# Risk-Based Interoperability Testing Using Reinforcement Learning

André Reichstaller, Benedikt Eberhardinger, Alexander Knapp, Wolfgang Reif, Marcel Gehlen

# Risk-based Interoperability Testing using Reinforcement Learning

André Reichstaller[1], Benedikt Eberhardinger[1],
Alexander Knapp[1], Wolfgang Reif[1], and Marcel Gehlen[2]

[1] Institute for Software & Systems Engineering
University of Augsburg, Germany
{lastname}@isse.de

[2] MaibornWolff GmbH, Munich, Germany
marcel.gehlen@maibornwolff.de

**Abstract.** Risk-based test strategies enable the tester to harmonize the number of specified test cases with imposed time and cost constraints. However, the risk assessment itself often requires a considerable effort of cost and time, since it is rarely automated. Especially for complex tasks such as testing the interoperability of different components it is expensive to manually assess the criticality of possible faults. We present a method that operationalizes the risk assessment for interoperability testing. This method uses behavior models of the system under test and reinforcement learning techniques to break down the criticality of given failure situations to the relevance of single system actions for being tested. Based on this risk assessment, a desired number of test cases is generated which covers as much relevance as possible. Risk models and test cases have been generated for a mobile payment system within an industrial case study.

## 1 Introduction

*Interoperability testing* of a distributed system checks whether the components of the system are able to communicate with each other and thus render requested services correctly through interaction [5]. The typically high number of possible interaction scenarios (e.g., combination of messages) makes interoperability testing a complex task. Since it seems impossible to cover all scenarios, their relevance for being tested has to be prioritized somehow. A *criticality-based test strategy* should focus the test effort on revealing faults which are expected to lead to the most critical failures [2]. However, the existence of implementation faults and the ensuing reachability of failures in real operation is unknown. Still, extending behavior models of the communicating components of the *System under Test* (*SuT*) with possible implementation faults, we can at least estimate the impact of a fault on the reachability of failures. The challenge is to find causal connections between the implementation faults and resulting failures.

We tackle this challenge by combining *model-based* [16] and *risk-based* testing methods [1, 2] with *reinforcement learning* [15]. Our approach builds on given behavior models of the interacting components of the SuT, common implementation faults, and a set of the most critical failure situations, each of them described by the combination of

component states and a score of its deemed effect. A failure situation is reached if all of the combined component states are active at the same point in time. None of the failures is actually reachable in the given behavior models, as the models do not describe faults. If we introduce common implementation faults into the behavior models, however, we are able to assess the *criticality* of faults, i.e., their "ability" to cause the effect of failures.

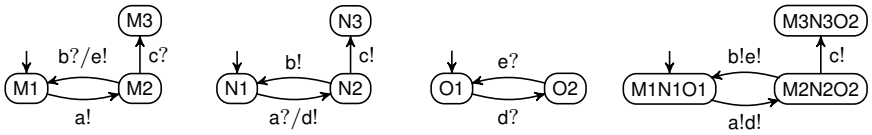
Let us imagine that a malicious developer of a component actually tries to induce the whole system to reach the maximum effect of failures in real operation by implementing the "right" component faults. These faults would be the ones to be tested for with highest priority. What he could do is to use a learning technique, such as reinforcement learning [15], on a simulated environment: He could implement his component as an intelligent agent which makes its own local decisions to achieve the global goal of reaching the most critical failures. This agent then would map received rewards to the preceding actions (either specified in the behavior model or faults) so as to assess the expected return for every possible action. The ultimate reward to be reinforced would be reaching a critical failure situation. Then the agent's learned expected return for executing a fault can be understood as the fault's criticality.

For finding those test cases which cover the most critical faults with highest priority, it seems reasonable to apply the same technique as our imaginary malicious developer. This procedure can be seen as defending the system against the faults he could inject. After the learning phase, each agent contains a function that maps its actions to their expected return, i.e., their criticality. From a *mutation testing* perspective [12], the actions representing anticipated faults can be seen as mutants of the specified actions in the behavior models. The functions of the agents weight these mutants by their criticality. Thus, test cases can now be prioritized by the criticality of the mutants they are assumed to kill. We propose a method for reducing the criticality of the mutants to the relevance of specified actions in the given behavior models for being tested. Building on this, we generate a desired number of logical test cases covering the most relevant actions. We have implemented our approach of deriving risk-optimized test cases using reinforcement learning and we have applied the method to a case study provided by an industrial partner. The case study showed, in particular, that the assessment of the criticality of actions by experts is matched by the values learned by reinforcement learning.

The remainder of this paper is structured as follows: In Sect. 2, we outline the used behavior models. Based on these models, we show in Sect. 3 how the criticality of faults can be estimated using reinforcement learning. In Sect. 4, we present our method for deriving a desired number of test cases that cover as much criticality as possible. Section 5 shows how the overall approach scales in an industrial case study in which we applied the concepts to a mobile payment system. After placing our approach in context with related work in Sect. 6, we give an outlook to future investigations in Sect. 7.

## 2 Test Model Specifications

We build our approach on three inputs: (1) models for the desired behavior of all local SuT components together with a precise description of their communication paradigm; (2) a fault model for implementation and communication defects spanning the fault variability space; and (3) an (expert's) estimate of critical global system situations.




(a) Component M (b) Component N (c) Component O (d) System behavior model of M, N, and O

**Fig. 1.** Model of the components M, N and O and their broadcasting composition.

### 2.1 Behavior SuT Models and Their Communication

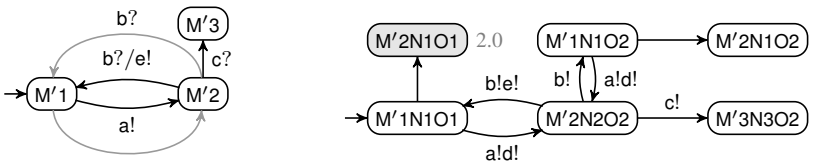
Though our testing approach is not limited to a particular formalism of behavior models, for conciseness, we use in the following a rather simple model of non-deterministic finite state machines communicating over a broadcasting message bus. The execution of an assembly of such components happens in rounds in which each component chooses some currently possible action: An action is possible if all of its message dependencies have been satisfied; a component stutters if, and only if, no such action is available.

