlands, Department of Computer Science (2010)
Rocha, M., Simão, A., Sousa, T.: Model-based test case generation from uml sequence diagrams using extended finite state machines. Software Qual. J. **29**(3), 597–627 (2021). https://doi.org/10.1007/s11219-020-09531-0
Runge, O., Khan, T., Heckel, R.: Test case generation using visual contracts. Electron. Commun. EASST **58** (2013)
Sahoo, R.K., Ojha, D., Mohapatra, D.P., Patra, M.R.: Automated test case generation and optimization: a comparative review. Int. J. Comput. Sci. Inf. Technol. **8**(5), 19–32 (2016)
Schieferdecker, I.: Model-based testing. IEEE Softw. **29**(01), 14–18 (2012)
Schmidt, A., Varró, D.: Checkvml: A tool for model checking visual modeling languages. International Conference on the Unified Modeling Language pp. 92–95 (2003)
Sulaiman, R., Jawawi, D., Halim, S.: Cost-effective test case generation with the hyper-heuristic for software product line testing. Adv. Eng. Softw. **175**, 103335 (2023)
Sutton, R.: Reinforcement learning: An introduction (2018)
Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance, pp. 446–453. Springer, Berlin Heidelberg, Berlin, Heidelberg (2004)
Thöne, S.: Dynamic software architectures: A style-based modeling and refinement technique with graph transformations. Ph.D. thesis, University of Paderborn (2005)
Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A.: Recent advances in model-based testing. Adv. Comput. **101**, 53–120 (2016)
Varro, D., Balogh, A.: The model transformation language of the viatra2 framework. Sci. Comput. Program. **68**, 214–234 (2007). https://doi.org/10.1016/j.scico.2007.05.004
Villani, E., Pastl, R., Coracini, G., Ambrosio, A.: Integrating model checking and model based testing for industrial software development. Comput. Ind. **104**, 88–102 (2019). https://doi.org/10.1016/j.compind.2018.08.003
Yang, Y.: Improve model testing by integrating bounded model checking and coverage guided fuzzing. Electronics (Basel) **12**, 1573 (2023)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Simin Ghasemi[1] · Maryam Asgari Araghi[2] · Vahid Rafe[3] · Reiko Heckel[4]

✉ Vahid Rafe
vahid.rafe@city.ac.uk

Simin Ghasemi
simin-ghasemi@pnu.ac.ir

Maryam Asgari Araghi
ma_asgar@encs.concordia.ca

Reiko Heckel
rh122@le.ac.uk

1 Department of Computer Engineering, Payame Noor University (PNU), Tehran, Iran

2 Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada

3 Department of Computer Engineering, City St Georg's, University of London, London, UK

4 School of Computing and Mathematical Sciences, University of Leicester, Leicester, UK