For a concrete example, consider the three components M, N and O in Figs. 1a to 1c. The transition $\mathsf{M1} \xrightarrow{\mathsf{a!}} \mathsf{M2}$ of M defines an action that broadcasts the message a; $\mathsf{N1} \xrightarrow{\mathsf{a?/d!}} \mathsf{N2}$ of N defines an action that broadcasts the message d when message a is received; and $\mathsf{O1} \xrightarrow{\mathsf{d?}} \mathsf{O2}$ of O defines an action that does not broadcast any message when d is received. In addition to possible broadcasts, every action defines the executing component's change in state. Figure 1d shows the composition of M, N, and O according to broadcasting communication: A local action that broadcasts a message by M leads to a local action receiving the message by N and vice versa. Furthermore, the actions $\mathsf{N1} \xrightarrow{\mathsf{a?/d!}} \mathsf{N2}$ and $\mathsf{M2} \xrightarrow{\mathsf{b?/e!}} \mathsf{M1}$ trigger the actions $\mathsf{O1} \xrightarrow{\mathsf{d?}} \mathsf{O2}$ and $\mathsf{O2} \xrightarrow{\mathsf{e?}} \mathsf{O1}$ in O, respectively. In particular, action $\mathsf{M1} \xrightarrow{\mathsf{a!}} \mathsf{M2}$ causes the global effect that the entire system will change its state to M2N2O2. By contrast, if in the next round N chooses action $\mathsf{N2} \xrightarrow{\mathsf{c!}} \mathsf{N3}$, M has to choose $\mathsf{M2} \xrightarrow{\mathsf{c?}} \mathsf{M3}$ and O has to stutter.

The composition of those *component behavior models* forms the *system behavior model*. This model shows the resulting composed actions as well as the reachable composed states forming again a (component) behavior model.

### 2.2 Fault Models

The given behavior models for the SuT describe exclusively desired system behavior. The actual system behavior could, however, differ in an unknown way because of, e.g., implementation or communication faults. Since considering all imaginable faults would be quite demanding, we content ourselves with "common faults", i.e., classes of faults that are known to be frequently made. We assume the common faults to be specified in fault models defining their representation in the behavior models of the SuT. Based on them, we generate our proper *component test models* extending the given behavior models with faulty behavior using mutation.




(a) Mutated component M′ (for M)

(b) Composition of M′, N, O

**Fig. 2.** Component test model based on the fault model of message losses and resulting system test model composing M′, N and O. Depicted in gray in (b), the system test model has been complemented by a negative situation M′2−O1 with negativity score $\nu = 2.0$.

In communicating components such as M, N and O, an exemplary common fault is the loss of messages. The associated fault model could define its representation as mutations of broadcasting transitions leaving out some messages to send. Figure 2a shows the generated test model M′ of component M for this fault model; M′1, M′2, and M′3 are just different names for M1, M2, and M3, only transitions have been added. The composition of the component test model M′, i.e., the mutated component M, and the original component behavior models N and O, shown in Fig. 2b, renders several new composed states reachable, in particular M′2N1O1, M′2N1O2, and M′N1O2. A composition involving at least one component test model is called *system test model*, as opposed to the fault-free system behavior model.

### 2.3 Negative Situations

By their non-deterministic nature, our test models comprise different behavior variations of the SuT, and we do not know which of them is actually implemented. Our aim is to identify those variations that would be associated with the most critical failures. However, the assessment of a failure's criticality is rather subjective and therefore hard to automate. Thus, we assume critical failures as well as a score for their criticality to be given as inputs. We represent such failures by *negative situations* described by a tuple of component states that associates each component with at most one state. Negative situations are annotated with a *negativity score* $\nu \in \mathbb{R}_{>0}$ quantifying their criticality.

In the example of components M, N, and O, a negative situation could be given by M2−O1, i.e., all those system states where component M is in state M2 and component O is in state O1; we choose $\nu = 2.0$. For the system test model, state M2 now corresponds to M′2 (see Fig. 2a) and thus all composed states of the form M′2−O1 are deemed critical. In our example (see Fig. 2b), just M′2N1O1 is reachable. Obviously, this negative situation is not reachable in the system behavior model (cf. Fig. 1). This observation matches with the fact that failures should only be reachable through faulty behavior.

## 3 From Failure Negativity To Fault Criticality

Let now a system test model, composed of component test models according to a fault model, and an additional set of negative situations be given. In this system test model,

an action looks the more critical the higher the probability that negative situations are reached through this action and the higher the negativity scores of the reachable negative situations. Consequently, to be able to focus the test effort on revealing the most critical faults, we first have to quantify these expectations for every action. In other words, we have to map the critical failures (represented by negative situations) to those faulty and specified actions of the component test models which lead to them, and give local negativity scores to such actions. This process shows similarities to introducing decision makers, such as malicious developers, that exercise control over our models. They aim to collect the maximally possible negativity score and thus are trying to find the most critical actions.

A more formal way for modeling this task of making sequential decisions is provided by the framework of *Markov Decision Processes* (MDPs) [6]. An MDP is described by a state space $S$ and an action space $A$; a map $T : S \times A \times S \to [0, 1]$ giving probabilities over state transitions, such that $T(s, a, s')$ indicates the probability that action $a$ in state $s$ leads o state $s'$; and a reward function $R : S \times A \times S \to \mathbb{R}$ denoting rewards for taking particular transitions.

In fact, we are able to express our setting as an MDP: $S$ and $A$ are directly constructed from the composed states and actions of the system test model. Since we do not assume that transition probabilities of the SuT are known by the tester, we suppose $T$ for every state $s$ and every action $a$ to be uniformly distributed over the target states $s'$ that are forming transitions $(s, a, s')$ of the system test model. $R$ reinforces transitions $(s, a, s')$ with the negativity score of $s'$ if $s'$ is a negative situation, and with $0$ otherwise. MDPs are meant to be partially controlled by a decision maker (often called *agent*) in the following way: in every (discrete) time step, the agent is supposed to select an action $a \in A$ that is enabled in the current state $s \in S$. This triggers a state transition according to $T$ and offers a numerical reward signal according to $R$.

### 3.1 Solving MDPs

The task typically associated with an MDP is to find a strategy, i.e., a rule for selecting an action in any given state, that maximizes the agent's expected return (in terms of collected reward). In [15], Sutton and Barto summarize the class of so-called *reinforcement learning* methods which are designed for solving this task.

However, even though a malicious developer had to solve a reinforcement learning task for reaching a critical failure situation, ours seems different. Instead of finding a path through the system test model (or the MDP) that is supposed to offer the maximum reward, we first of all aim to assess the criticality of every action in order to eventually form a risk-optimized test suite. In terms of an MDP, we are searching for the expected returns of all actions. Fortunately, most of the reinforcement learning methods also provide us with these values. They are based on estimating value functions, i.e., mappings of states (or state-action tuples) to the expected return when being in the given state (or selecting the given action in a given state) [15]. Thus, in using one of these algorithms, we are able to estimate the actions' criticality.

*Temporal difference learning*, as a subclass of reinforcement learning, offers the special charm of working on sample experience and thus not requiring a model. In using a temporal difference method, we thus do not have to explicitly build the system test

model which may be prohibitively large due to the number of components and possible faults. In order to exploit this advantage, we have chosen a fully decentralized approach: Within a simulation of system runs, we associate each single component of the SuT with an agent that learns the expected return of its actions. An agent's action corresponds to the simulated execution of a transition specified in the component test model. The action can be executed as soon as the specified inputs of the associated transition are present. If an action is executed, the specified messages to broadcast are sent to the other agents. In this way an agent interacts with its environment (i.e., the entirety of agents) under the rules of the system test model. The agents are synchronized by logical time steps at which each agent performs exactly one action per step. Dependencies during a time step are resolved by a scheduler which implements the chosen communication paradigm over a message bus. At the end of each time step, each agent is situated in a state defined by the associated component test model. A composed state is formed by collecting the states of every agent and each agent is reinforced according to $R$.

### 3.2 $Q$-learning

More specifically, we follow the $Q$-learning approach [18]. Each agent owns a so-called $Q$-function mapping environmental states together with actions to their expected return. We call the pair of an environmental state and an action a *decision*. After a reward $R_{t+1}$ has been received for action $a_t$ executed at time step $t$ out of the environmental (global) state $\overline{s}_t$, the expected return for the decision $(\overline{s}_t, a_t)$ is updated as follows:

$$Q_{t+1}(\overline{s}_t, a_t) = Q_t(\overline{s}_t, a_t) + \alpha\big(R_{t+1} + \gamma \max_a Q_t(\overline{s}_{t+1}, a) - Q_t(\overline{s}_t, a_t)\big)\,. \qquad \text{(Q)}$$

The parameters $\alpha \in \,]0, 1]$ and $\gamma \in [0, 1]$ denote the *learning rate* and the *discount factor*. Decisions which have not been mapped on an expected return yet get a default assignment of $0$. As one can see in Eq. (Q), the expected returns – that are representing our measure of criticality – are updated with respect to a policy in which the agent chooses anytime the action with the highest criticality (represented by the $\max$-term in (Q)). This *optimal policy* invokes the worst-case behavior of the component, that, as we suppose, is the most appropriate one in case of risk-based testing.

As known for *off-policy* learning [15], the evaluated policy (in our case the optimal one) is not affected by the way of generating behavior during the learning process (*behavior policy*). However, our simultaneous simulation of several agents weakens this independence, since the reachability of a negative situation may depend on decisions of multiple agents. In our running example, such a dependence can be seen at the decision of agent B for choosing action N2 $\xrightarrow{\text{b!}}$ N1 in composed state $\mathsf{s}_1$ = M′2N2O2. This decision could lead to different composed states depending on the selected action of agent A. Let us assume that agent A implements a behavior policy which selects each possible action with equal probability (*uniformly distributed policy*). Then, the decision for N2 $\xrightarrow{\text{b!}}$ N1 in $\mathsf{s}_1$ will lead in half of the executions to $\mathsf{s}_2$ = M′1N1O1 and in the other half to $\mathsf{s}_3$ = M′1N1O2. Thus, we expect $Q(\text{M′2N2O2}, \text{N2} \xrightarrow{\text{b!}} \text{N1})$ in the equilibrium for (Q) (where $Q_{t+1}(\overline{s}, a) = Q_t(\overline{s}, a)$ for all $\overline{s}$ and $a$) to be the average of the two different outcomes $\nu(\mathsf{s}_2) + \gamma \max_a Q(\mathsf{s}_2, a)$ and $\nu(\mathsf{s}_3) + \gamma \max_a Q(\mathsf{s}_3, a)$. Table 1 shows the

**Table 1.** $Q$-functions computed by agents for the system test model of Fig. 2b with negative situation $\mathsf{M'2}-\mathsf{O1}$, $\nu = 2.0$, and $\gamma = 0.5$.

| | Agent with action (left) and criticality (right column) | | | | | |
|---|---|---|---|---|---|---|
| State | A for $\mathsf{M'}$ (see Fig. 2b) | | B for N (see Fig. 1b) | | C for O (see Fig. 1c) | |
| $\mathsf{M'1N1O1}$ | $\mathsf{M'1} \longrightarrow \mathsf{M'2}$ | 2.0 | $\mathsf{N1} \xrightarrow{\mathsf{a?/d!}} \mathsf{N2}$ | 0.29 | $\mathsf{O1} \xrightarrow{\mathsf{d?}} \mathsf{O2}$ | 0.5 |
| | $\mathsf{M'1} \xrightarrow{\mathsf{a!}} \mathsf{M'2}$ | 0.5 | $\mathsf{N1} \longrightarrow \mathsf{N1}$ | 2.0 | $\mathsf{O1} \longrightarrow \mathsf{O1}$ | 2.0 |
| $\mathsf{M'2N2O2}$ | $\mathsf{M'2} \xrightarrow{\mathsf{b?/e!}} \mathsf{M'1}$ | 1.0 | $\mathsf{N2} \xrightarrow{\mathsf{b!}} \mathsf{N1}$ | 0.57 | $\mathsf{O2} \xrightarrow{\mathsf{e?}} \mathsf{O1}$ | 1.0 |
| | $\mathsf{M'2} \xrightarrow{\mathsf{b?}} \mathsf{M'1}$ | 0.25 | $\mathsf{N2} \xrightarrow{\mathsf{c!}} \mathsf{N3}$ | 0.0 | $\mathsf{O2} \longrightarrow \mathsf{O2}$ | 0.06 |
| | $\mathsf{M'2} \xrightarrow{\mathsf{c?}} \mathsf{M'3}$ | 0.0 | | | | |
| $\mathsf{M'1N1O2}$ | $\mathsf{M'1} \xrightarrow{\mathsf{a!}} \mathsf{M'2}$ | 0.5 | $\mathsf{N1} \xrightarrow{\mathsf{a?/d!}} \mathsf{N2}$ | 0.29 | $\mathsf{O2} \longrightarrow \mathsf{O2}$ | 0.25 |
| | $\mathsf{M'1} \longrightarrow \mathsf{M'2}$ | 0.0 | $\mathsf{N1} \longrightarrow \mathsf{N1}$ | 0.0 | | |

$Q$-functions of agents A, B and C for $\gamma = 0.5$ in their equilibria, assuming uniformly distributed behavior policies.

## 4 Deriving Tests with High Risk-based Impact

Up to this point, we have a set of agents, each one containing a $Q$-function mapping decisions to criticality values. This, as an intermediate result, would enable the imaginary malicious developer of Sect. 1 to implement the most critical faults in his component; and it enables us to assess observed decisions of any of the SuT's components in real operation. However, we still want to use this learned information for generating test cases covering the most critical faults. For this purpose, two things have to be considered: (1) Positive test cases, as we exclusively consider in this paper, do only include decisions with specified actions (*specified decisions*) but are able to detect implemented decisions with mutated actions (*mutated decisions*). More precisely, we assume the test of a specified decision to detect all of its mutants, i.e., decisions with the same state but with actions that are mutants of that contained in the specified decision. Hence, we have to distinguish between a decision's criticality and a specified decision's relevance for being tested that, in fact, should even comprise the criticality values of its mutants. (2) A $Q$-function, as we formed it, assesses decisions with local actions (*local decisions*). A system test case, however, should specify the execution of *global decisions* involving one local decision per component.

Thus, we assess the relevance in (1) local and (2) global relevance functions whereby the latter depends on the first. System test cases then are generated and assessed using the global relevance functions.

### 4.1 Relevance Functions

The local relevance function $r$ maps each specified local decision to its relevance for being tested. We define the relevance of a decision by the sum of the criticality values of its mutants. This is reasonable, since a specified decision is deemed to reveal all of

**Table 2.** Relevance functions on the basis of the $Q$-functions in Tab. 1.

| Local decision | | |
|---|---|---|
| State | Action | Relevance |
| M′1N1O1 | M′1 $\xrightarrow{\text{a!}}$ M′2 | 2.5 |
| M′2N2O2 | M′2 $\xrightarrow{\text{b?/e!}}$ M′1 | 1.25 |
| M′2N2O2 | M′2 $\xrightarrow{\text{c?}}$ M′3 | 0.0 |
| M′1N1O2 | M′1 $\xrightarrow{\text{a!}}$ M′2 | 0.5 |

(a) Local relevance function for agent A.

| Decision name | Action | Relevance |
|---|---|---|
| $d1$ | M′1N1O1 $\xrightarrow{\text{a!d!}}$ M′2N2O2 | 3.29 |
| $d2$ | M′2N2O2 $\xrightarrow{\text{b!e!}}$ M′1N1O1 | 2.82 |
| $d3$ | M′2N2O2 $\xrightarrow{\text{c!}}$ M′3N3O2 | 0.06 |
| $d4$ | M′1N1O2 $\xrightarrow{\text{a!d!}}$ M′2N2O2 | 1.04 |

(b) Global relevance function on the basis of the local relevance functions in Tab. 2a.

its mutants if they are implemented. From a mutation-based testing perspective, the relevance can be seen as the reward for killing a set of mutants. More formally, for a specified decision $d = (\overline{s}, a)$, let $M(d)$ be the set of mutated decisions whose actions are mutants of $a$ and whose composed state is $\overline{s}$. Then we define

$$r(d) = Q(d) + \textstyle\sum_{d' \in M(d)} Q(d') \; .$$

Continuing the above example, Tab. 2a shows the local relevance function for agent A. The local relevance functions for agents B and C are the $Q$-functions of these agents as shown in Tab. 1, as they involve no mutated decision.

The global relevance function $\overline{r}$ maps each *possible* global decision to its relevance for being tested. A global decision $\overline{d}$ consists of one specified (local) decision per agent. A global decision is possible iff the contained local decisions can be made at the same time. Thus, the local decisions contained in a possible global decision share the same composed state and actions which satisfy the chosen communication paradigm. Since only the specified local decisions are considered, but not their mutants that would result in much more possible global decisions, the computation of the set of global decisions turns out to be feasible, even for rather complex models. The execution of a global decision by a test case implies the execution of all included specified (local) decisions. We define a global decision $\overline{d}$ to be as relevant as the sum of its local decisions:

$$\overline{r}(\overline{d}) = \textstyle\sum_{d \in \overline{d}} r(d) \; .$$

Table 2b shows the global relevance function for our running example of agents A, B and C where we abbreviate a global decision by its resulting composed action in the system test model.

### 4.2 Deriving Logical Test Cases

Building on the global relevance function, we are now able to derive a *risk-optimized test suite*, i.e., a suite of a desired number of logical interoperability test cases that covers as much relevance as possible. A logical test case comprises a path through the system behavior model starting from the initial state. However, we still want to avoid computing

**Table 3.** Test case selection algorithm and test cases generated for the case study.

1: $ts \leftarrow \emptyset$
2: **while** $|ts| \leq \sigma \wedge \exists \pi \in paths \,.\, \sum_{\overline{d} \in \pi} \overline{r}(\overline{d}) > 0$ **do**
3: **choose** $\pi \in \arg\max\{\sum_{\overline{d} \in \pi} \overline{r}(\overline{d}) \mid \pi \in paths\}$
4: $ts \leftarrow ts \cup \{\pi\}$ ▷ (2)
5: **for all** $\overline{d} \in \pi$ **do** $\overline{r}(\overline{d}) \leftarrow 0$ ▷ (3)
6: **return** $ts$

(a) Iterative test case selection

| No. | Test case | Cov. relev. |
|---|---|---|
| 1 | $d1 \rightarrow d2$ | 6.11 |
| 2 | $d1 \rightarrow d3$ | 0.06 |

(b) Test cases generated for Tab. 2b with covered relevance.

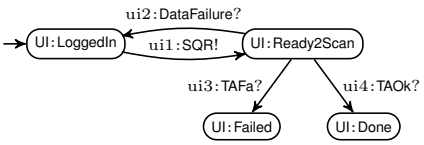
the complete system behavior model. In fact, since the criticality and relevance values were learned by sample experience, we have no guarantee that each composed action of the system behavior model has been reached and thus assessed by the global relevance function. Hence we directly consider the graph of the global relevance function, linking the states of the assessed global decisions with their defined composed actions. Decisions that cannot be reached from the composed source state, such as $d4$ in our example, are ignored. Then, a logical test case, i.e., a sequence of global decisions in the graph of the global relevance function, covers the relevance of each comprised decision.

In practice, the number of executable and assessable test cases is typically limited by an upper bound $\sigma$, particularly for huge systems with many components. Hence, we are looking for the $\sigma$ test cases covering the most relevant decisions with $\sigma$ given by the tester. More formally, if $\mathcal{P}_\sigma$ is the set of all cycle-free path sets with cardinality $\sigma$ and $\overline{r}(P) = \sum_{\overline{d} \in P} \overline{r}(\overline{d})$ for each $P \in \mathcal{P}_\sigma$, we aim to find $ts = \arg\max\{\overline{r}(P) \mid P \in \mathcal{P}_\sigma\}$, i.e., a test suite $ts$ with the maximum relevance. We solve this maximization problem by (1) identifying all cycle-free paths through the graph of the global relevance function, (2) iteratively adding test cases to the test suite, and (3) updating the values of the global relevance function after adding a test case. Table 3a implements (2) and (3) for the set $paths$ identified in (1). Since the relevance values of covered global decisions are set to zero (see l. 5), the derived test suite cannot contain a path twice. Table 3b shows the test cases of a generated test suite with an upper bound $\sigma = 2$ for our running example.

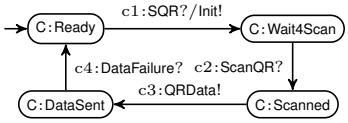
The presented algorithm generally works for every graph with weighted edges. In addition to relevance values resulting from the proposed learning procedure, the tester is thus able to introduce any desired custom weights. The algorithm in Tab. 3a, in particular, generates test cases out of the global relevance function which exclusively comprises specified decisions. Thus, it does not suffer from the major state space blow up resulting from the inclusion of mutations. Though it repeatedly iterates through the state space, the following case study will show that this test case generation is indeed feasible, at least for moderately sized systems.

## 5 Evaluation within a Mobile Payment Application

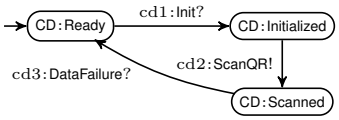
We implemented our risk-based interoperability testing procedure in a research prototype. For evaluation, we applied it to the specification of a mobile payment system provided by an industrial partner.



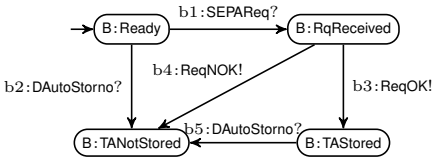

(a) Component UI for the user interface



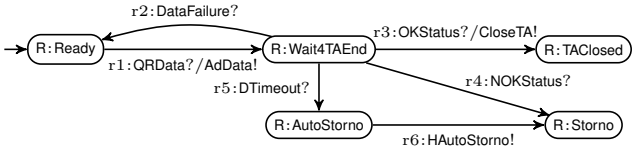

(b) Component C for the cashier



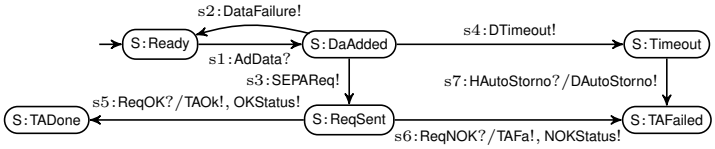

(c) Component CD for the cash desk




(d) Component B for the bank system




(e) Component R for the retailer system




(f) Component S for the service provider system

**Fig. 3.** Component behavior models for the mobile payment case study.

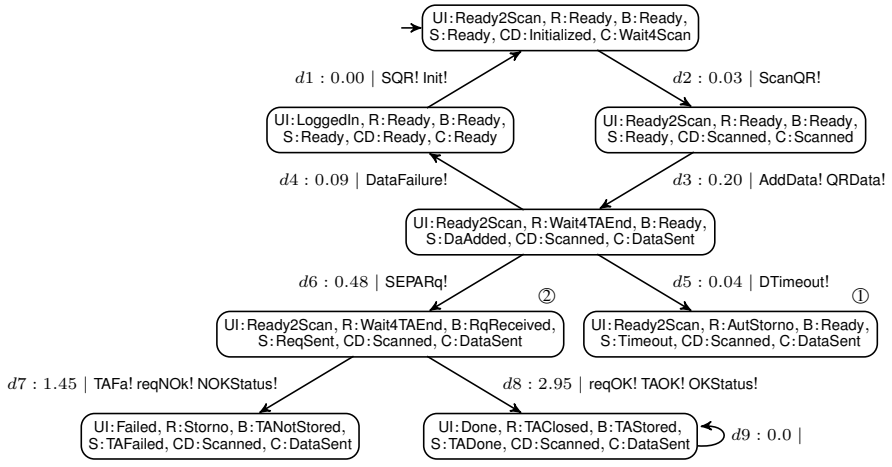



**Fig. 4.** Generated graph of the mobile payment system's global relevance function.

### 5.1 Inputs and Implications for an Optimal Test Suite

Mobile payment systems enable customers to pay goods or services cashless with their mobile phone. We extracted the involved components of such an application and translated them into finite state machines as inputs for our prototype. For conciseness, we thereby focused on the use case of the actual payment process. Figures 3a to 3f show the resulting component models: the *user interface* (typically an app on the user's mobile phone, component UI), the *cashier* (the cashier himself, C), the *cash desk* (the software deployed on the cash desk for processing the payment, CD), the *retailer system* (a central server on the retailer side, R), the *service provider* (a server handling the payment process, S), and the *bank system* (the bank's service for processing a transaction, B). Every action in the presented models is named by a prefix at the message specification, e.g., UI:LoggedIn $\xrightarrow{\text{ui1:SQR!}}$ UI:Ready2Scan. Each component exclusively consumes the messages for which it is authorized. Thus, even though the considered communication over a broadcasting message bus would not be explicitly implemented for a mobile payment system, it does not restrict the presented models in representing the intended communication behavior.

We consider situations to be critical where the retailer assumes the success of a transaction although the bank refused it. In the models, these situations can be described by those system states in which the bank is in B:TANotStored while the retailer system is in R:TAClosed. In such system states, the service provider could be either in S:TAFailed or in S:TADone. Thus, we distinguish the following two negative situations, each one annotated with an exemplary negativity score of $1.0$ ("$-$" indicates that the corresponding component may be in arbitrary state):

1. UI:$-$, C:$-$, CD:$-$, R:TAClosed, S:TAFailed, B:TANotStored
2. UI:$-$, C:$-$, CD:$-$, R:TAClosed, S:TADone, B:TANotStored

For the purpose of this case study, we exclusively considered output faults, i.e., faults in sending messages. Thereby we identified three classes of common faults: (1) message loss, (2) sending of a wrong message, and (3) delay in sending a message. The associated fault models mutate transitions as follows: For (1), message losses, no outgoing message is sent at all. For (2), wrong messages, messages are sent which originally are defined for being sent on other transitions with the same source state as the mutated one. The messages which are defined for being sent on the mutated transition are not sent. Finally, for (3), delayed messages, specified outgoing messages on a transition $t_1$ are shifted to any other transition $t_2$ reachable from the target state of $t_1$. Whilst $t_1$ then sends no message at all, $t_2$ sends a random message out of all delayed messages in addition to the originally defined ones on $t_2$.

Considering the resulting mutants, the defined negative situations are only reachable if the bank system chooses transition $\mathrm{b4}$. If the bank system sends the wrong message ReqOK on $\mathrm{b4}$, the service provider has to enter S:TADone. Then, negative situation 2 occurs, if the service provider sends the specified message OKStatus. Even if the bank system sends the specified message on $\mathrm{b4}$ a negative situation could occur: The service provider has to choose $\mathrm{s6}$ into S:TAFailed; if it thereby sends the faulty message OKStatus, the retailer enters R:TAClosed and negative situation 1 is reached.

**Table 4.** Case study results: (a) Test cases generated for the global relevance function in Fig. 4 with their covered relevance. (b) Results for ten executions with different numbers of simulation runs; the right-most columns show the number of generated test suites that contain both expected test cases of (a) or at least one of them.

| No. | Test case | Cov. relev. |
|---|---|---|
| 1 | $d1 \rightarrow d2 \rightarrow d3 \rightarrow d6 \rightarrow d8 \rightarrow d9$ | 3.68 |
| 2 | $d1 \rightarrow d2 \rightarrow d3 \rightarrow d6 \rightarrow d7$ | 1.45 |
| 3 | $d1 \rightarrow d2 \rightarrow d3 \rightarrow d4$ | 0.09 |
| 4 | $d1 \rightarrow d2 \rightarrow d3 \rightarrow d5$ | 0.04 |

(a) Test cases generated for Fig. 4.

| #Sim. runs | ∅Neg. sit. reached | #Test suites with 2 | #Test suites with $\geq 1$ |
|---|---|---|---|
| 100 | 0.0 | 0 | 0 |
| 1,000 | 0.4 | 0 | 0 |
| 5,000 | 1.7 | 2 | 6 |
| 10,000 | 2.0 | 7 | 9 |
| 100,000 | 33.7 | 10 | 10 |

(b) Results for different simulation runs.

Thus, it can be assumed that composed actions which include $\mathrm{r3}$, $\mathrm{s5}$, and $\mathrm{b4}$ or $\mathrm{r3}$, $\mathrm{s6}$, and $\mathrm{b4}$ are crucial for the reachability of negative situations. These actions, which we are referencing in the following as $\{\mathrm{r3}, \mathrm{s5}, \mathrm{b4}\}$ and $\{\mathrm{r3}, \mathrm{s6}, \mathrm{b4}\}$, should be associated with higher relevance values than the others. The other composed actions cannot directly lead to a negative situation, and their relevance should depend on the length of the paths which are leading to $\{\mathrm{r3}, \mathrm{s5}, \mathrm{b4}\}$ and $\{\mathrm{r3}, \mathrm{s6}, \mathrm{b4}\}$. A risk-optimized test suite should preferably test paths leading through $\{\mathrm{r3}, \mathrm{s5}, \mathrm{b4}\}$ and $\{\mathrm{r3}, \mathrm{s6}, \mathrm{b4}\}$.

### 5.2 Application and Results

We applied our prototype on the inputs described in Sect. 5.1. For this evaluation, we chose a fixed learning rate of $\alpha = 0.1$ and a fixed discount factor of $\gamma = 0.5$. Furthermore, for promoting exploration within the simulation procedure, all agents were reset to their initial states when (1) a negative situation was reached, (2) no negative situation was reachable anymore, or (3) the number of global decisions made since the last reset exceeded the upper bound of 100. Figure 4 shows the graph of the global relevance function generated with 100,000 simulation runs, i.e., sequences of global decisions from the agents' initial states up to the next reset. The graph contains one edge per crucial composed action: edge $d7$ contains $\{\mathrm{r3}, \mathrm{s5}, \mathrm{b4}\}$, $d8$ contains $\{\mathrm{r3}, \mathrm{s6}, \mathrm{b4}\}$. As expected in Sect. 5.1, their relevance is predominant. In correspondence with the system behavior model and the reset conditions mentioned above, the graph includes the following paths:

1. $d1 \rightarrow d2 \rightarrow d3 \rightarrow d4$
2. $d1 \rightarrow d2 \rightarrow d3 \rightarrow d5$
3. $d1 \rightarrow d2 \rightarrow d3 \rightarrow d6 \rightarrow d7$
4. $d1 \rightarrow d2 \rightarrow d3 \rightarrow d6 \rightarrow d8 \rightarrow d9$

Paths 1 and 2 cover the most relevant composed actions $d7$ and $d8$ and thus should be contained in a risk optimized test suite with an upper bound of 2 as test cases. Table 4a shows the actually generated test cases for the global relevance function shown in Fig. 4 together with their covered relevance. The test case order seems to be plausible—for an upper bound of 2 exactly the defined test cases would be chosen.

However, the chosen behavior strategy cannot assure that every possible combination of local decisions is covered during the simulation. Since number and kind of chosen decisions may differ, we can not even assure that different executions of the prototype will lead to the same results. In fact, because of the uniformly distributed behavior policy, it is rather unlikely to get the same absolute relevance values twice. Because of the convergence of learning, the returned values, however, should become the more representative the higher the number of executed simulation runs. To investigate the stability of the results, we executed the prototype several times with the same inputs but a different number of simulation runs, in each case ten times. Table 4b shows the frequency the expected test suite had been generated, the frequency the generated test suite at least contained one of the two expected test cases, and the average number of reached negative situations within the ten executions for the different number of simulation runs. Obviously, the average number of reached negative situations increases with the number of simulation runs. The higher the number of reached negative situations, the more often the expected test suite is generated. For $100{,}000$ simulation runs and an average of $33.7$ reached negative situations, every execution generates the expected test suite. This result implies, on the one hand, that a more focused behavior strategy could be useful. If it would lead to a higher average number of reached negative situations, the expected test suite could be generated constantly for less simulation runs. On the other hand, the proposed approach leads to acceptable results, even if the state space is not fully explored many times. Although the expected test suite had been generated only two times out of ten executions for $5{,}000$ simulation runs, it contained $6$ times at least one of the desired test cases. Such a test suite covers wide parts of the model's relevance.

## 6 Related Work

Our approach combines interoperability testing, risk-based testing, and test case generation with reinforcement learning.

*Machine learning.* The application of *machine learning techniques* on software testing has already been identified as a fruitful perspective by Groce et al. [9]. In [8], Groce uses reinforcement learning via *adaption-based programming* for test input generation. This method rewards coverage increases during test execution to achieve a higher coverage than random testing. By contrast, our method uses reinforcement learning for assessing the criticality of possible faults before test execution. Veanes et al. [17] present a technique inspired by reinforcement learning for choosing coverage optimizing test actions in online testing, i.e., the combination of test generation and test execution in a single algorithm. They also do not consider risk estimations.

*Interoperability Testing.* For generating *interoperability* test cases, it is a common technique to form a system test model by the composition of several component test models. Luo et al. [11] reduce a set of *communicating non-deterministic finite state machines* to a single machine and generate test sequences from this machine. Seol et al. [13] propose a method that composes *input/output state machines* to generate interoperability test cases. Though our algorithm for generating test cases from the global relevance function implements a similar approach, it additionally takes into account the relevance of actions for being tested. In fact, in [11] and [13] the number of generated test cases depends on

the composed, global model's complexity, whereas our approach generates a desired number of risk-optimized test cases.

*Risk-based Testing.* Building on general high-level considerations from authors such as Bach [2] or Amland [1], several methods for integrating *risk estimations* in testing evolved though at different levels of automation. Similar to our approach, several works propose the use of test models for the SuT, which are getting annotated by risk values, for deriving or even generating test cases: Kloos et al. [10] construct test models from the results of a fault tree analysis from which test cases can be generated. Bauer et al. [4] transfer the risk of annotated UML diagrams to a test model, from which test cases are derived. Zimmermann et al. [20] extend this approach by refining the test models so that from these only so-called *critical* test cases are generated. Wendland et al. [19] propose to formulate requirements for the SuT in so-called *integrated behavior trees*. These are annotated with risk values associated with certain risk levels. A risk-optimized test suite is generated from the annotated models by using test directives. In all of these approaches the risk assessment is done by experts. Also our approach builds on expert estimation, since the most critical failure situations have to be given. However, in contrast to the mentioned methods, we automatically derive the contribution of the component's actions to critical situations.

Stallbaum and Metzger [14] note that the *risk assessment* of test cases done by experts could get a critical cost factor. They propose an approach that automates the risk assessment based on requirement metrics. Such metrics refer for example to the revision frequency or the cyclomatic complexity of a use case. However, the determination of risk exposures is still done by experts. The use of metrics for risk estimation in testing was also proposed by Amland [1]. He calculates so called *risk indicators* for every function of the SuT from which the occurrence probability of failures can be estimated. The exposure of possible failures is quantified by expert estimates. No hint is given on how to derive test cases based on these considerations. Since Amland [1] assesses rather the probability and costs of possible failures than a fault criticality, his method could be used for identifying the most hazardous failures together with their deemed effect as input for our approach. Altogether, we assume that metrics are eligible for approximating the occurrence probability of faults in different parts of the system. The assessment of the criticality of faults, however, is hard to determine using code or requirement metrics. Our approach of assessing the criticality of faults by their contribution to the reachability of failures seems more reasonable. In the future, metrics-based approaches could be used to extend our approach with assumed occurrence probabilities of faults.

*Probabilistic Model Checking.* Not only from the strict risk-based testing perspective, the model-based test case generation is an active research topic. Fraser et al. [7] summarize methods which are using *model checkers* for this task. The fundamental idea behind this approach is to formulate logical properties on a model of the SuT in such a way that the counterexamples returned by the model checker can be interpreted as test cases. Closest to our approach are the mutation-based test case generation approaches [7]: Mutations are introduced in the inputs of the model checker according to some fault model and then logical properties are formulated for getting counterexamples representing test cases that kill the mutants. Traditional model checkers, however, only return single counterexamples for absolute properties, such as "The system will never reach a negative

situation", whereas the metrics of fault criticality and action relevance imply the need for quantitative properties, such as "The system will reach a negativity score of $10$ with a probability of at most $0.4$". For evaluating such quantitative properties we would need to use *probabilistic model checkers* which are able to solve verification tasks on Markov-Chains and Markov-Decision-Processes [3]. In fact, we believe that our approach is implementable by such model checkers. The performance could, however, be rather unsatisfactory, since a probabilistic model checker would have to consider the system test model with all of the introduced mutants for evaluating given quantitative properties.

## 7 Conclusions and Future Work

We have presented a risk-based generation procedure for interoperability test cases. It extends behavior models of the SuT with possible faults and assesses them by their criticality w.r.t. reachable failures. An agent-based simulation using the technique of reinforcement learning automates wide parts of this process: Each component of the system is associated with a software agent which learns the criticality of possible faults during a parallel simulation of all agents. The global relevance function is formed by merging the learned criticality values of the agents. Afterwards, Tab. 3a generates a risk-optimized test suite out of the graph of the global relevance function. We applied a prototype on parts of a specification of a mobile payment system. It could be seen that the quality of results increases with the number of executed simulation runs. With $100{,}000$ simulation runs the prototype generated constantly the expected test suite. These observations emphasize the general eligibility of the presented approach.

However, several concepts still can be optimized. More sophisticated agent behavior strategies could lead to the expected results with fewer simulation runs. The agents currently make random decisions between their possible actions thus not always hitting the worst-case behavior of the system. Letting the agents, however, always choose the decision with the highest criticality value, exploration is nearly dropped. Hence, some other strategies should be studied that balance between exploration and exploitation. We exclusively considered the worst-case behavior of the SuT, as we did not assume probabilities for choosing specified actions or for the occurrence of faults to be given. Annotating the models with such probabilities would incorporate such system behavior assumptions and lead to a kind of on-policy learning. To increase the efficiency of our algorithms, we aim to avoid calculating the set of possible global decisions from the local ones. The application of meta-heuristic techniques could further improve the scaling of the test cases generation algorithm on the global relevance function. Further case studies will be made for different system models to identify additional optimizations.

Apart from the definition of behavior models for the components we automatized every activity of the proposed approach. For further automation, these behavior models could also be generated, e.g., from common interface definitions, and the generated test cases could be transformed for a direct import into common test automation systems.

*Acknowledgment.* This research is partly funded by the research project *Testing self-organizing, adaptive Systems (TeSOS)* of the German Research Foundation.

## References


1. Amland, S.: Risk-based testing: Risk Analysis Fundamentals and Metrics for Software Testing Including a Financial Application Case Study. J. Syst. Softw. 53(3), 287–295 (2000)
2. Bach, J.: Heuristic Risk-based Testing. Softw. Test. Qual. Eng. Mag. 11(9) (1999)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT press Cambridge (2008)
4. Bauer, T., Stallbaum, H., Metzger, A., Eschbach, R.: Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest. Softw. Eng. 121, 99–111 (2008)
5. Chen, N.: Passive Interoperability Testing for Communication Protocols. Ph.D. thesis, Université Rennes 1 (2013)
6. Feinberg, E.A., Shwartz, A.: Handbook of Markov Decision Processes: Methods and Applications, vol. 40. Springer Science & Business Media (2012)
7. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with Model Checkers: A Survey. Software Testing, Verification and Reliability 19(3), 215–261 (2009)
8. Groce, A.: Coverage Rewarded: Test Input Generation via Adaptation-based Programming. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 380–383. IEEE Computer Society (2011)
9. Groce, A., Fern, A., Erwig, M., Pinto, J., Bauer, T., Alipour, A.: Learning-based Test Programming for Programmers. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 572–586. Springer (2012)
10. Kloos, J., Hussain, T., Eschbach, R.: Risk-based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In: 4th Int. Software Testing, Verification and Validation Wsh.s (ICSTW). pp. 26–33. IEEE (2011)
11. Luo, G., Bochmann, G.v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method. IEEE Trans. Softw. Eng. 20(2), 149–162 (1994)
12. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the orthogonal. In: Mutation testing for the new century, pp. 34–44. Springer (2001)
13. Seol, S., Kim, M., Chanson, S.T., Kang, S.: Interoperability Test Generation and Minimization for Communication Protocols Based on the Multiple Stimuli Principle. IEEE J. Sel. Areas Comm. 22(10), 2062–2074 (2004)
14. Stallbaum, H., Metzger, A.: Employing Requirements Metrics for Automating Early Risk Assessment. In: Wsh. Measuring Requirements for Project and Product Success (MeReP). pp. 1–12 (2007)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
16. Utting, M., Legeard, B.: Practical Model-based Testing: A Tools Approach. Elsevier (2006)
17. Veanes, M., Roy, P., Campbell, C.: Online Testing with Reinforcement Learning. In: Rev. Sel. Papers 1st Joint Wsh.s Formal Approaches to Software Testing (FATES) and Runtime Verification (RV), pp. 240–253. Springer (2006)
18. Watkins, C.J.C.H., Dayan, P.: Q-Learning. Mach. Learn. 8(3-4), 279–292 (1992)
19. Wendland, M.F., Kranz, M., Schieferdecker, I.: A Systematic Approach to Risk-based Testing Using Risk-annotated Requirements Models. In: 7th Int. Conf. Software Engineering Advances (ICSEA). pp. 636–642 (2012)
20. Zimmermann, F., Eschbach, R., Kloos, J., Bauer, T.: Risk-based Statistical Testing: A Refinement-based Approach to the Reliability Analysis of Safety-Critical Systems. In: 12th Europ. Wsh. Dependable Computing (EWDC) (2009)